2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Fundamental Latency Trade-offs in Architecting DRAM Caches*

Outperforming Impractical SRAM-Tags with a Simple and Practical Design

Moinuddin K. Qureshi
Dept. of Electrical and Computer Engineering
Georgia Institute of Technology
moin@gatech.edu

Gabriel H. Loh
AMD Research
Advanced Micro Devices, Inc.
gabe.loh@amd.com

Abstract

This paper analyzes the design trade-offs in architecting large-scale DRAM caches. Prior research, including the recent work from Loh and Hill, have organized DRAM caches similar to conventional caches. In this paper, we contend that some of the basic design decisions typically made for conventional caches (such as serialization of tag and data access, large associativity, and update of replacement state) are detrimental to the performance of DRAM caches, as they exacerbate the already high hit latency. We show that higher performance can be obtained by optimizing the DRAM cache architecture first for latency, and then for hit rate.

We propose a latency-optimized cache architecture, called Alloy Cache*, that eliminates the delay due to tag serialization by streaming tag and data together in a single burst. We also propose a simple and highly effective* Memory Access Predictor *that incurs a storage overhead of 96 bytes per core and a latency of 1 cycle. It helps service cache misses faster without the need to wait for a cache miss detection in the common case. Our evaluations show that our latency-optimized cache design significantly outperforms both the recent proposal from Loh and Hill, as well as an impractical SRAM Tag-Store design that incurs an unacceptable overhead of several tens of megabytes. On average, the proposal from Loh and Hill provides 8.7% performance improvement, the "idealized" SRAM Tag design provides 24%, and our simple latency-optimized design provides 35%.*

1. Introduction

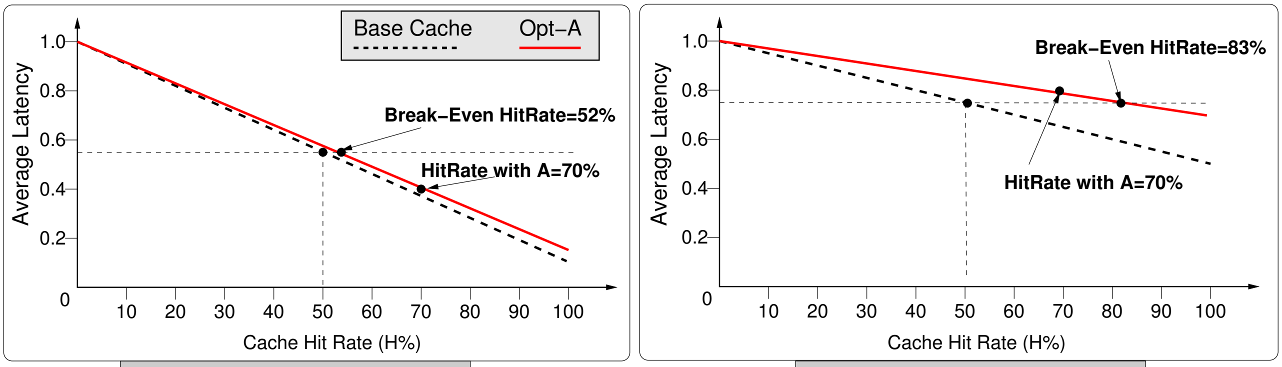
Emerging 3D-stacked memory technology has the potential to provide a step function in memory performance. It can provide caches of hundreds of megabytes (or a few gigabytes) at almost an order of magnitude higher bandwidth compared to traditional DRAM; as such, it has been a very active research area [2, 4, 7, 12, 13, 19]. However, to get performance benefit from such large caches, one must first handle several key challenges, such as architecting the tag store, optimizing hit latency, and handling misses efficiently. The prohibitive overhead of storing tags in SRAM can be avoided by placing the tags in DRAM, but naively doing so doubles the latency of DRAM cache (one access each for tag and data). A recent work from Loh and Hill [10, 11] makes the tags-in-DRAM approach efficient by co-locating the tags and data in the same row. However, similar to prior work on DRAM caches, the recent work also architects DRAM caches in largely the same way as traditional SRAM caches. For example by having a serialized tag-and-data access and employing typical optimizations such as high associativity and intelligent replacement.

We observe that the effectiveness of cache optimizations depends on technology constraints and parameters. What may be regarded as indispensable in one set of constraints, may be rendered ineffective when the parameters and constraints change. Given that the latency and size parameters of a DRAM cache are so widely different from traditional caches, and the technology constraints are disparate, we must be careful about the implicit optimizations that get incorporated in the architecture of the DRAM cache. In particular, we point out that DRAM caches are much slower than traditional caches, so optimizations that exacerbate the already high hit latency may degrade overall performance even if they provide a marginal improvement in hit rate. While this may seem to be a fairly simple and straight-forward concept, it has a deep impact (and often counter-intuitive implications) on the design of DRAM cache architectures. We explain the need for reexamining conventional cache optimizations for DRAM caches with a simple example.

Consider a system with a cache and a memory. Memory accesses incur a latency of 1 unit, and cache accesses incur 0.1 unit. Increasing the cache hit rate from 0% to 100% reduces the average latency linearly from 1 to 0.1, shown as "Base Cache" in Figure 1(a). Assuming the base cache has a hit rate of 50%, then the average memory access time for the base cache is 0.55. Now consider an optimization A that eliminates 40% of the misses (hit rate with A: 70%) but increases hit latency to 1.4x (hit latency with A: 0.14 unit). We want to implement A only if it reduces average latency. We may begin by examining the target hit-rate for A given the higher hit-latency, such that the average latency is equal to the base case, which we call the *Break-Even Hit Rate (BEHR)*. If the hit-rate with A is higher than the BEHR, then A will reduce average latency. For our example, the BEHR for A is 52%. So, we deem A to be a highly effective optimization, and indeed it reduces average latency from 0.55 to 0.40.

*The work on Memory Access Prediction (Section 5) was done in 2009 while the first author was a research scientist at IBM Research [15].

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.30



(a) Fast Cache [Hit Latency 0.1]

(b) Slow Cache [Hit Latency 0.5]

Figure 1: Effectiveness of cache optimizations depend on cache hit latency. Option A increases hit latency by 1.4x and hit-rate from 50% to 70%. (a) For a fast cache, A is highly effective at reducing average latency from 0.55 to 0.4 (b) For a slow cache, A increases average latency from 0.75 to 0.79.

Now, consider the same "highly effective" optimization A, but now the cache has a latency of 0.5 units, much like the relative latency of a DRAM cache. The revised hit latency with A will now be 1.4x0.5=0.7 units. Consider again that our base cache has a hit-rate of 50%. Then the average latency for the base cache would be 0.75 units, as shown in Figure 1(b). To achieve this average latency, A must have a hit rate of 83%. Thus optimization A, which was regarded as highly effective in the prior case, ends up increasing average latency (from 0.75 to 0.79). The Break Even Hit Rate depends also on the hit rate of the base cache. If the base cache had a hit rate of 60%, then A would need a 100% hit-rate simply to break even! Thus, seemingly indispensable and traditionally effective cache optimizations may be rendered ineffective if they have a significant impact on cache hit latency for DRAM caches. Note that typical cache optimizations, such as higher associativity and better replacement, do not usually provide a miss reduction as high as 40%, which we have considered for A. However, our detailed analysis (Section 2) shows that to support these optimizations, previously analyzed DRAM cache architectures do incur a hit latency overhead of more than 1.4x as considered for A.

It is our contention that DRAM caches should be designed from the ground-up keeping hit latency as a first priority for optimization. Design choices that increase hit latency by more than a negligible amount must be carefully analyzed to see if it indeed provides an overall improvement. We find that previously proposed designs for DRAM caches that try to maximize hit-rate are not well suited for optimizing overall performance. For example, they continue to serialize the tag and data access (similar to traditional caches), which increases hit latency significantly. They provide high associativity (several tens of ways) at the expense of hit latency. We can significantly improve the performance of DRAM caches by optimizing them for latency first, and then for hit rate. With this insight, this paper makes following contributions:

1. We analyze the latency of three designs: SRAM-Tags, the proposal from Loh and Hill, and an ideal latency-optimized DRAM cache. We find that the Loh-Hill proposal suffers from significant latency overheads due to tag serialization and due to the MissMap predictor. For SRAM-Tags, tag serialization latency limits performance. Both designs leave significant room for performance improvement compared to the latency-optimized design.
2. We show that *de-optimizing* the DRAM cache from a highly-associative structure to direct-mapped improves performance by reducing the hit latency, even if it degrades cache hit rate. For example, simply configuring the design of Loh and Hill from 29-way to direct-mapped enhances performance improvement from 8.7% to 15%. However, this design still suffers from tag serialization due to separate accesses to the "tag-store" and "data-store."
3. We propose the *Alloy Cache*, a highly-effective latency-optimized cache architecture. Rather than splitting cache space into "tag store" and "data store," it tightly integrates or *alloys* the tag and data into one unit (Tag and Data, TAD). Alloy Cache streams out a TAD unit on each cache access, thus avoiding the tag serialization penalty.
4. We present a simple and effective *Memory Access Predictor* [15] to avoid the cache access penalty in the path of servicing cache miss. Unlike MissMap, which incurs multi-megabyte storage and L3 access delay, our proposal requires a storage overhead of 96 bytes per core and incurs a latency of 1 cycle. Our predictor provides a performance improvement within 2% of a perfect predictor.

Our evaluations with a 256MB DRAM cache show that, on average, our latency-optimized design (35%) significantly outperforms both the proposal from Loh and Hill (8.7%) as well as the impractical SRAM-Tag design (24%). Thus, our simple design with less than 1KB overhead (due to predictor) provides 1.5x the performance benefits of the SRAM design that requires several tens of megabytes of overhead.

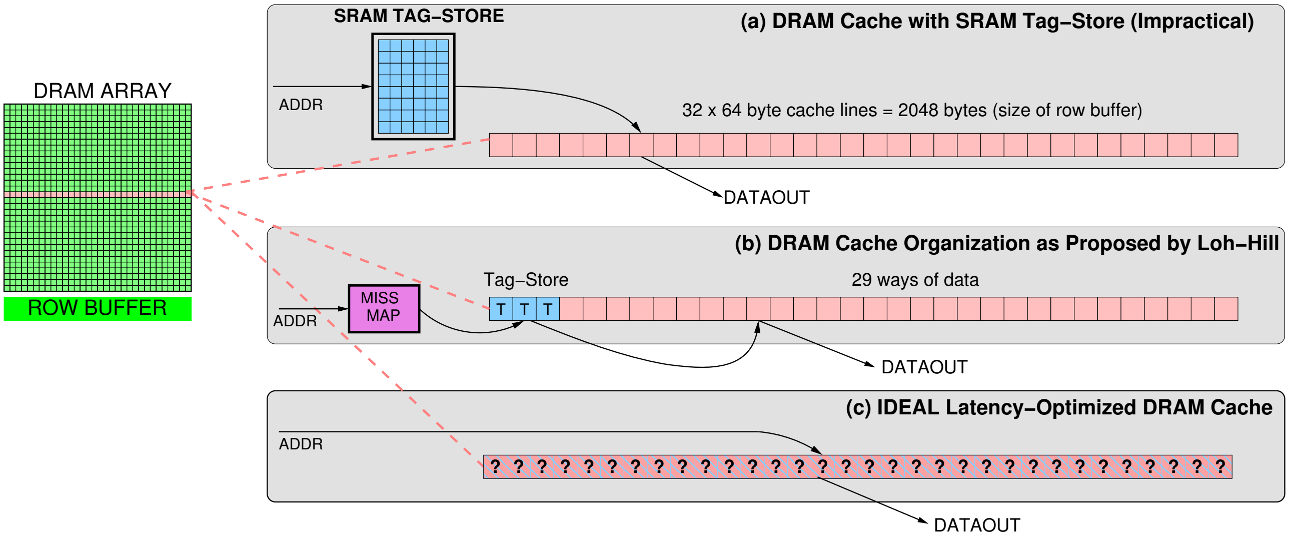

DRAM ARRAY
ROW BUFFER
SRAM TAG–STORE
(a) DRAM Cache with SRAM Tag–Store (Impractical)
ADDR
32 x 64 byte cache lines = 2048 bytes (size of row buffer)
DATAOUT
(b) DRAM Cache Organization as Proposed by Loh–Hill
Tag–Store
29 ways of data
MISS MAP
ADDR
T T T
DATAOUT
(c) IDEAL Latency–Optimized DRAM Cache
ADDR
? ?
DATAOUT

Figure 2: DRAM Cache organization and flow for a typical access for (a) SRAM Tag-store, (b) the organization proposed by Loh and Hill, and (c) an IDEAL latency-optimized cache.

2. Background and Motivation

While stacked memory can enable giga-scale DRAM caches, several challenges must be overcome before such caches can be deployed. An effective design of DRAM cache must balance (at-least) four goals. First, it should minimize the non-DRAM storage required for cache management (using a small fraction of DRAM space is acceptable). Second, it should minimize hit latency. Third, it should minimize miss latency, so that misses can be sent to memory quickly. Fourth, it should provide a good hit-rate. These requirements are often conflicting with each other, and a good design must balance these appropriately to maximize performance.

It is desirable to organize DRAM caches at the granularity of a cache line in order to efficiently use cache capacity, and to minimize the consumption of main memory bandwidth [10]. One of the main challenges in architecting a DRAM cache at a line granularity is the design of the tag store. A per-line tag overhead of 5-6 bytes quickly translates into a total tag-store overhead of a few tens of megabytes for a cache size in the regime of a few hundred megabytes. We discuss the options to architect the tag store, and how it impacts cache latency.

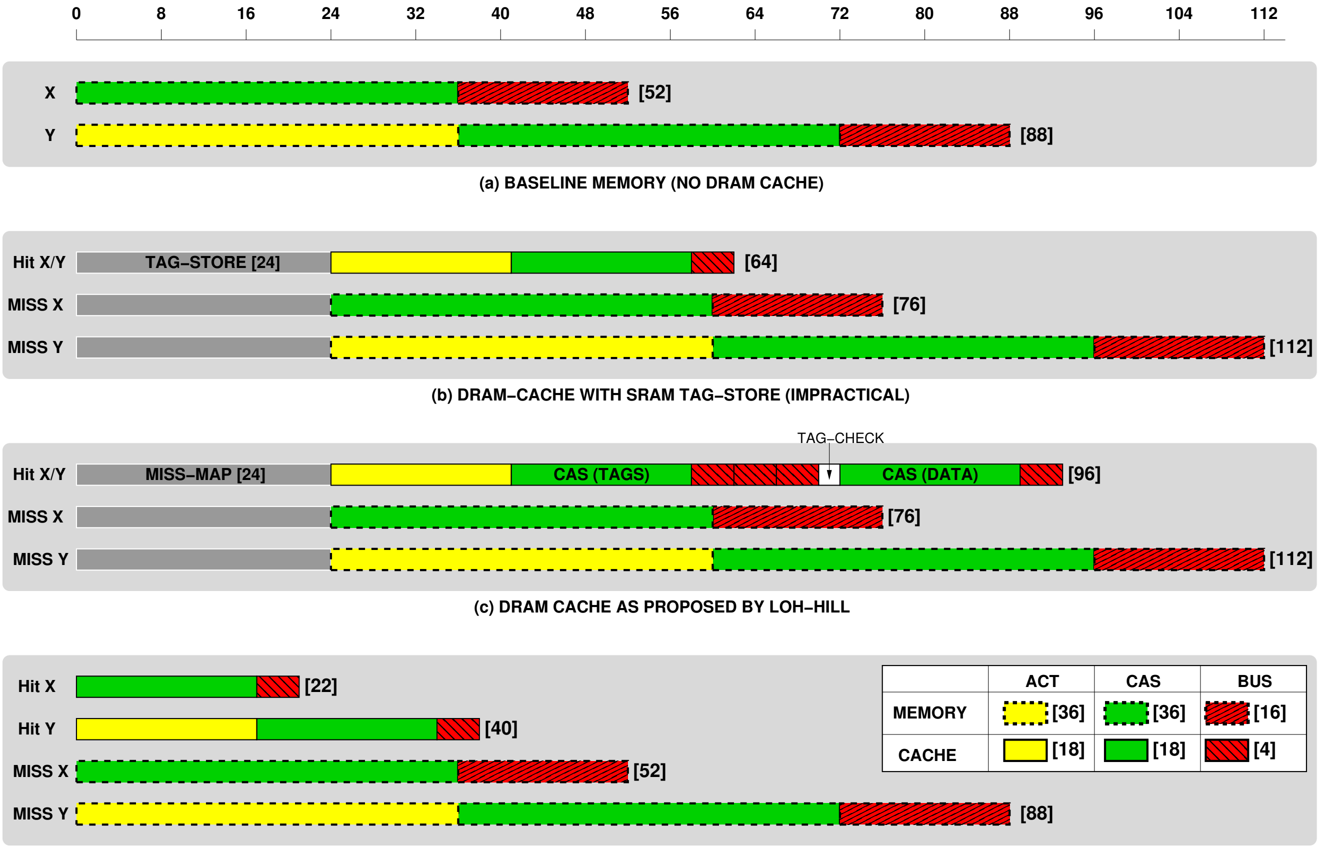
2.1. SRAM-Tag Design

This approach stores tags in a separate SRAM structure, as shown in Figure 2(a). For the cache sizes we consider, this design incurs an unacceptably high overhead (24MB for 256MB DRAM cache). We can configure the DRAM cache as a 32-way cache and store the entire set in one row of the cache [2, 10]. To obtain data, the access must first go through the tag-store. We call the latency due to serialization of tag access as *"Tag Serialization Latency" (TSL)*. TSL directly impacts the cache hit latency, and hence must be minimized.

2.2. Tags-in-DRAM: The LH-Cache

We can place the tags in DRAM to avoid the SRAM overhead. However, naively doing so would require that each DRAM cache access incurs a latency of two accesses, one for tag and the other for data, further exacerbating the already high hit latency. A recent work from Loh and Hill [10, 11] reduces the access penalty of DRAM tags by co-locating the tags and data for the entire set in the same row, as shown in Figure 2(b). It reserves three lines in a row for tag store, and makes the other 29 lines available as data lines, thus providing a 29-way cache. A cache access must first obtain the tags, and then the data line. The authors propose *Compound Access Scheduling* so that the second access (for data) is guaranteed to get a row buffer hit. However, the second access still incurs approximately half the latency of the first, so this design still incurs significant TSL overhead.

Given that the tag check incurs a full DRAM access, the latency for servicing a cache miss is increased significantly. To service cache misses quickly, the authors propose a *MissMap* structure that keeps track of the lines in the DRAM cache. If a miss is detected in the MissMap, then the access can go directly to memory without the need to wait for a tag check. Unfortunately, the MissMap structure requires multi-megabyte storage overhead. To implement this efficiently, the authors propose to embed the MissMap in the L3 cache. The MissMap is queried on each L3 miss, which means that the extra latency of the MissMap, which we call *Predictor Serialization Latency (PSL)*, is added to the latency of both cache hit and cache miss. Thus, the hit latency suffers from both TSL and PSL. Throughout this paper, we will assume that the design from Loh and Hill [10] is always implemented with the MissMap, and we will refer to it simply as the *LH-Cache*.



(a) BASELINE MEMORY (NO DRAM CACHE)

(b) DRAM–CACHE WITH SRAM TAG–STORE (IMPRACTICAL)

(c) DRAM CACHE AS PROPOSED BY LOH–HILL

(d) IDEAL LATENCY–OPTIMIZED DRAM CACHE

Figure 3: Latency breakdown for two classes of isolated accesses X and Y. X has good row buffer locality and Y needs to activate the memory row to get serviced. The latency incurred in an activity is marked as [N] processor cycles.

2.3. IDEAL Latency-Optimized Design

Both SRAM-Tags and LH-Cache have hit latency due to TSL. To reduce conflict misses, both designs are configured similar to conventional set-associative caches. They place the entire set in a row for conflict miss reduction, sacrificing the row-buffer hits for cache accesses (sequentially-addressed lines map to different sets, and the probability of temporally-close accesses going to same set is $\ll 1\%$). Furthermore, for LH-Cache, supporting high associativity incurs higher latency due to streaming a large number of tag lines, and the bandwidth consumed due to replacement update and victim selection further worsens the already high hit latency.

We argue that DRAM caches must be architected to minimize hit latency. This can be done by a suitable cache structure that avoids extraneous latency overheads and supports row buffer locality. Ideally, such a structure would have zero TSL and PSL, and would stream out exactly one cache line after a latency equal to the raw latency of the DRAM structure (ACT+CAS for accesses that open the row, and only CAS for row-buffer hits). Also, it would know a priori if the access would hit in cache or go to memory. We call such a design as *IDEAL-LO (Latency Optimized)*. As shown in Figure 2(c), it does not incur any latency overheads.

2.4. Raw Latency Breakdown

In this section, we quantitatively analyze the latency effectiveness of different designs. While there are several alternative implementations of both SRAM-Tags and LH-Cache, we will restrict the analysis in this section to the exact implementation of SRAM-Tags and LH-Cache as previously described, including identical latency numbers for all parameters [10], which are summarized in Table 2. We report latency in terms of processor cycles. Off-chip memory memory has tACT and tCAS of 36 cycles each, and needs 16 cycles to transfer one line on the bus. Stacked DRAM has tACT and tCAS of 18 cycles each, and needs 4 cycles to transfer one line on the bus. The latency for accessing the L3 cache as well as the SRAM-Tag store is assumed to be 24 cycles.

To keep the analysis tractable, we will initially consider only isolated accesses of two types, X and Y. Type X has a high row buffer hit-rate for off-chip memory and is serviced by memory with a latency equal to a row buffer hit. Type Y needs to open the row in order to get serviced. The baseline memory system would service X in 52 cycles (36 for CAS, and 16 for Bus), and Y in 88 cycles (36 for ACT, 36 for CAS, and 16 for Bus). Figure 3 shows the latency incurred by different designs to service X and Y.

As both SRAM-Tags and LH-Cache map the entire set to a single DRAM row, they get poor row buffer hit-rates in the DRAM cache. Therefore for both X and Y, neither cache design will give a row buffer hit. Therefore, a hit for both X and Y will incur a latency of ACT. However, with IDEAL-LO, X gets a row buffer hit and Y will need a latency of ACT.

The SRAM-Tag suffers a Tag Serialization Latency of 24 cycles for both cache hits and misses. A cache hit needs another 40 cycles (18 ACT + 18 CAS + 4 burst), for a total of 64 cycles. Thus SRAM-Tag increases latency for hits on X, decreases latency for hits on Y, and increases latency for misses on both X and Y due to the inherent latency of tag-lookup.

LH-Cache first probes the MissMap, which incurs a latency of 24 cycles.[1] For a hit, LH-Cache then issues a read for tag information (ACT+CAS, 36 cycles), then it streams out the three tag lines (12 cycles), followed by one DRAM cycle for tag check. This is followed by access to the data line (CAS+burst). Thus a hit in LH-Cache incurs a latency of 96 cycles, almost doubling the latency for X on hit, degrading the latency for Y on hit, and adding MissMap latency to miss.

An IDEAL-LO organization would service X with a row buffer hit, reducing the latency to 22 cycles. A hit for Y would incur 40 cycles. IDEAL-LO does not increase miss latency.

To summarize, we assumed that the raw latency of the stacked DRAM cache is half that of the off-chip memory. However, due to the inherent serialization latencies, LH-Cache (and in most cases SRAM-Tag) has a higher raw latency than off-chip memory. Whereas, IDEAL-LO continues to provide a reduction in hit latency on cache hits.

2.5. Bandwidth Benefits of DRAM Cache

Even with a higher raw hit latency than main memory, both LH-Cache and SRAM-Tag can still improve performance by providing two indirect benefits. First, stacked DRAM has $\sim$8x more bandwidth than off-chip DRAM, which means cache requests wait less. Second, contention for off-chip memory is reduced as DRAM cache hits are filtered. The performance benefit of LH-Cache and SRAM-Tags comes largely from these two indirect benefits and not due to raw latency. The first benefit relies on having a cache that has high-bandwidth. Although stacked DRAM has 8x raw bandwidth compared to off-chip, LH-Cache uses more than 4x line transfers on each cache access (3 for tag, 1 for data, and some for update), so the effective bandwidth becomes < 2x. Both SRAM-Tag and IDEAL-LO maintains 8x bandwidth by efficiently using the bandwidth. Therefore, they are more effective than LH-Cache at reducing waiting time for cache requests. We found that the latency for servicing requests from off-chip memory is similar for all three designs.

2.6. Performance Potential

Figure 4 compares the performance of three designs: SRAM-Tag, LH-Cache, and IDEAL-LO. The numbers are speedups with respect to a baseline that does not have a DRAM cache, and are reported for a DRAM cache of size 256MB (methodology in Section 3).

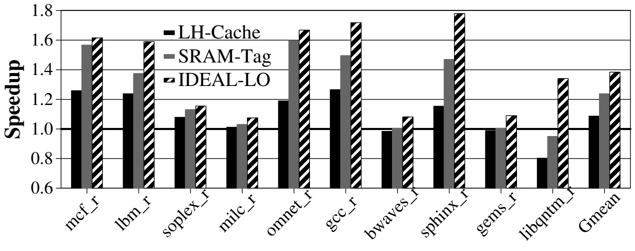


Figure 4: Performance potential of IDEAL-LO design.

Validation with Prior Work: On average, SRAM-Tags provide a performance improvement of 24% and LH-Cache 8.7%. Thus, LH-Cache obtains only one-third of the performance benefit of SRAM-Tags which is inconsistent with the original LH-Cache study [10], which reported that the LH-Cache obtains performance very close to SRAM-Tag. Given the difference in raw hit latencies between the two designs (see Figure 3) and 4x bandwidth consumption of LH-Cache compared to SRAM-Tags, it is highly unlikely that LH-Cache would perform close to SRAM-Tags. A significant part of this research study was to resolve this inconsistency with previously reported results. The authors of the LH-Cache study [10] have subsequently published an errata [9] that shows revised evaluations after correcting deficiencies in their evaluation infrastructure. The revised evaluations for 256MB show on average $\approx$10% improvement for LH-Cache and $\approx$ 25% for SRAM-Tag, consistent with our evaluations.

Note that IDEAL-LO outperforms both SRAM-Tags and LH-Cache, and provides an average of 38%. For libquantum, the memory access patterns has very high row-buffer hit rates in the off-chip DRAM resulting in mostly type X requests. Therefore, both SRAM-Tag and LH-Cache show performance degradations due to their inabilities to exploit the spatial locality of sequential access streams.

2.7. De-Optimizing for Performance

We now present simple *de-optimizations* that improve the overall performance of LH-Cache, at the expense of hit-rate. The first is using a replacement scheme that does not require update (random replacement, instead of LRU-based DIP): This avoids LRU-update and victim selection overheads, which improves hit latency due to the reduced bank contention. The second converts LH-Cache from 29-way to direct mapped. This has two advantages: direct, in that we do not need to stream out three tag lines on each access, and indirect, employing open page mode for lower latency. For

[1] The MissMap serialization latency can be avoided by probing the MissMap in parallel with L3 access. However, this would double the L3 accesses, as MissMap would be probed on L3 hits as well, causing bank/port contention and increasing L3 latency and power consumption. Hence, prior work [10] used serial access for MissMap, and so did we.

SRAM-Tag and LH-cache, sequentially addressed cachelines are mapped to different sets, and because each set is mapped to a unique row, the probability of a row-buffer hit is very low. With a direct-mapped organization, several consecutive sets map to the same physical DRAM row, and so accesses with spatial locality result in row buffer hits. The row-buffer hit rate for the direct-mapped configuration was measured to be 56% on average, compared to less than 0.1% when the entire set (29-way or 32-way) is mapped to the same row.

Table 1: Impact of De-Optimizing LH-Cache.

Configuration	Speedup	Hit-Rate	Hit Latency (cycles)
LH-Cache	8.7%	55.2%	107
LH-Cache + Rand Repl	10.2%	51.5%	98
LH-Cache (1-way)	**15.2%**	**49.0%**	**82**
SRAM-Tag (32-way)	23.8%	56.8%	67
SRAM-Tag (1-way)	24.3%	51.5%	59
IDEAL-LO (1-way)	38.4%	48.2%	35

Table 1 shows the speedup, hit-rate, and average hit latency for various flavors of LH-Cache. We also compare them with SRAM-Tag and IDEAL-LO. LH-Cache has hit latency of 107 cycles, almost 3x compared to IDEAL-LO. De-optimizing LH-Cache reduces the latency to 98 cycles (random replacement) and 82 cycles (direct mapped). These optimizations reduce hit-rate and increase misses significantly (a reduction in hit-rate from 55% to 49% represents almost 15% more misses). However, this still improves performance significantly. For SRAM-Tag, converting from 32-way to 1-way had little benefit ($\approx 0.5\%$), as the reduction in hit latency is offset by reduction in hit-rate.

While a direct-mapped implementation of LH-cache is more effective than the set-associative implementation, it still suffers from Tag Serialization Latency, as well as the Predictor Serialization Latency, resulting in a significant performance gap between LH-Cache and IDEAL-LO (15% vs. 38%). Our proposal removes these serialization latencies and obtains performance close to IDEAL-LO. We describe our experimental methodology before describing our solution.

3. Experimental Methodology

3.1. Configuration

We use a Pin-based x86 simulator with a detailed memory model. Table 2 shows the configuration used in our study. The parameters for the L3 cache and DRAM (off-chip and stacked) are identical to the original LH-Cache study [10], including a 24-cycle latency for the SRAM-Tag. For LH-Cache, we model an idealized unlimited-size Miss Map that resides in the L3 cache but does not consume any L3 cache capacity. For both LH-Cache and SRAM-Tag we use LRU-based DIP [16] replacement. We will perform detailed studies for a 256MB DRAM cache. In Section 6.1, we will analyze cache sizes ranging from 64MB to 1GB.

Table 2: Baseline Configuration

Processors	
Number of cores	8
Frequency	3.2GHz
Width	1 IPC
Last Level Cache	
L3 (shared)	8MB, 16-way 24 cycles
Off-Chip DRAM	
Bus frequency	800 MHz (DDR 1.6 GHz)
Channels	2
Ranks	1 Rank per channel
Banks	8 Banks per rank
Row buffer size	2048 bytes
Bus width	64 bits per channel
tCAS-tRCD-tRP-tRAS	9-9-9-36
Stacked DRAM	
Bus frequency	1.6GHz (DDR 3.2GHz)
Channels	4
Banks	16 Banks per rank
Bus width	128 bits per channel

3.2. Workloads

We use a single SimPoint [14] slice of 1 billion instructions for each benchmark from the SPEC2006 suite. We perform evaluations by executing 8 copies of each benchmark in rate mode. Given that our study is about large caches, we perform detailed studies only for the 10 workloads that have a speedup of more than 2 with a perfect L3 cache (100% hit-rate). Other workloads are analyzed in Section 6.4.

Table 3 shows the workloads sorted based on perfect L3 speedup, the Misses Per 1000 Instructions (MPKI), and footprint (the number of unique lines multiplied by linesize). We model a virtual-to-physical mapping to ensure two benchmarks do not map to the same physical address. We use a suffix *_r* with the name of the benchmark to indicate rate mode.

We perform timing simulation until all benchmarks in the workload finish execution and measure the execution time of the workload as the average execution time across all 8 cores.

Table 3: Benchmark Characteristics.

Workload Name	Perfect-L3 Speedup	MPKI	Footprint
mcf_r	4.9x	74.0	10.4 GB
lbm_r	3.8x	31.8	3.3 GB
soplex_r	3.5x	27.0	1.9 GB
milc_r	3.5x	25.7	4.1 GB
omnetpp_r	3.1x	20.9	259 MB
gcc_r	2.8x	16.5	458 MB
bwaves_r	2.8x	18.7	1.5 GB
sphinx_r	2.4x	12.3	80 MB
gems_r	2.2x	9.7	3.6 GB
libquantum_r	2.1x	25.4	262 MB

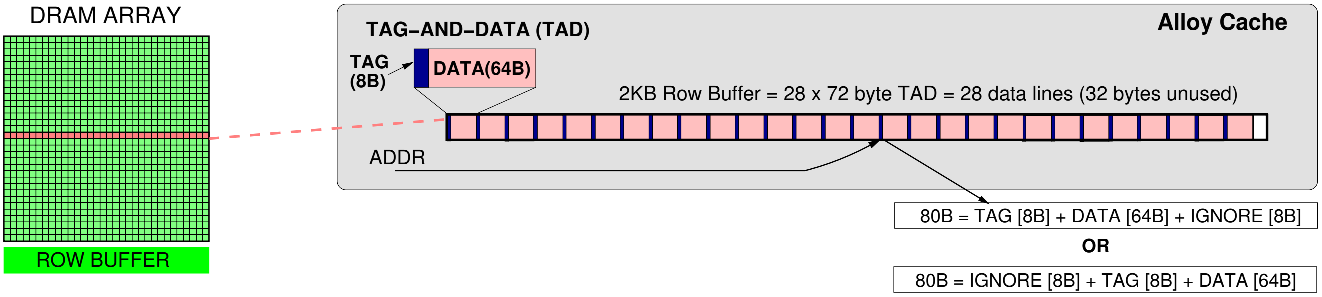

DRAM ARRAY
ROW BUFFER
TAG–AND–DATA (TAD)
Alloy Cache
TAG (8B)
DATA(64B)
2KB Row Buffer = 28 x 72 byte TAD = 28 data lines (32 bytes unused)
ADDR
80B = TAG [8B] + DATA [64B] + IGNORE [8B]
OR
80B = IGNORE [8B] + TAG [8B] + DATA [64B]

Figure 5: Architecture and Operation of Alloy Cache that integrates Tag and Data (TAD) into a single entity called TAD. The size of data transfers is determined by a 16-byte wide data-bus, hence minimum transfer of 80 bytes for obtaining one TAD.

4. Latency-Optimized Cache Architecture

While configuring the LH-Cache from a 29-way structure to a direct-mapped structure improved performance (from 8% to 15%), it still left significant room for improvement compared to a latency-optimized solution (38%). One of the main sources of this gap is the serialization latency due to tag lookup. We note that LH-Cache created a separate "tag-store" and "data-store" in the DRAM cache, similar to conventional caches. A separate tag-store and data-store makes sense for a conventional cache, because they are indeed physically separate structures. The tag-store is optimized for latency to support quick-lookups and can have multiple ports, whereas the data-store is optimized for density. We make an important observation that creating a separate contiguous tag-store (similar to conventional caches) is not necessary when tags and data co-exist in the same DRAM array.

4.1. Alloy Cache

Obviating the separation of tag-store and data-store can help us avoid the TSL overhead. This is the key insight in our proposed cache structure, which we call the *Alloy Cache*. The Alloy Cache tightly integrates or *alloys* tag and data into a single entity called *TAD (Tag and Data)*. On an access to the Alloy Cache, it provides one TAD. If the tag obtained from the TAD matches with the given line address, it indicates a cache hit and the data line in the TAD is supplied. A tag mismatch indicates cache miss. Thus, instead of having two separate accesses (one to the "tag-store" and the other to the "data-store"), Alloy Cache tightly integrates those two accesses into a single unified access, as shown in Figure 5. On a cache miss, there is a minor cost in that bandwidth is consumed transferring a data line that is not used. Note that this overhead is still substantially less than the *three* tag lines that must be transferred for both hits and misses in the LH-Cache.

Each TAD represents one set of the direct-mapped Alloy Cache. Given that the Alloy Cache has a non-power-of-two number of sets, we cannot simply use the address bits to identify the set. We assume that a modulo operation on the line address is used to determine the set index of the Alloy Cache.[2] A non-power-of-two number of sets also means that the tag entry needs to store full tags, which increases the size of the tag entry. We estimate that a tag entry of 8 bytes is more than sufficient for the Alloy Cache (for a physical address space of 48-bits, we need 42 tag bits, 1 valid bit, 1 dirty bit, and the remaining 20 bits for coherence support and other optimizations). The minimum size of a TAD is thus 72 bytes (64 bytes for data line and 8 bytes for tag). The Alloy Cache can store 28 lines in a row, reaching close to the 29-lines per row storage efficiency of the LH-Cache.

The size of data transfer from the Alloy Cache is also affected by the physical constraints of the DRAM cache. For example, the size of the databus assumed for our stacked DRAM configuration is 16 bytes, which means transfers to-and-from the cache occur at the granularity of 16 bytes. Thus, it will take a burst of five transfers to obtain one TAD of 72 bytes. To keep our design simple, we restrict the transfers to be aligned at the granularity of the data-bus size. This requirement means that for odd sets of the Alloy Cache, the first 8 bytes are ignored and for even sets the last 8 bytes are ignored. The tag-check logic checks either the first eight bytes or the next eight bytes depending on the low bit of the set index.

4.2. Impact on Effective Bandwidth

Table 4 compares the effective bandwidth of servicing one cache line from various structures. The raw bandwidths and effective bandwidths are normalized to off-chip memory. On a cache hit, LH-Cache transfers (3 lines of tag + 1 data + replacement update) reducing raw bandwidth of 8x into an effective bandwidth of less than 2x. Whereas, Alloy Cache can provide an effective bandwidth of up-to 6.4x.

[2] Designing a general purpose modulo-computing unit incurs high area and latency overheads. However, here we compute modulo with respect to a constant, so it is much simpler and faster compared to a general-purpose solution. In fact, modulo with respect to 28 (number of sets in one row of Alloy Cache) can be computed easily with eight 5-bit adders using residue arithmetic (28=32-4). This value can then be removed from the line address to get row-id of DRAM cache. We estimate the calculation to take two cycles and only a few hundred logic gates. We assume that the index calculation of the Alloy Cache happens in parallel with the L3 cache access (thus, we have up to 24 cycles to calculate the set index of the Alloy Cache).

Table 4: Bandwidth comparison (relative to off-chip memory).

Structure	Raw Bandwidth	Transfer per access (hit)	Effective Bandwidth
Off-chip Memory	1x	64 byte	1x
SRAM-Tag	8x	64 byte	8x
LH-Cache	8x	(256+16) byte	1.8x
IDEAL-LO	8x	64 byte	8x
Alloy Cache	8x	80 byte	6.4x

4.3. Latency and Performance Impact

The Alloy Cache avoids tag serialization. Instead of two serialized accesses, one each for tag and data, it provides tag and data in a single burst of five transfers on the data-bus. Comparatively, a transfer of only the data line would take four transfers, so the latency overhead of transferring TAD instead of only the data line is 1 bus cycle. However, this overhead is negligible compared to the TSL overhead incurred by SRAM-Tag (24 cycles) and LH-Cache (32-50 cycles). Because of the avoidance of TSL, the average hit latency for Alloy Cache is significantly better (42 cycles), compared to both SRAM-Tag (69 cycles) and LH-Cache (107 cycles).

The Alloy Cache reduces the TSL but not the PSL, so the overall performance depends on how misses are handled. We consider three scenarios: First, no prediction (wait for tag access until cache miss is detected). Second, use the MissMap (PSL of 24 cycles). Third, perfect predictor (100% accuracy, 0 latency). Figure 6 compares the speedup of these to the impractical SRAM-Tag design configured as 32-way.

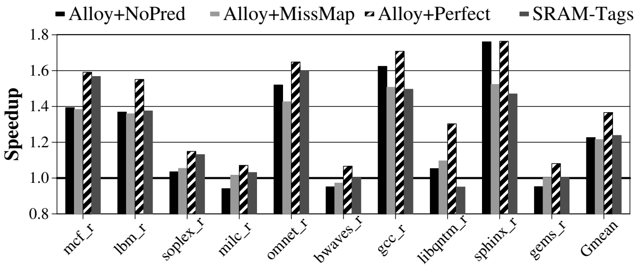


Figure 6: Speedup with Alloy Cache.

Even without any predictor, the Alloy Cache provides a 21% performance improvement, much closer to the impractical SRAM-Tag. This is primarily due to the lower hit latency. A MissMap provides better miss handling, but the 24-cycle PSL is incurred on both hits and misses, so the performance is actually worse than not using a predictor. With a perfect predictor (100% accuracy and zero-cycle latency), the Alloy Cache's performance increases to 37%. The next section describes effective single-cycle predictors that obtain performance close to that with a perfect predictor.

5. Low-Latency Memory Access Prediction

The MissMap approach focuses on getting perfect information about the presence of the line in the DRAM cache. Therefore, it needs to keep track of information on a per-line basis. Even if this incurred a storage of one-bit per line, given that a large cache can have many millions of lines, the size of the MissMap quickly gets into the megabyte regime. Given the large size of the MissMap, it is better to avoid dedicated storage and store it in an already existing on-chip structure such as the L3 cache. Hence, it incurs a significant latency of L3 cache access (24 cycles). In this section, we will describe accurate predictors that incur negligible storage and delay. We lay the background for operating such a predictor before describing the predictor. The ideas described in this Section are derived from the prior work from Qureshi [15].

5.1. Serial Access vs. Parallel Access

The implicit assumption made in the LH-Cache study was that the system needs to ensure that there is a DRAM cache miss before accessing memory. This assumption is similar to how conventional caches operate. We call this the *Serial Access Model (SAM)*, as the cache access and memory access get serialized. The SAM model is bandwidth-efficient as it sends only the cache misses to main memory, as shown in Figure 7.

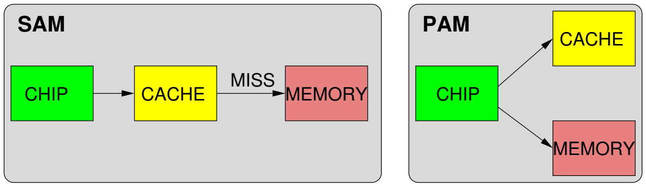


Figure 7: Cache Access Models: Serial vs Parallel

Alternatively, we may choose to use a less bandwidth efficient model, which probes both the cache and memory in parallel. We call this the *Parallel Access Model (PAM)*, as shown in Figure 7. The advantage of PAM is that it removes the serialization of the cache-miss detection latency from the memory access path. To implement PAM correctly though, we should give priority to cache content rather than the memory content, as cache content can be dirty. Also, if the memory system returns data before the cache returns the outcome of the tag check, then we must wait before using the data as the line could still be present in a dirty state in the cache.

At first blush, it may seem wasteful to access the DRAM cache in case of a DRAM cache miss. However, for both LH-Cache and Alloy Cache, the tags are located in DRAM. So, even on a DRAM cache miss, we still need to read the tags anyway to select a victim line and check if the victim is dirty (to schedule writeback). So, PAM does not have a significant impact on cache utilization compared to a perfect predictor.

5.2. To Wait or Not to Wait

We can get the best of both SAM and PAM by dynamically choosing between the two, based on an estimate of whether the line is likely to be present in the cache or not. We call this *Dynamic Access Model (DAM)*. If the line is likely to be present in the cache, DAM uses SAM to save on memory bandwidth. And if the line is unlikely to be present, DAM uses PAM to reduce latency. Note that DAM does not require *perfect information* for deciding between SAM and PAM, but simply a *good estimate*. To help with this estimate, we propose a hardware-based *Memory Access Predictor (MAP)*. To keep the latency of our predictor to a bare minimum, we consider only simple predictors.

5.3. Memory Access Predictor

The latency savings of PAM and the bandwidth savings of SAM depend on the cache hit rate. If the cache hit rate is very high, then SAM can reduce bandwidth. If the cache hit-rate is very low, then PAM can reduce latency. So, we can simply use cache hit rate for memory-access prediction. However, it is well known that both cache misses and hits show good correlation with previous outcomes [5] and exploiting such correlation results in more effective prediction than simply using the hit-rate. For example, if H is hit and M is miss, and the last eight outcomes are MMMMHHHH, then using the hit-rate would give an accuracy of 50%, but a simple last-time predictor would give an accuracy of 87.5% (assuming the first M was predicted correctly). Based on this insight, we propose to use *History-Based Memory-Access Predictors.*

5.3.1. Global-History Based MAP (MAP-G)

Our basic implementation, called *MAP Global* or *MAP-G*, uses a single saturating counter called the *Memory Access Counter (MAC)* that keeps track if the recent L3 misses resulted in a memory access or a hit in the DRAM cache. If the L3 miss results in a memory access, then the MAC is incremented, otherwise MAC is decremented (both operations are done using saturating arithmetic). For prediction, MAP-G simply uses the MSB of the MAC to decide if the L3 miss should employ SAM (MSB=0) or PAM (MSB=1). We employ MAP-G on a per-core basis and use a 3-bit counter for the MAC. Our results show that MAP-G bridges more than half the performance gap between SAM and perfect prediction. Note that because writes are not on the critical path (at this level, writes are mainly due to dirty evictions from on-chip caches), we do not make predictions for writes and simply employ SAM.

5.3.2. Instruction-Based MAP (MAP-I)

We can improve the effectiveness of MAP-G by exploiting the well-known observation that the cache hit/miss information is heavily correlated with the instruction address that caused the cache access [3, 8, 18]. We call this implementation *Instruction-Based MAP* or simply *MAP-I.* Instead of using a single MAC, MAP-I uses a table of MACs, called the *Memory Access Counter Table (MACT)*. The address of the L3 miss causing instruction is hashed (using folded-xor [17]) into the MACT to obtain the desired MAC. All predictions and updates happen based on this MAC. We found that simply using 256 entries (8-bit index) in the MACT is sufficient. The storage overhead for this implementation of MAP-I is 256*3-bit=96 bytes. We keep the MACT on a per-core basis to avoid interference between the cores (for eight cores, total overhead is only 96*8=768 bytes). Like MAP-G, MAP-I does not make predictions for write requests.

Note that our predictors do not require that the instruction address be stored in the cache. For read misses, the instruction address of miss causing load is forwarded with the miss request. As writeback misses are serviced with SAM, we do not need instruction addresses for writebacks.

5.4. Performance Results

Figure 8 shows the speedup from the Alloy Cache with different memory access predictors. If we use a prediction of always-cache-hit the system behaves like SAM, and if we use a prediction of never-cache-hit the system behaves like PAM. The perfect predictor assumes 100% accuracy at zero latency.

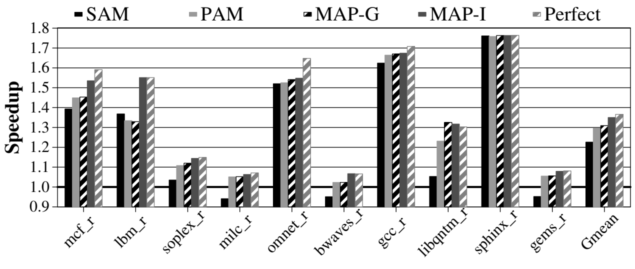


Figure 8: Performance improvement of Alloy Cache for different Memory Access Predictors

On average, there is a 14% gap between SAM (22.6%) and perfect prediction (36.6%). PAM provides 29.6% performance improvement but results in almost twice as many memory accesses as perfect prediction. MAP-G provides 30.9% performance, bridging half the performance difference between SAM and the perfect predictor. It thus performs similar to PAM but without doubling the memory traffic. MAP-I provides an average of 35%, coming within 1.6% of the performance of a perfect predictor. Thus, even though our predictors are simple ($<$ 100 bytes per core) and low latency (1 cycle), they get almost all of the potential performance.

For libquantum, MAP-G performs 3% better than the perfect predictor. This happens because some of the mispredictions avoid the row buffer penalty for later demand misses. For example, consider four lines A, B, C, D that map to the same DRAM row. Only A and B are present in the DRAM cache. A, B, C, D are accessed in a sequence. If A and B are predicted correctly, C would incur a row opening penalty when it goes to memory. If, on the other hand, A is mispredicted it would avoid the row opening penalty for C.

5.5. Prediction Accuracy Analysis

To provide insights into the effectiveness of the predictors, we analyzed different outcome-prediction scenarios. There are four cases: 1) L3 miss is serviced by memory and our predictor predicts it as such, 2) L3 miss is serviced by memory and our predictor predicts that it will be serviced by the DRAM cache, 3) L3 miss is serviced by the DRAM cache and our predictor predicts memory access, and 4) L3 miss is serviced by the DRAM-cache and our predictor predicts it to be so. Scenarios 2 and 3 denote mispredictions. However, note that the cost of mispredictions are quite different in the two scenarios (scenario 2 incurs higher latency and scenario 3 extra bandwidth). Table 5 shows the scenario distribution for different predictors averaged across all workloads.

Table 5: Accuracy for Different Predictors

	Serviced by Memory		Serviced by Cache		Overall
Prediction	Memory	Cache	Memory	Cache	Accuracy
SAM	0	51.8%	0	48.1%	48.1%
PAM	51.8%	0	48.2%	0	51.8%
MAP-G	45.1%	6.7%	10.8%	37.4%	82.5%
MAP-I	**48.3%**	3.5%	1.9%	**46.2%**	**94.5%**
Perfect	51.8%	0%	0%	48.2%	100%

PAM almost doubles the memory traffic compared to other approaches (48% of L3 misses are wastefully deemed to access memory when they are in-fact serviced by the DRAM-cache). Compared to a perfect predictor, MAP-I has higher latency for 3.5% of the L3 misses, and extraneous bandwidth consumption for 1.9% of the L3 misses. For the remaining 94.5% of the L3 misses, MAP-I prediction is correct. Thus, even though our predictors are quite simple, low-cost, and low-latency, they are still highly-effective, provide high accuracy, and obtain almost all of the potential for performance improvement from memory access prediction. Unless stated otherwise, the Alloy Cache is always implemented with MAP-I in the remainder of this paper.

5.6. Implications on Memory Power and Energy

Accessing memory in parallel with the cache, as done in PAM and conditionally in DAM, increases power in memory system due to wasteful memory accesses. For PAM, all of the L3 misses would be sent to off-chip memory. Whereas with SAM, only the misses in the DRAM cache would get sent to memory. From Table 5, it can be concluded that PAM would almost double the memory activity compared to SAM. Hence, we do not recommend unregulated use of PAM (except as a reference point). For DAM, our MAP-I predictor is quite accurate which means wasteful parallel accesses account for only 1.9% of L3 misses, compared to 48% with PAM.

6. Analysis and Discussions

6.1. Sensitivity to Cache Size

The default DRAM cache size for all of our studies is 256MB. In this section, we study the impact of different schemes as the cache size is varied from 64MB to 1GB. Figure 9 shows the average speedup with LH-Cache (29-way), SRAM-Tag (32-way), Alloy Cache, and IDEAL-LO. IDEAL-LO is the latency optimized theoretical design that transfers only 64 byte on a cache hit and has perfect zero-latency predictor.

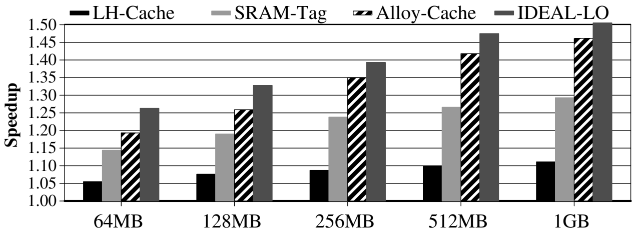


Figure 9: Performance impact across various cache size. Alloy-Cache continues to significantly outperform impractical SRAM-Tag and reaches close to the upperbound of IDEAL-LO.

The SRAM-Tag design suffers from Tag Serialization Latency (TSL). LH-Cache suffers from both TSL and PSL due to the MissMap. Alloy Cache avoids both TSL and PSL, hence it outperforms both the LH-Cache and SRAM-Tag across all studied cache sizes. For the 1GB cache size, LH-Cache provides an average improvement of 11.1%, SRAM-Tag provides 29.3%, and Alloy Cache provides 46.1%. Thus, Alloy Cache provides approximately 1.5 times the improvement of the SRAM-Tag design. Note that the SRAM-Tag implementation incurs an impractical storage overhead of 6MB, 12MB, 24MB, 48MB, and 96MB for DRAM cache sizes of 64MB, 128MB, 256MB, 512MB and 1GB, respectively. Our proposal, on the other hand, requires less than one kilobyte of storage, and still outperforms SRAM-Tag significantly, consistently reaching close to the performance of IDEAL-LO.

6.2. Impact on Hit Latency

The primary reason why the Alloy Cache performs so well is because it is designed from the ground-up to have lower latency. Figure 10 compares the average read latency of LH-Cache, SRAM-Tags, and Alloy Cache. Note that SRAM-Tags incur a tag serialization latency of 24 cycles, and LH-Cache incurs MissMap delay of 24 cycles in addition to the tag serialization latency (32-50 cycles). For the Alloy Cache, there is no tag serialization, except for the one additional bus cycle for obtaining the tag with dataline. The average hit latency for LH-Cache is 107 cycles. The Alloy Cache cuts this latency by 60%, bringing it to 43 cycles. This significant reduction causes Alloy Cache to outperform LH-Cache despite the lower hit rate. The SRAM-Tag incurs an average latency of 67 cycles, hence lower performance than the Alloy Cache.

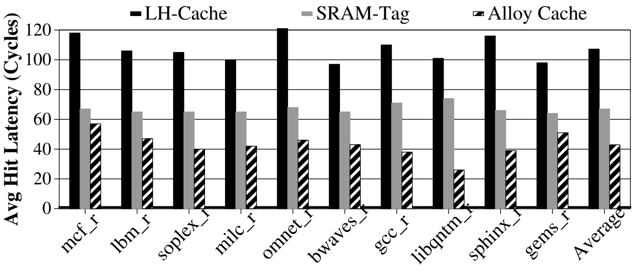


Figure 10: Average Hit-Latency: LH-Cache 107 cycles, SRAM-Tag 67 cycles, and Alloy Cache 43 cycles.

6.3. Impact on Hit-Rate

Our design de-optimizes the cache architecture from a highly-associative structure to a direct-mapped structure in order to reduce hit latency. We compare the hit rate of a highly-associative 29-way LH-cache with the direct-mapped Alloy Cache. Table 6 shows the average hit rate for different cache sizes. For a 256MB cache, the absolute difference in hit rates between the 29-way LH-Cache and direct-mapped Alloy Caches is 7%. Thus, the Alloy Cache increases misses by 15% compared to LH-Cache. However, we show that the 60% reduction in hit latency compared to LH-Cache provides much more performance benefit than a slight performance degradation from the reduced hit rate. Table 6 also shows that the hit-rate difference between a highly-associative cache and a direct-mapped cache reduces as the cache size is increased (at 1GB it is 2.5%, i.e., 5% more misses). The reducing gap between the hit rate of a highly-associative cache and direct-mapped cache as the cache size is increased is well known [6].

Table 6: Hit Rate: Highly associative vs. direct mapped

Cache Size	LH-Cache (29-way)	Alloy-Cache (1-way)	Delta Hit Rate
256 MB	55.2%	48.2%	7.0%
512 MB	59.6%	55.2%	4.4%
1 GB	62.6%	59.1%	2.5%

6.4. Other Workloads

In our detailed studies, we only considered memory-intensive workloads that have a speedup of at least 2x if L3 cache is made perfect (100% hit rate). Figure 11 shows the performance improvement from LH-Cache, SRAM-Tags, and Alloy-Cache for the remaining workloads that spend at least 1% of time in memory. These benchmarks were executed in rate mode as well. The bar labeled Gmean represents the geometric mean improvement over these fourteen workloads.

As the potential is low, the improvements from all designs are lowered compared to the detailed study. However, the broad trend remains the same. On average, LH-Cache improves performance by 3%, SRAM-Tag by 7.3%, and Alloy Cache by 11%. Thus, the Alloy Cache continues to outperform both LH-Cache and SRAM-Tag.

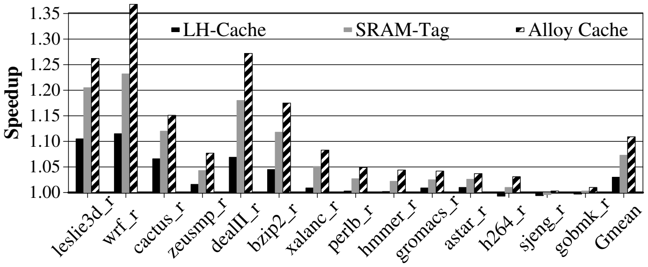


Figure 11: Performance impact for other SPEC workloads

6.5. Impact of Odd Size Burst Length

Our proposal assumes a burst length of five for the Alloy Cache, transferring 80 bytes on each DRAM cache access. However, conventional DDR specifications may restrict the burst length to a power-of-two even for stacked DRAM. If such a restriction exists, then the Alloy Cache can stream out burst of eight transfers (total 128 bytes per access). Our evaluation shows that a design with a burst of 8 provides 33% performance improvement on average, compared to 35% if the burst length can be set to five. Thus, our assumption of odd-size burst length has minimal impact on the performance benefit of Alloy Cache. Note that die-stacked DRAMs will likely use different interfaces than conventional DDR. The larger number of through-silicon vias could make it easier to provide additional control signals to, for example, dynamically specify the amount of data to be transferred.

6.6. Potential Room for Improvement

Our proposal is a simple and practical design that significantly outperforms the impractical SRAM-Tag design, but there is still room for improvement. Table 7 compares the average performance of Alloy Cache + MAP-I, with (a) perfect Memory Address Prediction (Perf-Pred) (b) IDEAL-LO, a configuration that incurs minimum latency and bandwidth and has Perf-Pred and (c) IDEAL-LO with no tag overhead, so all of the 256MB space is available to store data.

Table 7: Room for improvement

Design	Performance Improvement
Alloy Cache + MAP-I	35.0%
Alloy Cache + PerfPred	36.6%
IDEAL-LO	38.4%
IDEAL-LO + NoTagOverhead	41.0%

We observe that for our design we would get 1.6% additional performance improvement from a perfect predictor and another 1.8% from an IDEAL-LO cache. Thus, our practical solution is within 3% of the performance of an idealized design that places tags in DRAM. If we can come up with a way to avoid the storage overhead of tags in DRAM, then there is another 2.6% improvement possible. While all of the three optimizations show small opportunity for improvement, we must be mindful that solutions to obtain these improvements must incur minimal latency overheads, otherwise the marginal improvements may be quickly negated.

6.7. How About Two-Way Alloy Caches?

We also evaluated two-way Alloy Caches that stream out two TAD entries on each access. While this improved the hit-rate from 48.2% to 49.7%, we found that the hit latency increased from 43 cycles to 48 cycles. This was due to increased burst length (≈2x), associated bandwidth consumption (≈2x), and the reduction in row buffer hit rate. Overall, the performance impact of degraded hit latency outweighs the marginal improvement from hit-rate. We envision that future researchers will look at reducing conflict misses in DRAM caches (and we encourage them to do so); however, we advise them to pay close attention to the impact on hit latency.

7. Conclusion

This paper analyzed the trade-offs in architecting DRAM caches. We compared the performance of a recently-proposed design (LH-Cache) and an impractical SRAM-based Tag-Store (SRAM-Tags) with a latency-optimized design, and show that optimizing for latency provides a much more effective DRAM cache than optimizing simply for hit-rate. To obtain a practical and effective latency-optimized design, this paper went through a three-step process:

1. We showed that simply converting the DRAM cache from high associativity to direct mapped can itself provide good performance improvement. For example, configuring LH-Cache from 29-way to 1-way enhances the performance improvement from 8.7% to 15%. This happens because of the lower latency of a direct-mapped cache as well as the ability to exploit row buffer hits.

2. Simply having a direct-mapped structure is not enough. A cache design that creates a separate "tag-store" and "data-store" still incurs the tag-serialization latency even for direct-mapped caches. To avoid this tag serialization latency, we propose a cache architecture called the *Alloy Cache* that fuses the data and tag together into one storage entity, thus converting two serialized accesses for tag and data into a single unified access. We show that a direct-mapped Alloy Cache improves performance by 21%.

3. The performance of the Alloy Cache can be improved by handling misses faster, i.e., sending them to memory before completing the tag check in the DRAM cache. However, doing so with a MissMap incurs megabytes of storage overhead and tens of cycles of latency, which negated much of the performance benefit of handling misses early. Instead, we present a low-latency (single cycle), low storage overhead (96 bytes per core), highly accurate (95% accuracy) hardware-based *Memory Access Predictor* that enhances the performance benefit of Alloy Cache to 35%.

Optimizing for latency enabled our proposed design to provide better performance than even an impractical option of having the tag store in an SRAM array (24% improvement), which would require tens of megabytes of storage. Thus, we showed that simple designs can be highly effective if they can exploit the constraints of the given technology.

While the technology and constraints of today are quite different from the 1980's, in spirit, the initial part of our work is similar to that of Mark Hill [6] from twenty-five years ago, making a case for direct-mapped caches and showing that they can outperform set-associative caches. Indeed, sometimes *"Big and Dumb is Better"* [1].

Acknowledgments

Thanks to André Seznec and Mark Hill for comments on earlier versions of this paper. Moinuddin Qureshi is supported by NetApp Faculty Fellowship and Intel Early CAREER award.

References

[1] *Quote from Mark Hill's Bio (short link http://tinyurl.com/hillbio):*. https://www.cs.wisc.edu/event/mark-hill-efficiently-enabling-conventional-block-sizes-very-large-die-stacked-dram-caches.

[2] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support. In *Supercomputing*, 2010.

[3] M. Farrens, G. Tyson, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO-28*, 1995.

[4] M. Ghosh and H.-H. S. Lee. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In *MICRO-40*, 2007.

[5] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma. Cache miss behavior: is it sqrt(2)? In *Computing Frontiers*, 2006.

[6] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, Dec 1988.

[7] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Adaptive filter-based dram caching for CMP server platforms. In *HPCA-16*, 2010.

[8] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi. Using dead blocks as a virtual victim cache. In *PACT-19*, 2010.

[9] G. H. Loh and M. D. Hill. Addendum for "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches". http://www.cs.wisc.edu/multifacet/papers/micro11_missmap_addendum.pdf.

[10] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In *MICRO-44*, 2011.

[11] G. H. Loh and M. D. Hill. Supporting very large DRAM caches with compound access scheduling and missmaps. In *IEEE Micro TopPicks*, 2012.

[12] N. Madan, L. Zhao, N. Muralimanohar, A. Udipi, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell. Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy. In *HPCA-15*, 2009.

[13] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. *Computer Architecture Letters*, Feb 2012.

[14] E. Perelman et al. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[15] M. K. Qureshi. Memory access prediction. U.S. Patent Application Number 12700043, Filed Feb 2010, Publication Aug 2011.

[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. Adaptive insertion policies for high-performance caching. In *ISCA-34*, pages 167–178, 2007.

[17] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. In *Journal of Instruction Level Parallelism*, 2006.

[18] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer. Ship: signature-based hit predictor for high performance caching. In *MICRO-44*, 2011.

[19] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring DRAM cache architectures for CMP server platforms. In *ICCD*, 2007.