Adaptive Victim DRAM Cache: Optimizing Bandwidth for HPC Workloads

ABSTRACT

High performance scientific computing (HPC) applications have high memory bandwidth demands. The Intel Knights Landing platform uses on-package DRAM caches to transparently meet these needs. We model the Knights Landing cache and show that it wastes roughly half the DRAM cache bandwidth on auxiliary accesses. We define a new metric, *access amplification*, to quantify these excess accesses and motivate alternative DRAM cache policies. We propose a new *adaptive victim cache* policy that seeks to reduce access amplification without overwhelming the off-package (main memory) DRAM. Our design uses new on-chip state—laundry list and counts—to manage the DRAM cache as a mostly clean victim cache, but it robustly adapts to store dirty data when needed to reduce off-package DRAM bandwidth.

Using gem5's execution-driven full-system simulation to evaluate HPC workloads, we observe wide variation in miss ratios and read/write ratios including during different applications phases. Our *adaptive victim cache* robustly reduces access amplification to within 7% of an unrealistic SRAM tag design and universally improves performance over the Knights Landing design (on average by 10%).

1. INTRODUCTION

High performance computing (HPC) applications often have high memory bandwidth demands, especially when executed on highly-parallel hardware like GPUs or many-core processors. To meet this bandwidth demand, system designers are including high-bandwidth memory (HBM) on package through 3D die-stacking. Vendors are now shipping chips marketed to HPC with on-package memory (e.g., Intel's Knights Landing [35] and NVIDIA's Tesla P100 [27]). *The main benefit of this memory comes from increased bandwidth, not reduced access latency, since it uses the same underlying DRAM technology*. A straightforward use of this limited capacity HBM is as another level of cache between the on-chip SRAM last-level cache (LLC) and main memory [6, 13, 16, 23, 24, 29] since caches are transparent to programmers.

There are two primary differences between SRAM and DRAM caches that influence DRAM cache design. First, SRAM caches typically separate the tag and state into a different structure from the data [37], but DRAM caches use the same array structure to hold the tags, state, and data. Second, each level of SRAM cache typically has lower latency, but DRAM caches have comparable latency to main memory.

To support high bandwidth, DRAM caches must have much more parallelism than SRAM caches, and interference between the metadata accesses (tag and cache block state) and the data accesses can have a significant performance impact. Many works have shown that interference between DRAM requests hurts performance for main memory [14, 25, 30, 36] and this is exacerbated by auxiliary accesses (e.g., tag, dirty bit reads) in DRAM caches.

We introduce a new metric to quantify this interference: *access amplification*. We define access amplification to be the total memory transactions divided by the number of demand memory requests. Access amplification is similar in spirit to write amplification in NAND-flash disks [12]. The impact of DRAM cache metadata accesses was initially explored by Chou et al. [7] which showed there is bandwidth waste in some DRAM cache designs. Access amplification differs from Chou et al.'s "bandwidth bloat" in two ways. First, access amplification counts memory transactions instead of raw bytes since using ECC to store cache tags requires reading and writing the entire cache block for all accesses. Second, access amplification considers all read and write requests from the LLC "useful", not just read hits. We show that access amplification is a better predictor of performance than bandwidth bloat, especially for workloads with high DRAM cache miss rates.

We use access amplification, the NAS parallel benchmarks, and detailed gem5 simulations [4] to analyze alternative DRAM cache designs. The NAS parallel benchmarks (NPB) are widely used and are considered to be representative of an important class of HPC workloads [1, 3]. For these workloads, DRAM cache miss rates vary from 0% to nearly 100% and write to read ratios vary from 0% writes to 100% writes, sometimes within different phases of a single application. Due to their varying phase behavior and large working set sizes, NPB workloads put different demands on a DRAM cache than general-purpose workloads (e.g., SPEC).

Our analysis starts with the DRAM cache of a commercially available many-core system. Then, we discuss how victim caches can reduce access amplification and potentially improve performance. However, for some workloads, reducing access amplification to the DRAM cache results in excessive traffic to the off-package main memory DRAM. Thus, we introduce an adaptive victim cache design that dynamically balances these competing factors.

KNL-like mostly-inclusive cache Intel's Knights Landing (KNL) [35] is a commercially-available multicore processor that targets HPC workloads. According to published literature, KNL implements a DRAM cache with a mostly-inclusive policy (i.e., DRAM cache evictions do not invalidate the LLC or higher levels of the cache hierarchy). We call our model of this DRAM cache *KNL-like*, since we had to assume certain details not included in the public literature.

Our model implements two demand cache accesses: cache reads that result from LLC-misses and cache writes from LLC-writebacks. HBM accesses occur as follows.

Cache read hits cause one DRAM cache access: ① read the tag & data. The KNL design leverages Qureshi and Loh's Alloy cache which stores the tag and data together in a direct-mapped cache eliminating excess accesses [29].

Cache read misses cause three DRAM cache accesses: ① read the tag & data to check the tag and retrieve dirty data on a miss, ② write the cache tag to mark the frame as busy (KNL uses a "busy" state instead of MSHRs since 1000s of misses may be outstanding [34]), and ③ write the new data once it returns from main memory to maintain inclusion.

Cache writes cause two DRAM cache accesses: ① read the tag & data to check if the cache frame holds dirty data (from the victim block) that must be written back to memory and ② write the new data.

This analysis shows that both cache read misses and cache writes result in excess accesses increasing the access amplification above 1.0. Simulation results show that, on average, access amplification for the NPB workloads is 2.03, or two DRAM cache accesses for each demand request.

Dirty victim cache Changing the cache policy from mostly-inclusive to a *dirty victim cache* reduces access amplification by eliminating the two excess accesses on cache read misses.

Cache read hits are the same as KNL-like.

Cache read misses cause a single DRAM cache access: ① read the tag & data to check the tag. There is no need to reserve a location in the cache since the fill is delayed until LLC writeback.

Cache writes are handled the same as KNL-like. However, the LLC must also writeback clean blocks to the DRAM cache to implement the victim cache policy.

For the NPB workloads, we find that this dirty victim cache design reduces access amplification to 1.75, leading to performance improvements over the KNL cache design by 6%, on average. We leverage a clean-evict bit in the LLC [20] to write clean data at most once.

Clean victim cache A victim cache policy eliminates excess accesses on cache read misses, but cache writes must still check whether a cache frame holds dirty data that must be written back to main memory. These excess accesses can be eliminated by implementing a clean victim cache policy, where the contents of the DRAM cache are always consistent with the main memory.

Cache read hits are the same as KNL-like cache.

Cache read misses are the same as the dirty victim cache.

Cache writes cause a single DRAM cache access: ① write the tag & data. In a clean victim cache, it is always safe to overwrite the cache frame with new tag and data. Like the dirty victim cache, the LLC must writeback clean data to a clean victim cache.

For the NPB workloads, we find that the clean victim design reduces access amplification to 1.24, outperforming the dirty victim cache for workloads with relatively low write traffic. However, because a clean victim cache cannot store dirty data, all LLC-writebacks of modified data must also update main memory. For write-heavy workloads, the limited bandwidth of off-package DRAM becomes a bottleneck, degrading performance toward a write-through cache. For these workloads, despite the very low access amplification a clean victim cache performs worse than a dirty victim cache.

Adaptive victim cache To address differing workload properties, we propose an adaptive victim cache design that seeks to balance the performance characteristics of both the dirty victim cache and the clean victim cache designs. By default, this policy behaves like a clean victim cache, but falls back to a dirty victim cache policy for workload phases with high write intensity.

Our adaptive victim cache design uses on-chip SRAM to track metadata only about *dirty* cache frames, achieving most of the performance of full SRAM tags with less than 3% of the area. We use the following mechanisms.

1. **Laundry counts** track the number of dirty blocks within a cache region. Cache writes to clean regions (i.e., laundry count = 0) do not need to read the tag & data.
2. A **Laundry list** tracks tags of regions with dirty blocks, allowing writes with laundry list tag matches to proceed without first reading the tag & data. This optimizes for a common case in HPC workloads that data is written repeatedly.
3. **Proactive writeback** to keep the cache mostly clean by cleaning the dirty cache segments when sufficient main memory bandwidth is available.

The laundry counts, laundry list, and proactive writeback allow our adaptive victim cache to behave the same as a clean victim cache when the write traffic is low, and like a dirty victim cache under high write traffic workloads dynamically adapting to the workload or the workload phase.

Our proposed adaptive victim cache reduces access amplification from 2.03 in the KNL-like design to 1.35. This is within 7% of an unrealistic SRAM-tag design (1.26). Our design is decentralized, and each HBM channel operates fully independently, requiring 65 KB per HBM channel (total of 1040 KB) of on-chip SRAM (full SRAM-tags require 36 MB). Our adaptive victim design performs at least as well as either the dirty or clean victim designs for each workload. On average, our design increases performance by 10% over the KNL-like design and is within 2% of SRAM tags.

Our contributions are

- A novel design for an adaptive victim DRAM cache,
- The access amplification model for reasoning about DRAM cache policy performance, and
- An analysis of full-scale HPC workloads and realistic multi-channel DRAM cache policies.

2. DRAM CACHE DESIGN AND ACCESS AMPLIFICATION

In our system, we assume there are many CPU clusters, each containing a CPU core, split L1 I/D caches, and a private L2 cache. There is a unified shared SRAM L3 LLC on-chip, split into multiple banks to support high-bandwidth.

	HBM actions	Cache frame state		HBM actions	Main-memory actions	HBM actions
		Tag match	Frame dirty			
LLC-Read	Read tag & data	Hit	-	-	-	-
LLC-Read	Read tag & data	Miss	Clean	Write busy*	Fetch data	Fill*
LLC-Read	Read tag & data	Miss	Dirty	Write busy*	Writeback data, Fetch data	Fill*
LLC-Writeback	Read tag & data*	Hit	-	Fill	-	-
LLC-Writeback	Read tag & data*	Miss	Clean	Fill	-	-
LLC-Writeback	Read tag & data*	Miss	Dirty	Fill	Writeback data	-

Table 1: Details of the KNL-like DRAM cache operation. Non-demand requests are marked with an asterisk. The colors correspond to Figure 2.

Backing this LLC, is a memory-side DRAM cache. Since the DRAM cache is memory-side, it does not participate in the on-chip coherence traffic. All coherence requests are handled by the SRAM LLC. The storage for the DRAM cache is *multiple channels* of high-bandwidth memory (HBM). Addresses are striped across these channels using the lowest-order bits for the highest bandwidth [21]. We consider this HBM as on-package: 3D-stacked with TSVs [5, 15], 2.5D-stacked with a silicon interposer [2, 5], or attached via some other high-bandwidth interconnect [28].

There are two types of requests to the DRAM cache: *LLC-read* and *LLC-writeback*. LLC-read requests are issued to the DRAM cache when an LLC miss occurs, and fetches data from the DRAM cache. In the baseline mostly-inclusive KNL design, LLC-writeback requests are issued when the LLC evicts a modified block.

2.1 Baseline KNL-like DRAM cache

Table 1 shows the detailed function of our baseline DRAM cache design based on public information available for the Knights Landing (KNL) DRAM cache [35]. We assume a latency optimized design like Alloy cache [29] that is direct-mapped and stores the tags with data in the DRAM rows. We store the tags with the ECC data in the DRAM. The JEDEC HBM specification has 16 bits per 128-bit bus for ECC, which allows enough space for 20 bits of tag and metadata and leaves 44 bits for ECC per cache block (e.g., using a (576,532) ECC code) [9, 15, 26].

We implement the "Garbage" (G) state used in KNL [35]. This transient state is used for cases where a DRAM cache line is allocated but the corresponding data is written later [34]. Thus, for every miss to the DRAM cache we must write the block to mark it busy. The KNL designers chose to store the outstanding request state *in the cache* instead of using MSHRs for two reasons. First, to support high bandwidth, 1000s of MSHRs are required, which is costly both in terms of area and power. Second, we find it is very rare for concurrent requests to access the same DRAM cache frame.

The KNL design is a "mostly-inclusive dirty" cache. The cache is allowed to store dirty data, and thus, must write it back to main memory upon eviction. One reason KNL does not use a fully inclusive memory-side design is it requires significant changes to the LLC and directory controllers to

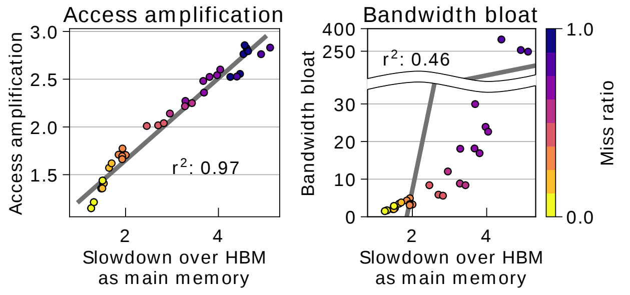

Figure 1: Comparison of measured access amplification to bandwidth bloat. Color represents varying miss rates. At a miss rate of 100% the bandwidth bloat is undefined.

support back-probes on DRAM cache block invalidations. In the case of KNL's distributed tag directory and distributed LLC [35], supporting back-probes would require adding many transient states to support all possible interleavings of simultaneous requests. Back-probes also increase the network traffic on the shared link between the LLC and the DRAM cache, consuming bandwidth that could be used for demand requests. Additionally, although invalidations may not be common, they can result in pathological behavior in a direct-mapped DRAM cache from repeated conflict misses that can further increasing network traffic and using LLC and directory access bandwidth.

2.2 Access amplification

DEFINITION 1 (ACCESS AMPLIFICATION). *The total number of DRAM cache accesses divided by the demand accesses initiated by the SRAM LLC misses.*

In Table 1, the actions marked with an asterisk are non-demand requests. For instance, on an LLC-read, the demand part of the access is the initial DRAM cache read to get the tag and data (no asterisk). The *write busy* and *fill* accesses are not servicing the original demand request, but performing auxiliary cache operations (asterisks). Table 2 compares the access amplification of the KNL-like cache with a nearly-ideal but infeasible-to-implement SRAM tag design. Each row in the table shows the counts of each action (e.g., derived from Table 1). The demand accesses from the SRAM LCC are all of the requests which are issued to memory in the absence of a DRAM cache.

Access amplification builds off of the ideas of bandwidth bloat as pioneered by Chou et al. [7] and write amplification in flash memories [12]. Both access amplification and bandwidth bloat measure the wasted DRAM cache bandwidth, but bandwidth bloat is defined as the total bytes transferred divided by the useful bytes (i.e., bytes serviced from the DRAM cache). We believe *transactional accesses* are more indicative of the performance overhead than the raw bytes, similar to write amplification (actual number of flash page writes divided by user page writes). Because the cache tags and metadata are encoded in the extra bits provided for ECC, we must read and write the entire cache block on each access. Additionally, DRAM performance is affected more by interference from different accesses on shared resources (e.g., the shared data bus, bank row buffer) than the raw number of

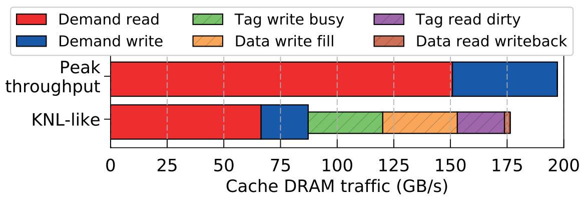

Figure 2: Breakdown of DRAM cache traffic between demand accesses and other overheads (hatched). This figure assumes a 50% miss ratio, 50% reads, and 50% dirty blocks. The results are similar with other miss and write ratios.

bytes read and written. This dichotomy is similar to using input/output operations per second (IOPs) instead of bandwidth to measure the performance of storage devices.

Figure 1 quantitatively compares these two metrics showing how the access amplification (left) and bandwidth bloat (right) correlate to the slowdown of the KNL DRAM cache design compared to directly accessing HBM as if it was main memory. We use a microbenchmark with varying DRAM cache miss ratios (color). The key quantitative difference is that bandwidth bloat significantly increases when the hit rate is low. In fact, if the hit rate is 0% (e.g., a streaming workload), the bandwidth bloat is infinite. Access amplification, on the other hand, shows a nearly linear relationship with execution time slowdown (execution time when using HBM as main memory divided by execution time when using HBM as a cache). Thus, we believe the transaction semantics of access amplification provide a better framework for comparing different DRAM cache policies. The goal of access amplification is not to predict performance, but to give designers a model to reason about the performance of different DRAM cache designs.

Figure 2 shows the potential performance impact of access amplification. To illustrate the effects of access amplification, we modeled the HBM based on the high bandwidth memory standard [11, 15]. We used an artificial microbenchmark to send traffic to the HBM. Figure 2 shows that over 50% of the DRAM cache traffic is wasted (thin, hatched bars) not servicing demand requests (thick bars). Colors correspond to the actions in Figure 1.

3. USING THE ACCESS AMPLIFICATION MODEL

We previously introduced the access amplification of the KNL-like and victim cache designs. In this section, we specifically quantify the access amplification by considering the impacts of two different design parameters, the *insertion policy* (either a mostly-inclusive cache or a victim cache) and the *writeback policy* (whether to allow the cache to contain dirty data or have a fully-clean cache).

Table 2 summarizes the access amplification of unrealistic SRAM tags, the KNL-like DRAM cache, and three victim cache designs. This table presents the access amplification in relation to the program characteristics including number of reads, writes, and misses at the DRAM cache.

3.1 KNL-like → Dirty victim

Using a victim cache design eliminates the extra DRAM cache write access marking the cache frame as busy while the miss is outstanding. Instead of allocating the cache frame at the time of the miss, a victim cache waits until the LLC evicts the block to allocate a frame. This removes $2m$ (m is the miss ratio) DRAM cache writes on each LLC-read (dirty victim row in Table 2).

The victim cache policy also eliminates an extra DRAM cache write access if a block is modified after a DRAM cache miss (part of $2m$ in Table 2). In the KNL cache, when an LLC-writeback occurs it overwrites the filled data, and the fill is "useless" as it is never read before it is overwritten.

However, a victim policy requires filling on every LLC eviction which significantly increases the number of LLC-writeback requests (W_u). A traditional victim cache policy [19] would maintain exclusion between the LLC and DRAM cache, requiring every LLC eviction to write back data to the DRAM cache. To reduce the bandwidth requirements, we enforce a non-inclusion property using a "clean-evict" bit [20]. The LLC maintains a per-block clean-evict bit, which is set when a block is fetched from main memory (and not on DRAM cache hits). When the LLC evicts a block, only blocks with dirty data or a set clean-evict bit are written to the DRAM cache. The clean-evict bit differs from Chou, et al.'s DRAM cache presence (DCP) bit [7] because DCP must be maintained exactly (requiring complicated back probes) while clean-evict is essentially a performance hint. Using the clean-evict bit, $W_u <= (R \times m)$.

3.2 Dirty victim → Clean victim cache

A clean cache, such as a conventional writethrough cache, does not need to check the tags before writing the tag and data. Therefore, we can reduce the access amplification of the dirty victim cache by treating it as a writethrough (clean) design.

The "Clean victim" row in Table 2 shows there are fewer DRAM cache accesses on LLC-writeback requests for the clean victim design than the other designs. The number of cache accesses reduces from more than $2W$ per access to just W for the clean victim design. This implies that the clean victim cache will perform better for high write traffic workloads. However, writethrough caches have a major drawback: all dirty data must be written to main memory, increasing the main memory traffic.

In Section 5 we find that implementing a fully-clean DRAM victim cache eliminates the most access amplification, even more than unrealistic SRAM tags. However, a clean victim cache hurts performance for some applications (Figure 9) since it increases the traffic to main memory, which is the bottleneck in write-heavy applications.

4. ADAPTIVE VICTIM CACHE

Ideally, we want an adaptive cache design that behaves like a fully-clean victim cache, without the downsides of squandering main memory bandwidth. Importantly, under a high-write load to the DRAM cache, we want the cache to behave like a writeback cache, not a writethrough cache.

The "Adaptive Victim" row in Table 2 shows the goal for our design. By leveraging the victim cache policy, we remove all unnecessary accesses on the LLC-read path, and by falling back on a writeback (dirty) policy, there is no main memory

Design	DRAM cache accesses
Demand accesses	$R+W$
Unreal. SRAM Tags	$R(1+m_d)+W(1+m_d)$
KNL-like	$R(1+2m)+2W$
Dirty victim	$R+2W+W_u(1+m)$
Clean victim	$R+W+W_u$
Adaptive victim	$R+W(1+m_d)+W_u(1+m_d)$

Table 2: Access amplification for each design. R is the reads. W is the writes. m is the miss rate of the DRAM cache. m_d is the rate of dirty data on each miss. And W_u is the unmodified writebacks. Access amplification is $accesses/(R+W)$.

access amplification. In the best case, we only want to have extra reads to the DRAM cache if we are sure the data is dirty, which is the same overheads as the full SRAM tag design.

Our adaptive victim design has about the same access amplification as the unrealistic SRAM tag design. The key idea is that we can get most of the benefits of SRAM tags by only storing tags that reference *dirty data* in SRAM. To limit the overhead of storing these tags, under high write conditions we fall back on the dirty victim design, and to increase performance when the main memory traffic is low we upgrade to the clean victim design. Thus, our design robustly performs as well as either the dirty or clean victim caches, as shown in Section 5.

In this section, we explain five insights driving our adaptive victim cache design. We first present a simplified design of our adaptive victim policy. Then, we extend this design to large multi-channel DRAM caches. We then discuss the implementation of the adaptive victim policy, an important optimization, and the area overheads of our proposal.

4.1 Simplified adaptive victim design

Figure 3 shows a simplified example with a 4-entry DRAM cache to illustrate these insights. Step 0 shows the initial state of the cache with four valid and clean frames.

Insight 1: *If there are no dirty blocks in the cache, then it is safe to overwrite a block without first reading the cache tag.*

Action: Store a count of the number of dirty cache frames in SRAM. If the count is zero, it is safe to overwrite. Otherwise, we must check before writing.

Example: In step 1 of Figure 3, the LLC sends an writeback request to the DRAM cache with the clean-evict bit set. The laundry count of zero signifies there is no dirty data in the cache. Thus, we can treat the cache as though it is fully-clean and write the incoming data without first reading the tag and data in the corresponding cache frame since it will only overwrite clean data.

In step 2 of Figure 3, the LLC sends a writeback request with modified data. This implies the data in main-memory is stale, and when the line is inserted into the cache, the cache will contain dirty data. Thus, when inserting the line, we increment the laundry count by one. This signifies it is no longer safe to write into the cache without first reading the corresponding frame's tag (like the dirty victim design) since it is possible a new line will overwrite the only copy of the data for a cache block.

Insight 2: *If we know all of the dirty blocks in the cache have the* same *tag as the current write, then we know it is safe to write the block without first checking.*

Action: Track the tag shared by all dirty blocks. If the tag matches, then it is safe to write. Otherwise, we must invalidate the tag and check before writing.

Example: In step 2 of Figure 3, when writing the dirty data for line $A2$, we also insert the tag (A) into the laundry list structure. A in the laundry list implies that only blocks with a valid tag of A can be dirty in the cache. Blocks with any other tag *must be* clean.

In step 3 of Figure 3, the LLC sends a modified writeback for $A0$. The laundry count is not zero; therefore, it may be unsafe to write this block into the cache. However, since the tag in the laundry list matches the tag of the writeback request, it is safe to overwrite the data in the cache. Either we will overwrite clean data that is found in main memory, or we will safely overwrite stale data with a more up-to-date version. In this step, we also increment the laundry count to match the number of dirty frames in the cache.

Finally, step 4 of Figure 3, shows a modified LLC-writeback to a block with a *different* tag (B). In this case, it is unsafe to write the data into the cache, so before writing the data, we must check the tag of the corresponding cache frame (exactly like a dirty victim cache). Additionally, when inserting the block in a dirty state, we must invalidate the laundry list entry for A since it is no longer safe to write blocks from A into the cache and increment the laundy count.

4.2 Applying insights 1 & 2 to large caches

Insights 1 and 2 do not directly scale to large cache sizes. These insights quickly break down since even a small percentage of dirty lines cause the cache to fall back on the dirty victim policy.

To extend these insights to large caches, we first define a cache super-frame as an aligned set of cache frames in a direct-mapped cache. Super-frames are similar to super-blocks; however, super-blocks are aligned regions of *memory*, and super-frames describe a region of cache storage. A super-frame can hold data from a single super-block or from multiple super-blocks since multiple super-blocks map to the

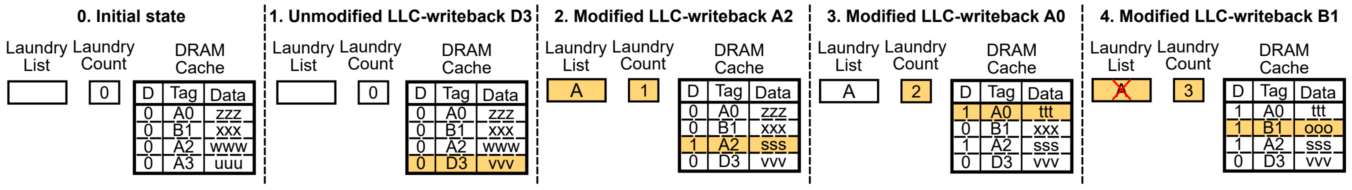

Figure 3: Example laundry count and laundry list usage. A simplified four entry cache is shown.

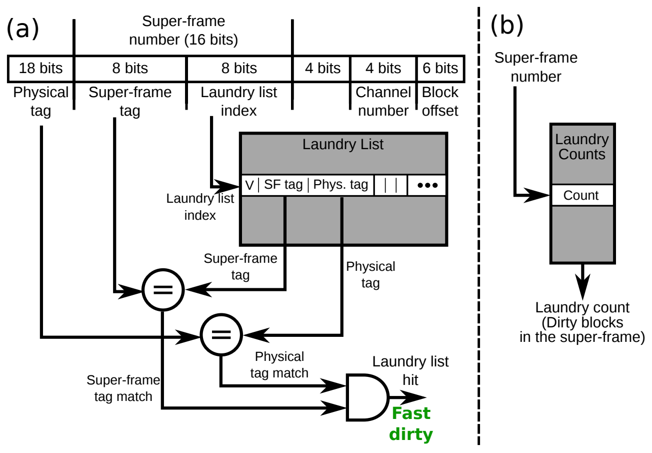

Figure 4: Operation of the laundry list (a) and laundry counts (b) structures.

same super-frame.

Insight 3: *We can partition the cache into super-frames and apply insights 1 and 2 on many small cache regions (super-frames) instead of the entire cache.*

Action: Each super-frame has a dirty count stored in the *laundry counts* and a tag stored in the *laundry list* (see Section 4.3). It is safe to write a block into the cache if the super-frame's laundry counts entry is zero *or* there is a tag match in the laundry list. Else, we must check the cache before overwriting the cache frame.

Above, we described the adaptive victim design as if it was a centralized structure. However, to support high bandwidth, die-stacked DRAM is often split into many channels (e.g., 16 channels for the HBM standard [15]). These channels operate independently and may not be co-located on chip. Thus, we do not assume a centralized laundry list and laundry counts, but split the structures across the channels into multiple banks. Each bank of the laundry structures tracks the subset of the super-frame assigned to that channel (super-frames are continuous in physical space). Figure 4a shows the physical address bits used to access the laundry counts and laundry list structures. Banking the laundry structures across multiple channels increases their area overhead (e.g., the tags in the laundry list may be replicated), but it allows these structures to flexibly support high bandwidth.

4.3 Laundry counts and laundry list

The laundry counts is an SRAM-based structure that tracks the number of dirty frames in each cache super-frame (Figure 4b). The laundry counts is a directly indexed array with one entry per super-frame in the cache. The count is updated on each DRAM action by incrementing when adding dirty data and decrementing when flushing dirty data to main memory (Figure 5 contains more details).

We choose to track super-frames instead of every cache frame to decrease the area overhead of the laundry counts. Tracking dirty information using a bitvector for each 64-byte block requires 2 MB of on-chip SRAM area for a 1 GB DRAM cache. Section 5.6 shows that HPC workloads have high spatial locality in the DRAM cache so a bitvector will not significantly improve performance over coarser super-frame tracking.

We require the laundry count to be consistent with the dirty bits in the cache. Thus, we cannot increment the laundry count if the modified LLC-writeback overwrites already dirty data. To facilitate dirty block tracking, we add another bit to each LLC-entry. On a response to an LLC-read hit at the DRAM cache, we include a cache-dirty bit which is set if the block is dirty in the DRAM cache. Like the clean-evict bit, this bit is tracked at the LLC and sent back to the DRAM cache on an LLC-eviction. If the cache-dirty bit is set, the line is already dirty in the cache, and we do not increment the laundry count.

Insight 4: *Insight 2 is not required for correctness. We do not need to track the tag for every super-frame in the cache.*

Action: Reduce the entries in the laundry list to only cover a subset of the super-frames in the DRAM cache and make the laundry list set-associative to reduce the conflicts between dirty super-frames.

Thus, the *laundry list* stores only *some* tags in SRAM to eliminate the DRAM cache reads on LLC-writeback in the case where the tag matches (Insight 2). Now, when checking if an LLC-writeback is safe, if the laundry counts entry is nonzero (i.e., the super-frame frame contains dirty data), we check the laundry list. If the laundry list contains a tag for that super-frame *and* that tag matches, then the LLC-writeback is safe to insert into the cache. Section 5.5 shows it is common to overwrite dirty data in the cache and find a matching tag in the laundry list. Thus, we find the laundry list significantly decreases the extra reads compared to only tracking the laundry counts.

To reduce the area overhead, we implement the laundry list as a set-associative array with fewer entries than super-frames in the cache. Each laundry list entry holds the physical tag that is found in the cache, the super-frame tag, and a valid bit.

The laundry list is indexed by a subset of the super-frame number (Figure 4a). On a laundry list access, the valid super-frame tags from each way are compared with the incoming request's super-frame tag. If any of these super-frame tags match, we compare the physical tag of that entry to the request's physical tag. On a physical tag match, we take the "fast dirty" path in Figure 5. If there are no matching super-frame tags or the physical tags mismatch, we take the "slow dirty" path in Figure 5 since we are unsure whether the request is safe to write into the super-frame. In this case, the adaptive victim cache behaves like a dirty victim cache.

In Section 5.5, we show that a laundry list that covers only $^1/_8$ of the DRAM cache performs as well as a full laundry list for most workloads. The key reason a partial laundry list provides sufficient performance is that most super-frames are fully clean and contain no dirty data. Thus, tracking a tag for each super-frame is unnecessary.

The dirty region tracker (DiRT) proposed by Sim et al. [33] has a similar goal to the laundry list and laundry counts. However, DiRT *constrains* the number of dirty pages, falling back on a writethrough (clean) policy under high-write traffic, which can hurt performance (evaluated in Section 5). Our adaptive victim design is more robust and falls back on the higher-performance dirty victim design under high write-traffic.

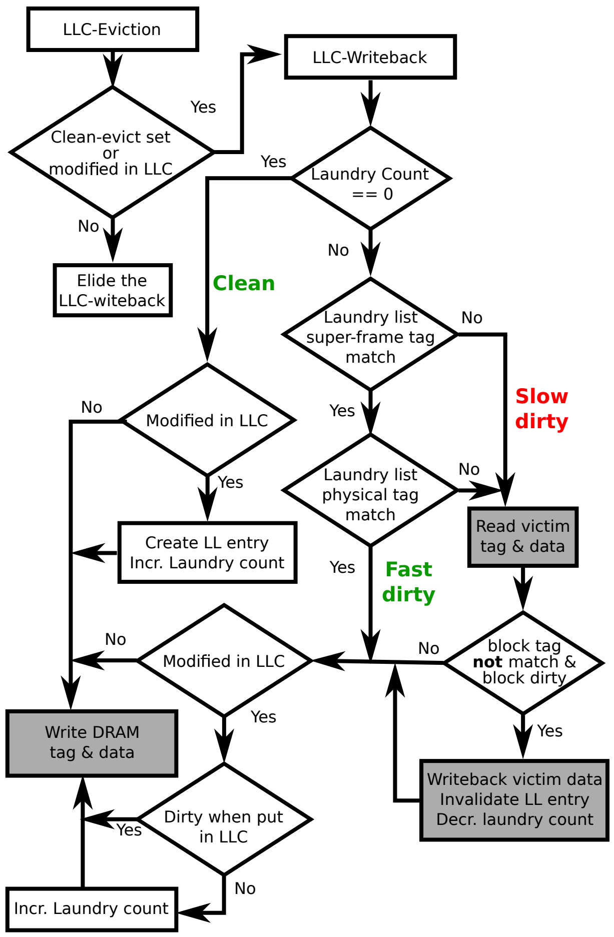

Figure 5: Overview of adaptive victim cache operation. The shaded rectangles represent accessing the DRAM cache. Corner cases and error cases are covered in detail in Table 5 in Appendix A.

The flowchart in Figure 5 shows the detailed DRAM cache state machine for LLC-writeback requests. The shaded rectangles represent accessing the DRAM cache. The "clean" and "fast dirty" path allow the adaptive victim cache to behave like a fully-clean cache and an unrealistic SRAM tag cache, respectively. LLC-read requests are not shown; they simply read the tag and data from the DRAM cache.

Before the LLC-writeback request, the LLC checks the clean-evict bit and LLC's dirty bit. If neither bit is set, the LLC-writeback is elided since the block is likely already cached and main memory or the DRAM cache contains the most up-to-date data.

Otherwise, the LLC sends an LLC-writeback request with the modified bit, the clean-evict bit, and the cache-dirty bit. On an LLC-writeback, we first check the laundry count for the super-block corresponding to the writeback request. If the count is zero, then it is safe to write the line into the super-frame and the super-frame behaves like a fully-clean cache (without requiring all dirty data to write through). Otherwise, we check the laundry list as shown in Figure 4. If there is a laundry list hit, then we take the "fast dirty" since the access is guarenteed to be safe.

The access *may* overwrite data that is stale in main memory if either there is no laundry list entry for the corresponding super-frame or the physical tag of the matching super-frame entry does not match. In this case, we fall back on the behavior of the dirty victim cache and read the tag and data in the DRAM cache. At this point, if the corresponding frame's data is dirty and the tag does not match the request, we write the data back to main memory.

Finally, we write the data in the LLC-writeback request to the DRAM cache. If we used the "slow dirty" path, we use the canonical information in the DRAM cache and the LLC-writeback request type to update the laundry structures to be consistent with the DRAM cache. Otherwise, we update the laundry structures based on the type of LLC-writeback request since we know the state of the cache frame.

4.4 Proactive writeback

Insight 5: *If the cache is mostly clean, we get more benefit from our adaptive victim design.*

Action: Proactively writeback dirty data to main memory *only when it will not hurt performance*.

The adaptive victim DRAM cache will perform best when most of the cache is clean. However, maintaining a fully-clean victim cache hurts performance when the backing main memory bandwidth is saturated. Therefore, we use a proactive writeback design that adaptively cleans blocks in the cache *only when it will minimally affect performance*.

There are two times where we *try* to writeback dirty data from the DRAM cache to the main memory. First, on a modified LLC-writeback we try to write the data through to memory as well as insert it into the cache like the clean victim design. Second, whenever a cache frame is read on an LLC-read request, if it is dirty, we try to clean the frame.

We only issue these proactive writebacks if the main memory bandwidth is not at or near saturation. We track whether or not we are receiving backpressure from main memory and never send a proactive writeback if the main memory port is full or if main memory has been blocked recently (e.g., the last 50 cycles). If we continue to issue writes when the main memory write buffer is full, the writes will take precedence over the main memory reads, hurting performance because main memory reads are the latency-sensitive demand requests from the CPU.

In addition to proactive writeback, we propose a dirty-data scrubber with the DRAM cache controller. This hardware walks the laundry counts structure and issues reads to proactively clean super-frames when there is extra DRAM cache and main memory bandwidth (e.g., when the application is not in a memory-intensive phase). In fact, this scrubber could work together with the DRAM refresh logic to proactively clean DRAM rows that are activated for refresh. We do not present an evaluation of the dirty data scrubber.

Proactive writeback is similar to self balancing dispatch (SBD) proposed by Sim et al. [33] except we only bypass LLC-writeback requests. Additionally, the proactive writeback algorithm is simpler than SBD. Proactive writeback only tracks the backpressure and number of writes, unlike SBD which calculates the expected queuing delay.

4.5 Adaptive victim cache overheads

Name	Class	Footprint	Description
BT	D	10.8 GB	Block tri-diagonal solver
CG	D	16.3 GB	Conjugate gradient
FT	C	5.1 GB	Discrete 3D FFT
IS	D	33.0 GB	Integer sort
LU	D	9.0 GB	LU Gauss-Seidel solver
MG	D	26.5 GB	Multi-grid on meshes
SP	D	16.0 GB	Scalar pentadiagonal solver
UA	D	9.1 GB	Unstructured adaptive mesh

Table 3: NPB version 3.3.1 workloads. Class defines the input size. Footprint is the actual resident memory size.

The total overhead of the adaptive victim caches structures is 1040 KB split evenly between 16 fully-independent HBM channels (65 KB per channel) compared to 36 MB for full tags for a 1 GB DRAM cache. Although this is a significant area overhead, it is less than 6% of the 16 MB LLC, and due to our decentralized design, each individual structure is small, so there is little added latency and power.

We use a super-frame size of 16 KB, split across 16 HBM channels. Thus, each dirty count array entry tracks 16 64-byte frames (1 KB per channel). The laundry count requires 5 bits for each entry, and so, the laundry count requires 40 KB per channel (640 KB for the whole system). We use a laundry list that can cover $^1/_8$ of the DRAM cache capacity, assuming high spatial locality. The laundry list has 8192 entries per HBM channel. We analyze the effect of increasing the laundry list size in Section 5.5. The area overhead of the laundry list is 25 K per channel.

5. EVALUATION

5.1 Methodology

DRAM caches make the most sense for workloads that are bandwidth-bound and have very large working set sizes. Therefore, we evaluate the NAS parallel benchmarks (NPB) [1, 3]. These are scientific computing benchmarks with large working set sizes and high memory traffic. Table 3 describes the workloads. These workloads run between a few minutes and a few hours, natively.

We run the NPB on gem5 [4]. However, NPB are too large to execute to completion in simulation. We use random sampling from the whole execution to ensure capturing all of the workloads' variation. We use simple random sampling and a method similar to the one used by Sandberg et al. using the virtualization hardware (KVM) to fast-forward to each sampled observation [31]. For each observation, we warm up the caches for 10 ms and run the detailed simulation for 2 ms. We have at least 100 observations for each configuration and application totaling about 200 ms of detailed simulation per workload per configuration (not counting cache warmup) or about 6 billion instructions for each sample.

Performance for NPB are measured in average billion floating point operations per second (GFLOPS) or user-mode instructions (UGIPS) for integer-only workloads. Measuring the performance with GFLOPS or UGIPS provides a fair comparison across configurations when using statistical sampling as the total work accomplished in the observation period is

On-die	**Memory system**
32 CPU cores 32 KB each split L1 I/D 256 KB private L2	16 HBM channels [15] 8 GB/s per channel 1 GB total capacity
16 MB shared LLC 16 LLC banks	2 DDR3 channels [14] 12.6 GB/s per channel 64 GB total capacity
10 cycles laundry counts + laundry list latency	

Table 4: Simulated system details.

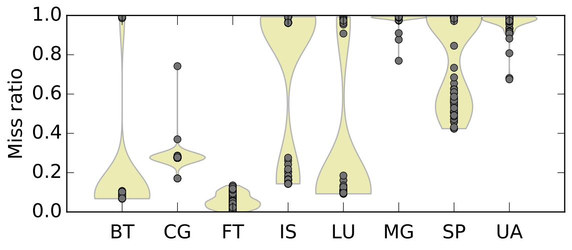

Figure 6: Miss ratio for NPB. Each point represents one random observation. Also shown is a Gaussian approximation of the distribution across the whole sample.

proportional to the number of floating point or user-mode instructions executed.

Table 4 describes the system we used to evaluate the NPB. We simulated a large multi-core CPU with abundant thread-level parallelism and a high-bandwidth cache system. The on-die CPU system is a proxy for a high-performance compute chip. We use the detailed DRAM models in gem5 to model both HBM and DDR3 [11]. These models are based on the datasheets for each type of memory. When modeling HBM as a DRAM cache, we add one extra BURST cycle to model ECC/tag transfer.

We evaluate four different DRAM cache designs.

KNL-like (2.1) Latency-optimized DRAM cache design.
Dirty victim (3.1) Victim cache that can store dirty data.
Clean victim (3.2) Victim cache that is fully-clean.
Adaptive victim (4) Victim cache that is mostly clean, but allows any amount of dirty data.

5.2 Workload analysis

The NPB show significant variation both across workloads and within the same workload, which motivates our adaptive victim cache design. Figures 6 and 7 show the miss ratio and write ratio for each 2 ms observation for each workload. The background area plots are violin plots and show a Gaussian approximation of the distribution. This shows the weights of many observations that show the same value (e.g., most of the BT observations show a miss rate of about 10%, but a few have a miss rate of nearly 100%).

There are a variety of behaviors across the NPB. Figure 6 shows the wide variety of DRAM cache miss ratios, even within a single workload. For instance, FT always has a low miss ratio, but IS sometimes has a low miss ratio and sometimes has a miss ratio near 100%.

Figure 7 shows the ratio of dirty write requests (roughly the percent of DRAM cache that holds dirty data for a victim cache) for the NPB. This figure shows there is a variety

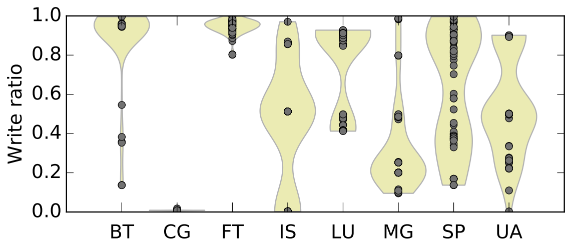

Figure 7: Percent of all LLC-writbacks that contain modified data. Each point represents one random observation. Also shown is a Gaussian approximation of the distribution across the whole sample.

of write behavior, sometimes even within a workload. For instance, CG shows almost no writes to the DRAM cache, but SP has a broad distribution with phases showing high write proportions and low write proportions. Thus, *it is important for a DRAM cache policy to be robust under many different application characteristics*.

5.3 Access amplification

Figure 8 shows the access amplification—the number of DRAM cache accesses per demand request from the on-chip LLC (demand reads and writes)—for the eight NPB evaluated. Any accesses above the 1.0 line are "unnecessary" accesses that are not servicing demand requests. Each bar is split by the different types of access amplification described in Section 2.2. The *data read writeback* and *tag read dirty* are combined into a single bar in this figure as each *tag read dirty* access reads both the tag and the data to reduce latency.

Figure 8 shows that the KNL-like design makes on average two DRAM cache accesses per demand request. In the worst case (MG), this design has an access amplification of 2.8. MG has a high miss rate, and each miss requires three DRAM cache accesses (Section 2.2).

The dirty victim design reduces the access amplification by removing the *tag write busy* accesses. However, this design increases the relative number of *data read writeback* accesses since the LLC performs must writeback both modified data and clean data if the clean-evict bit is set.

The clean victim design further reduces the access amplification. This design trades the demand write requests for *data write fill* accesses, since for every LLC-writeback this design fills a clean copy of the data. Although the clean victim design eliminates most access amplification, for some workloads it does not perform well because it increases main memory traffic (Section 5.4).

Next, the unrealistic SRAM tag design shows the minimum access amplification while still achieving good performance. This design significantly reduces the access amplification compared to the dirty victim design because, the cache controller knows exactly which accesses are required before accessing the DRAM cache.

The rightmost bar shows the access amplification of the adaptive victim design is only slightly higher than the unrealistic SRAM tag design. Like the SRAM tag design, the adaptive victim cache limits the unnecessary accesses and, in most cases, only accesses the DRAM cache when required. The adaptive victim is within 6% of the amplification reduction of unrealistic SRAM tags.

Like the clean victim design, the adaptive victim design trades some demand write (dirty write) accesses for *data write fill* accesses. In the cases where there is ample main memory bandwidth, the adaptive victim cache uses proactive writebacks perform a clean fill instead of a dirty write. For some workloads like SP and MG, there is not any spare main memory bandwidth, so the adaptive victim design rarely issues proactive writebacks (the number of demand writes is the same between the dirty victim design and adaptive victim design). BT and FT have many proactive writebacks and some of their demand writes are replaced with *data write fill* accesses (clean writes).

5.4 NPB performance

Figure 9 shows the overall performance for the eight NPB evaluated. The bars represent the average performance (either GFLOPS or UGIPS) of each application across the observations taken. The error bars show a 95% confidence interval for the mean performance.

Some applications see no performance improvement or only a minor performance improvement with a 1 GB DRAM cache (IS, UA, MG) compared to no DRAM cache. For MG and UA, the miss rate is very high: an average of over 90% (Figure 6). For IS, some iterations fit in the 1 GB cache and show low miss rates, but most do not and these iterations that do not fit in the cache dominate the execution time. For UA, even if all of the memory is HBM, there is not much performance improvement. This workload is not bandwidth-bound, but latency-bound due to pointer-chasing through the unstructured grid datastructure.

Across the NPB, some workloads show higher performance with a dirty victim DRAM cache (BT, FT, LU, MG, SP) and some perform better with a clean victim design (IS, CG). The adaptive victim DRAM cache design performs at least as well as the best of either victim design. In most cases, the adaptive victim design performs as well as the unrealistic SRAM tag design (not shown). The only case the unrealistic SRAM tags outperforms the adaptive victim is FT, where there is a very high write-ratio. In this case, the adaptive victim performs the same as the dirty victim design.

The lower-right graph shows the average relative performance of the four DRAM cache designs with equal weight between each application. Our adaptive victim design shows an average of about 10% performance improvement compared to the baseline KNL-like design. This also shows that the KNL-like design performs worse than a simple victim cache design, and the victim clean design hurts performance, on average. Our adaptive victim cache design is only 2.3% slower than an impractical design with all tags stored in SRAM.

5.5 Laundry list size

Recall that an unrealistic SRAM tag designs requires at least 36 MB of on-chip SRAM to store the cache tags. Our adaptive victim design requires about 1 MB of on-chip SRAM. We find that increasing the on-chip storage does not improve the performance of our adaptive victim cache.

Figure 10 shows the relative performance of each application for varying sizes of the laundry list. The "full tags" is provisioned with an entry for every super-frame, this would

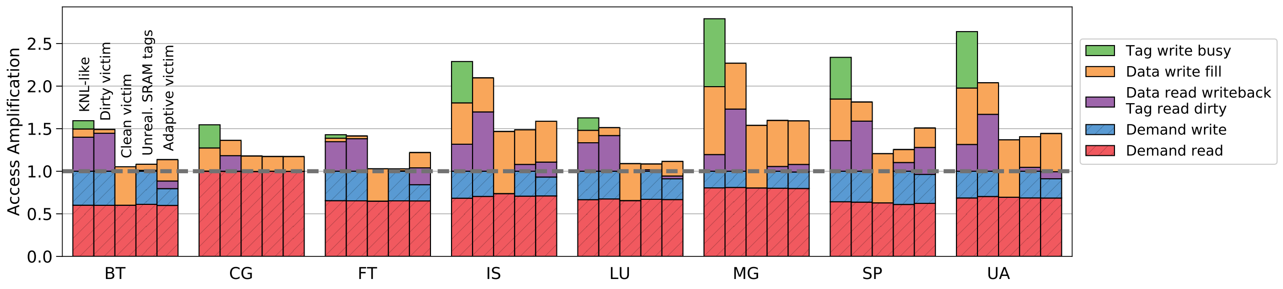

Figure 8: Access amplification for the NPB. The dashed line at 1.0 shows the demand requests from the LLC. Any accesses above 1.0 are due to access amplification. Note: there is some noise in the data from the sampling methodology.

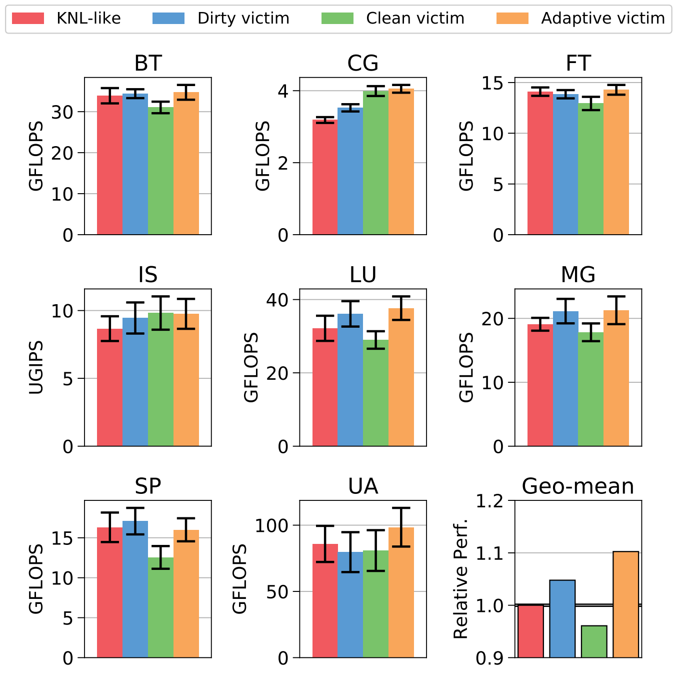

Figure 9: Performance of NPB. All workloads measured in GFLOPS, except for IS with UGIPS. The error bars represent a 95% confidence interval for the mean based on the Student's t-Test. The average assumes equal weight for each of the workloads.

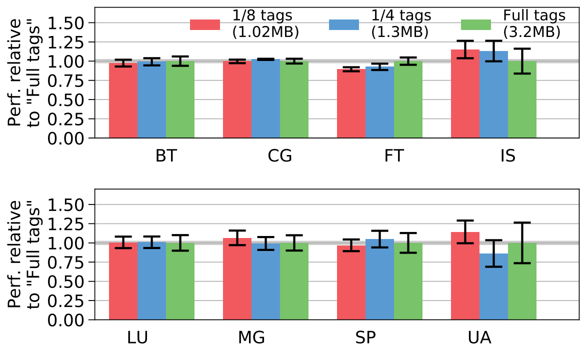

Figure 10: Relative performance of different tag array sizes for the laundry list. Error bars show a 95% confidence interval for the mean, and all bars are normalized to the average performance with "full tags".

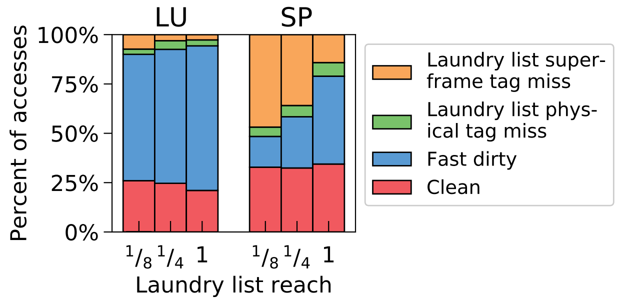

Figure 11: Breakdown of laundry list and laundry counts lookups for all write requests for two representative applications, FT and LU. Compares a laundry list with $^1/_8$ reach, $^1/_4$ reach, and large enough to cover the entire cache (impractically large). Labels correspond to Figure 5.

have an overhead of 3.2 MB. The "$^1/_8$ tags" is the design used in the previous results (1040 KB), and the "$^1/_4$ tags" shows the performance if the tag array is provisioned so at most $^1/_4$ of the cache is allowed to be dirty. The "$^1/_4$ tags" design has an area overhead of about 1.3 MB.

Figure 10 shows that for most applications the size of the laundry list does not significantly affect performance. Although some bars for smaller laundry list designs appear higher, they are within the noise of our sampling methodology. FT is one workload where there is a clear effect when the tag array size is increased. However, even for FT, the "$^1/_8$ tags" performs as well as the dirty victim design (Figure 9) demonstrating the robustness of our design. In the worst case, it falls back on the dirty victim performance, not the clean victim design.

Figure 11 shows the breakdown of the result of consulting the laundry structures for each write request as a percentage of all writes. The labels correspond to the flowchart in Figure 5. This figure shows the importance of tracking tags in the laundry list instead of just tracking the whether the superframe is dirty (i.e., just the laundry counts). The goal of the laundry list is to avoid the "slow dirty" path which reduces the number of DRAM cache reads before writing new data, since reading before writing incurs extra latency and occupancy overhead on the DRAM bus [14]. The sum of the red and blue bars (bottom two) show the effectiveness of the laundry list at removing these accesses.

For LU, most of the write requests are to data that already exists in the DRAM cache and the laundry list effectively elides the extra write requests. This is why the access ampli-

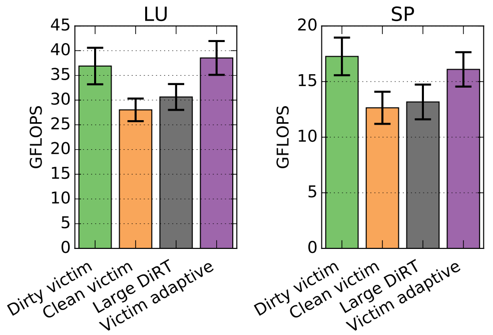

Figure 12: Performance for two workloads with a very aggressive DiRT-like design [33].

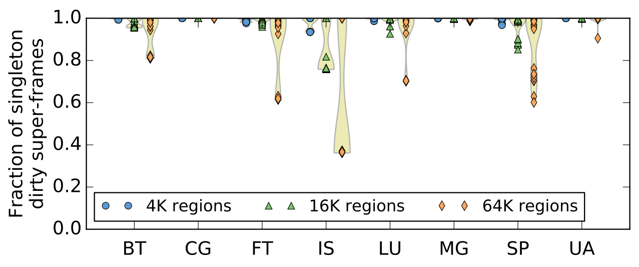

Figure 13: Percent of super-frames that have a single super-block cached. Each point represents one random observation. Also shown is a Gaussian approximation of the distribution across the whole sample.

fication for LU is low for our adaptive victim cache design (Figure 8). For LU, a larger laundry list allows more of the cache to contain dirty data (lower red bar). This reduces the main memory traffic but does not significantly affect performance since most of the accesses are on the "fast dirty" or "clean" paths.

For SP, increasing the size of the laundry list increases its effectiveness. With a larger laundry list we take the "fast dirty" path more often. However, as Figure 10 shows, this does not have a significant effect on performance for this workload. SP and FT (not shown) have the largest effect of increasing the laundry list size. If we instead only used the laundry counts or a bitvector tracking the dirty frames, all of the blue parts of the bars in Figure 11 would generate extra DRAM cache accesses increasing access amplification.

As shown by Figure 11 we find that tag mismatches in the laundry list are rare. This implies that tracking super-block tags instead of 64-byte block tags does not cause significant false sharing. It is rare that there is more than one dirty super-block within each super-frame in the DRAM cache even for large super-blocks (i.e., 16 KB).

5.6 Tracking dirty data

Figure 7 showed that some applications have mostly modified write requests to the DRAM cache. Thus, for these applications, a large fraction of the cache is dirty. Therefore, we should not use solutions like DiRT [33] that constrain the amount of dirty data in the cache as this solution reverts to the clean victim cache performance. Our adaptive victim cache is robust under different write traffic conditions.

Figure 12 shows the performance of victim designs and a DiRT-like design. To model DiRT, we use the laundry list but do not use the laundry counts. Instead of modeling exactly 1024 pages in the tag array (DiRT design), we allow up to $^1/_4$ of the cache (256 MB) to be dirty, compared to only 4 MB in Sim et al. [33]. This is a very aggressive model of the DiRT.

Figure 12 shows that for the workloads where the clean victim design performs poorly DiRT also performs poorly. In this case, DiRT performs about the same as a clean victim cache because more than $^1/_4$ of the active blocks in the cache are dirty. Therefore, the constraining bandwidth is the main memory bandwidth, not the DRAM cache bandwidth. In both of these cases, the adaptive victim performs as well as the dirty victim design because it changes to a writeback mode instead of writethrough.

Figure 13 shows the spatial locality of the NPB. This figure shows the number of super-frames for which only a single super-block is cached for three different super-frame sizes. A super-frame size of 1 KB is the same as directly tracking single cache frames since we have a 16-channel HBM.

Figure 13 shows that for most observations for most workloads almost all super-frames have a single super-block. However, if we increase the super-frame size to 64 KB, the spatial locality breaks down for some workloads. Thus, we choose 16 KB super-frames. This size provides a good tradeoff between area overhead and spatial locality.

5.7 Results summary

The adaptive victim DRAM cache design gets the best of the clean victim and dirty victim designs. The adaptive victim design is robust, and performs well under both high and low miss rate and high and low fractions of write traffic, unlike the other victim designs. The adaptive victim DRAM cache sees these benefits because it does not squander its bandwidth or the bandwidth to main memory. By tracking a small amount of information in SRAM just for dirty frames, our design limits the access amplification and outperforms the alternatives.

6. RELATED WORK

There is significant recent work treating HBM as a DRAM cache. We build on previous research that reduces the latency when accessing DRAM caches, and while it is not the focus of this work, our ideas can be used in conjunction with techniques to increase the hit rates of DRAM caches.

Much of the initial work on DRAM caches focused on reducing the latency when storing the cache tags in DRAM [13, 23, 29]. Loh and Hill proposed storing the cache tags in the same DRAM row as the data to reduce the hit latency [23]. Qureshi and Loh developed a latency-optimized DRAM cache design (Alloy cache) that stores the tag-and-data together in a single entity in the DRAM cache [29]. Additionally, Qureshi and Loh advocate for a direct-mapped cache instead of an associative cache. A direct-mapped design reduces the latency, and the increased hit rate of set-associativity is limited due to the large capacity.

BEAR's (Bandwidth efficient architecture) approach to reducing the bandwidth overheads of DRAM caches is to filter most misses and fills to the DRAM cache [7]. In contrast, our adaptive victim design does not use any filters and works synergistically in a system that implements filters like the BAB and NTC from BEAR to further reduce the access amplification.

Sim et al. [33] propose a DRAM cache design which limits the dirty data in the DRAM cache to allow requests to be satisfied by main memory instead of the DRAM cache. However, their cache *constrains* the dirty data. In a high-miss rate cases, the cache behaves like a writethrough cache (clean victim), hurting performance for some workloads. Stuecheli et al. use the SRAM LLC as a write queue for DRAM and proposes some similar hardware. [36]. The "cache cleaner" can be used in conjunction with our ideas to help keep the DRAM cache clean, and the set state vector tracks the oldest dirty lines in the cache. Our laundry list and counts track *all* of the dirty lines in the DRAM cache for correctness.

Many proposals track the DRAM cache tags at a coarse granularity, like our laundry list and laundry counts [6, 16, 17, 18, 22]. The footprint cache tracks the data in the cache at a page granularity, but manages the cache on a block granularity [17]. The Unison cache combines the ideas of the footprint cache with Alloy cache by storing the metadata in the DRAM cache providing the benefits of the footprint cache without the SRAM area overheads [16]. However, Unison cache has access amplification for managing the metadata that is now stored in the DRAM cache.

The Bi-Modal cache also uses a dual-granularity design that leverages page-based tracking to reduce the overheads of block-based cache [10]. In this design, instead of storing the tags with the data, the Bi-Modal cache reserves a DRAM cache bank/channel for the tags. This may reduce the interference between unnecessary accesses, but not the access amplification.

Tag tables stores the cache tags in main memory in a page-table-like structure and caches frequently accessed tags in the on-chip SRAM LLC [8]. Tag tables does not have any unnecessary accesses to the DRAM cache, but does increase the traffic to main memory, if there is low spatial locality.

Dirty-block index is similar to the laundry list and laundry counts in that it tracks just the dirty blocks in the cache [32]. However, as described in Section 4.3, the adaptive victim cache tracks which *cache frames*—the specific part of the cache—that is dirty, not the physical addresses that are dirty.

7. CONCLUSIONS

We present an adaptive victim DRAM cache design that robustly shifts from behaving as a fully-clean cache to behaving as a dirty cache depending on the program behavior. This design limits the access amplification (total DRAM cache accesses per LLC-miss demand request) to the DRAM cache, improving performance over other designs. Our adaptive victim design is orthogonal to other DRAM cache access filtering techniques opening the door to future synergies.

8. REFERENCES

[1] "NAS Parallel Benchmarks," https://www.nas.nasa.gov/publications/npb.html. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html

[2] AMD, "High Bandwidth Memory," http://www.amd.com/en-us/innovations/software-technologies/hbm. [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/hbm

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks-Summary and Preliminary Results," *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp. 158–165, 1991. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[4] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. M. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, p. 1, aug 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2024716.2024718

[5] B. Black, "MICRO 46 Keynote: Die Stacking is Happening," 2013. [Online]. Available: http://www.cs.wisc.edu/{~}markhill/Tmp/micro2013keynote{_}byran{_}black{_}on{_}die{_}stacking.pdf

[6] C. C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–12, 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7011373

[7] C. Chou, A. Jaleel, and M. K. Qureshi, "BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, vol. 1. New York, New York, USA: ACM Press, 2015, pp. 198–210. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2749469.2750387

[8] S. Franey and M. Lipasti, "Tag tables," *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 514–525, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7056059

[9] K. Gharachorloo, L. A. Barroso, and A. Nowatzyk, "Efficient ECC-Based Directory Implementations for Scalable Multiprocessors," in *12th Symposium on Computer Architecture and High-Performance Computing (HPCA 12)*, 2000.

[10] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, no. i. IEEE, dec 2014, pp. 38–50. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7011376http://ieeexplore.ieee.org/document/7011376/

[11] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi, "Simulating DRAM controllers for future system architecture exploration," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, mar 2014, pp. 201–210. [Online]. Available: http://ieeexplore.ieee.org/document/6844484/

[12] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference on - SYSTOR '09*, p. 1, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1534530.1534544

[13] C.-C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache," *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 51–60, 2014. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628089

[14] JEDEC, "DDR3 SDRAM Standard," http://www.jedec.org/standards-documents/docs/jesd-79-3d, JEDEC, Tech. Rep., 2012. [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd-79-3d

[15] JEDEC, "High Bandwidth Memory (HBM) DRAM," JESD235A, Tech. Rep. November, 2015. [Online]. Available: http://www.jedec.org/standards-documents/results/jesd235a

[16] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," *47th Annual IEEE/ACM International Symposium on Microarchitecture*, no. Micro, pp. 25–37, 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7011375

[17] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache," *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 404–415, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2508148.2485957

[18] X. Jiang, N. Madan, and L. Zhao, "CHOP: Adaptive filter-based dram caching for CMP server platforms," *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA 16)*, pp. 1–12, 2010. [Online]. Available: http://ieeexplore.ieee.org/ielx5/5410726/5416625/05416642.pdf?tp={&}arnumber=5416642{&}isnumber=5416625{%}5Cnhttp://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp={&}arnumber=5416642{&}queryText=chop{%}5Cnhttp://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=5416642

[19] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th annual international symposium on Computer Architecture - ISCA '90*. New York, New York, USA: ACM Press, 1990, pp. 364–373. [Online]. Available: http://portal.acm.org/citation.cfm?doid=325164.325162

[20] H. S. Kannan, B. P. Lilly, P. R. Subramoniam, and P. Kanapathipillai, "Delaying cache data array updates," 2016. [Online]. Available: https://www.google.com/patents/US9229866

[21] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A DRAM Page-mode Scheduling Policy for the Many-core Era," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11*. New York, New York, USA: ACM Press, 2011, p. 24. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2155620.2155624

[22] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless DRAM cache," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*. New York, New York, USA: ACM Press, 2015, pp. 211–222. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2749469.2750383

[23] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," *44th IEEE/ACM International Symposium on Microarchitecture*, p. 454, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2155620.2155673

[24] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, 2012.

[25] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 208–219, 2006.

[26] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, D. Lee, and M. Parkin, "The S3.mp scalable shared memory multiprocessor," *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences HICSS-94*, vol. 1, pp. 144–153, 1994. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=323149

[27] NVIDIA, "Tesla P100 Data Center Accelerator," www.nvidia.com/object/tesla-p100.html, 2017. [Online]. Available: www.nvidia.com/object/tesla-p100.html

[28] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips 23*, 2011.

[29] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, dec 2012, pp. 235–246. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6493623

[30] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory access scheduling," *Proceedings of 27th International Symposium on Computer Architecture*, pp. 1–11, 2000.

[31] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full Speed Ahead: Detailed Architectural

Simulation at Near-Native Speed," *2015 IEEE International Symposium on Workload Characterization*, pp. 183–192, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7314164

[32] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "The Dirty-Block Index," in *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, jun 2014, pp. 157–168. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6853204

[33] J. Sim, G. H. Loh, H. Kim, M. OConnor, and M. Thottethodi, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, dec 2012, pp. 247–257. [Online]. Available: http://ieeexplore.ieee.org/document/6493624/

[34] A. Sodani, "MEGI cache states," Private communication, 2016.

[35] A. Sodani, R. Gramunt, J. Corbal, H.-s. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, mar 2016. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7453080

[36] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, "The virtual write queue: coordinating DRAM and last-level cache policies," in *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*. New York, New York, USA: ACM Press, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1815961.1815972

[37] D. Weiss, M. Dreesen, M. Ciraula, C. Henrion, C. Helt, R. Freese, T. Miles, A. Karegar, R. Schreiber, B. Schneller, and J. Wuu, "An 8MB level-3 cache in 32nm SOI with column-select aliasing," in *2011 IEEE International Solid-State Circuits Conference*. IEEE, feb 2011, pp. 258–260. [Online]. Available: http://ieeexplore.ieee.org/document/5746309/

APPENDIX

A. ADAPTIVE VICTIM CACHE DETAILS

Table 5 shows the detailed implementation of the adaptive victim DRAM cache for LLC-writeback requests. On the left are the states of the LLC blocks when sending an LLC-writeback. On the right are the actions taken by the laundry hardware.

There are five invariants for the laundry hardware.

1. The laundry count for a super-frame is equal to the number of dirty blocks in the super-frame (insight 1).
2. If there is an entry in the laundry list for a super-frame, all dirty blocks in that super-frame have the same tag (insight 2).
3. The laundry count is always incremented on a modified LLC-writeback, unless the data is already dirty in the DRAM cache. There are two ways to detect when the data is dirty: reading the cache block tag and state (row 13) or inferring the frame is dirty when the laundry list entry is valid and the cache-dirty bit is set (row 3).
4. When cleaning a cache frame, the laundry list entry must be invalidated, if it exists. Without removing the laundry list entry in this case, the laundry count could become inconsistent if there is a block cached in the LLC with the cache-dirty bit set.
5. If the tag of the LLC-writeback request matches the tag stored in the cache frame, the data was modified in the LLC (rows 7, 13, 15). Otherwise, there is an error (rows 6, 8, 14, 16). If the block tag matches, the block was inserted into the LLC with the clean-evict bit set, and will only be written back to the DRAM cache if it was modified.

Proactive writebacks are not shown in Table 5. We try to send a proactive writeback on every "Read tag & data", including when the writeback of the victim block is not required. We only send a proactive writeback under the conditions detailed in Section 4.4. On a proactive writeback, the laundry count entry is decremented, and if there is a valid laundry list entry for the super-frame, it must be invalidated (invariant 4). Additionally, if we take a proactive writeback action on an LLC-read, we send a write to the DRAM cache to update the dirty bit in the cache to be consistent with the laundry counts. This write is off the critical path.

	Laundry count == 0?	Laundry list super-frame tag match?	Laundry list physical tag match?	Cache block tag match?	Cache block dirty?	Data modified in LLC?	Data dirty when inserted into LLC?	Read tag & data from DRAM cache	Writeback victim	Create new laundry list entry	Invalidate laundry list super-frame entry	Decrement laundry count	Increment laundry count	Write tag & data into DRAM cache
1	Yes	-	-	-	-	Yes	-			✓			✓	✓
2	Yes	-	-	-	-	No	-							✓
3	No	Yes	Yes	-	-	Yes	Yes							✓
4	No	Yes	Yes	-	-	Yes	No						✓	✓
5	No	Yes	Yes	-	-	No	-							✓
6	No	Yes	No	Yes	Yes	-	-	✓	Error/Invalid					
7	No	Yes	No	Yes	No	Yes	-	✓			✓		✓	✓
8	No	Yes	No	Yes	No	No	-	✓	Error/Invalid					
9	No	Yes	No	No	Yes	Yes	-	✓	✓		✓	✓	✓	✓
10	No	Yes	No	No	Yes	No	-	✓	✓		✓	✓		✓
11	No	Yes	No	No	No	Yes	-	✓			✓		✓	✓
12	No	Yes	No	No	No	No	-	✓						✓
13	No	No	-	Yes	Yes	Yes	-	✓						✓
14	No	No	-	Yes	Yes	No	-	✓	Error/Invalid					
15	No	No	-	Yes	No	Yes	-	✓					✓	✓
16	No	No	-	Yes	No	No	-	✓	Error/Invalid					
17	No	No	-	No	Yes	Yes	-	✓	✓			✓	✓	✓
18	No	No	-	No	Yes	No	-	✓	✓			✓		✓
19	No	No	-	No	No	Yes	-	✓					✓	✓
20	No	No	-	No	No	No	-	✓						✓

Table 5: Details of bandwidth optimized DRAM cache functions. Table 6 contains an explanation for each row.

1	First dirty data inserted into a clean super-frame
2	Clean data inserted into a clean super-frame
3	Dirty data inserted into a super-frame that contains dirty data from the same super-block. Can infer the cache frame contains dirty data (invariant 3); do not increment laundry count.
4	Dirty data inserted into a super-frame that contains dirty data from the same super-block.
5	Clean data inserted into a super-frame that contains dirty data from the same super-block. Must invalidate laundry list entry (invariant 4).
6	Error. Violates invariant 2.
7	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block.
8	Error. Clean-evict filter will elide this writeback (invariant 5).
9	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block.
10	Clean data inserted into a super-frame that contains dirty data from the same super-block. Must invalidate laundry list entry (invariant 4).
11	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block.
12	Clean data inserted into a super-frame that contains dirty data from a different super-block.
13	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block. The cache frame was previously dirty; no need to increment the laundry count (invariant 3)
14	Error. Clean-evict filter will elide this writeback (invariant 5).
15	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block. The cache frame was previously clean; increment the laundry count (invariant 3)
16	Error. Clean-evict filter will elide this writeback (invariant 5).
17	Dirty data inserted into a dirty super-frame that holds dirty data from a different super-block.
18	Clean data overwriting a dirty cache frame.
19	Dirty data inserted into a dirty super-frame overwriting clean data.
20	Clean data inserted into a dirty super-frame overwriting clean data.

Table 6: Explanation of rows in Table 5.