ATCache: Reducing DRAM-cache Latency via a Small SRAM Tag Cache

Citation for published version:
Huang, C-C & Nagarajan, V 2014, ATCache: Reducing DRAM-cache Latency via a Small SRAM Tag Cache. in PACT '14 Proceedings of the 23rd international conference on Parallel architectures and compilation. ACM, pp. 51-60. DOI: 10.1145/2628071.2628089

Digital Object Identifier (DOI):
10.1145/2628071.2628089

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Early version, also known as pre-print

Published In:
PACT '14 Proceedings of the 23rd international conference on Parallel architectures and compilation

ATCache: Reducing DRAM cache Latency via a Small SRAM Tag Cache

Cheng-Chieh Huang
Institute of Computing Systems Architecture
University of Edinburgh
cheng-chieh.huang at ed.ac.uk

Vijay Nagarajan
Institute of Computing Systems Architecture
University of Edinburgh
vijay.nagarajan at ed.ac.uk

ABSTRACT

3D-stacking technology has enabled the option of embedding a large DRAM onto the processor. Prior works have proposed to use this as a DRAM cache. Because of its large size (a DRAM cache can be in the order of hundreds of megabytes), the total size of the tags associated with it can also be quite large (in the order of tens of megabytes). The large size of the tags has created a problem. Should we maintain the tags in the DRAM and pay the cost of a costly tag access in the critical path? Or should we maintain the tags in the faster SRAM by paying the area cost of a large SRAM for this purpose? Prior works have primarily chosen the former and proposed a variety of techniques for reducing the cost of a DRAM tag access.

In this paper, we first establish (with the help of a study) that maintaining the tags in SRAM, because of its smaller access latency, leads to overall better performance. Motivated by this study, we ask if it is possible to maintain tags in SRAM without incurring high area overhead. Our key idea is simple. We propose to cache the tags in a small SRAM tag cache – we show that there is enough spatial and temporal locality amongst tag accesses to merit this idea. We propose the ATCache which is a small SRAM tag cache. Similar to a conventional cache, the ATCache caches recently accessed tags to exploit temporal locality; it exploits spatial locality by prefetching tags from nearby cache sets. In order to avoid the high miss latency and cache pollution caused by excessive prefetching, we use a simple technique to throttle the number of sets prefetched. Our proposed ATCache (which consumes 0.4% of overall tag size) can satisfy over 60% of DRAM cache tag accesses on average.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Design Styles—Cache memories

Keywords

DRAM cache, Design, Performance

To appear in the PACT 2014

1. INTRODUCTION

3D-stacking technology has enabled the option of embedding a large DRAM onto the processor. Prior works [18] have proposed to use this as a DRAM cache. Because of its large size (a DRAM cache can be in the order of hundreds of megabytes), the total size of the tags associated with it can also be quite large (in the order of tens of megabytes).

The large size of the tags has created a classic space/time trade-off issue. On the one hand, we would like the latency of a tag access to be small as it would contribute to both hit latency and miss latency. Accordingly, we would like to store these tags in a faster media such as SRAM (*tags-in-SRAM*). However, with hundreds of megabytes of die-stacked DRAM cache, the space overhead of the tags would be huge. For example, it would cost around 12 MB of SRAM space to store all the tags of a 256MB DRAM cache (if we used conventional 64B blocks). Clearly this is too large, considering that some of the current chip multiprocessors have an L3 that is smaller [6].

In order to solve the above problem, Loh and Hill [10] proposed an approach for storing the tags within the DRAM itself (*tags-in-DRAM*). To make this approach practical, they proposed a scheme for embedding the tag and data in the same row buffer, which would enable both tag and data to be accessed in a single *compound access*. Compared to a naive scheme, performing a compound access optimizes the hit latency by obviating the need to reopen a row to access data as shown in Fig. 2 (saving on *tACT*, the time to activate a row). However, it does not optimize the miss latency, since only the tag is accessed on a miss. To reduce the miss penalty, Loh and Hill also proposed a technique for tracking the contents of the DRAM cache in a structure called the *MissMap*. This enables them to avoid a DRAM tag access for misses, with only a few megabytes of SRAM overhead. Subsequently other works [14, 15] proposed miss predictors to achieve the same effect as a MissMap, but with significantly lesser SRAM overhead (in the order of a few kilobytes).

The impracticality of tags-in-SRAM, has led to the above tags-in-DRAM proposals. But how does the performance of recently proposed tags-in-DRAM techniques compare with tags-in-SRAM? We conduct a study to measure the average DRAM access latency for different configurations of DRAM cache, as shown in Fig. 1. Here, tags-in-SRAM refers to a scheme in which tags of all blocks in the DRAM are stored in the SRAM; tags-in-DRAM refers to a scheme that uses Loh and Hill's compound access; MissPred refers to the above, augmented with an oracle miss predictor (100% accuracy

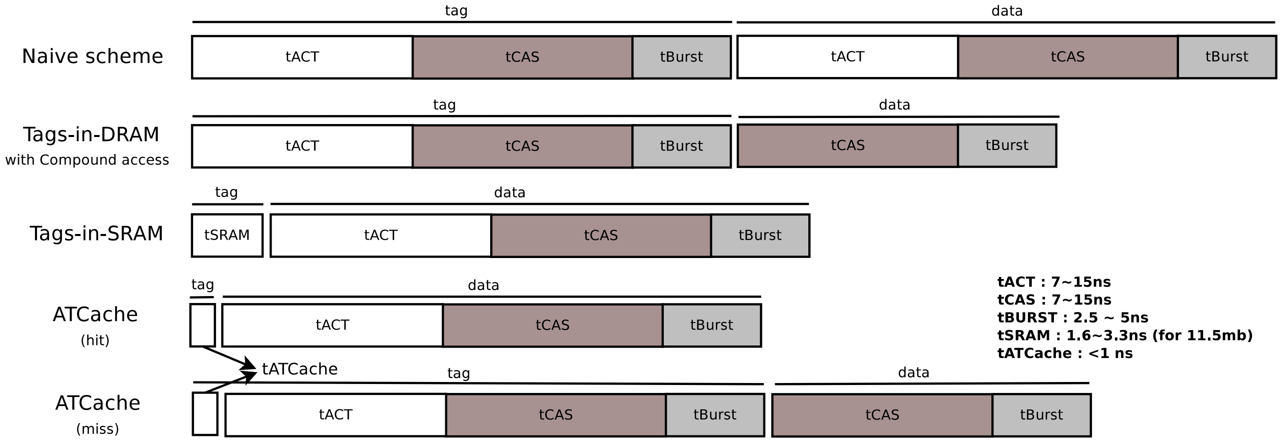

Figure 2: Access latency of different types of DRAM cache.

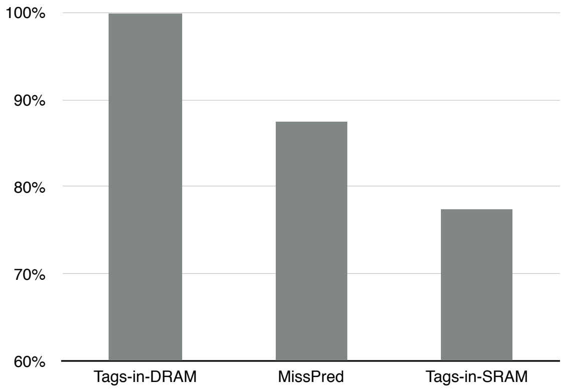

Figure 1: Average latency of the DRAM cache (including main memory access latency). The architectural parameters and workloads are shown in Table 5.

and zero latency). For this study, we use an SRAM access latency of 6 cycles which we modeled using CACTI (32nm, itrs-hp cell); further, we use 7ns for tACT and tCAS and 2.5ns for tBURST for stacked DRAM (other architectural parameters used in this study are shown in Table 5). All results are normalized to the access latency of tags-in-DRAM. As we can see, using the tags-in-SRAM configuration results in a 23.7% reduction in DRAM cache access latency. Furthermore, this latency is 10.2% lesser than MissPred (which is an oracle miss predictor with zero access latency).

Given that there is a significant gap in performance between tags-in-SRAM and tags-in-DRAM, we want to benefit from the tags-in-SRAM approach, but without incurring the cost of a high SRAM overhead. Our key idea is to maintain the tags in the DRAM cache, *but also cache a small amount of tags in SRAM in a dedicated cache.* We call this the *Aggressive Tag Cache (ATCache)*, as we will later show that we can achieve very good performance with a small tag cache. In addition to this, having a small cache is beneficial as this means that the ATCache can be accessed faster than a larger tags-in-SRAM design.

Like a conventional cache, ATCache exploits temporal locality by only caching the tags of recently accessed DRAM cache sets. In a similar vein, it exploits spatial locality by prefetching the tags of adjacent DRAM cache sets. One potential problem is to know how many sets' tags to prefetch, as prefetching too much can lead to cache pollution and also increased miss penalty. To deal with this, we propose a scheme in which we prefetch tags of the adjacent sets only if there is evidence of spatial locality amongst the sets.

We evaluate our approach on the gem5 cycle-accurate simulator [1]. On memory-intensive single-threaded SPEC 2006 benchmarks, our ATCache can achieve 10.3% performance improvement on average compared to a tags-in-DRAM (with compound access) baseline. This compares favorably with the speedup of 6.8% provided by adding a high-accuracy miss predictor (MAP-I) to tags-in-DRAM. Our ATCache can also be integrated with miss predictors. Indeed, on multiprogrammed workloads, our ATCache is 9.3% faster than the tags-in-DRAM baseline, but 10.9% faster when we integrate a miss predictor (MAP-I) to ATCache. Finally, our results are almost as good as a full tags-in-SRAM cache, which provides a performance improvement of 11.9%. Our ATCache only uses 47.375kB space (including the overheads) and around 50kB if we include the MAP-I as part of our design. Therefore, the overall SRAM space consumed is only around 0.5% of a tags-in-SRAM design.

2. BACKGROUND

2.1 DRAM cache

Reducing the latency of tag accesses is fundamental to cache design, as this latency would be added to critical path of the total latency, no matter a hit or miss. Prior works [9, 10, 18] have proposed to use stacked DRAM as a large cache inside the processor chip. By exploiting the high density of stacked DRAM, we are able to have multiple hundreds of megabytes of on-chip DRAM cache. Ideally, we want to architect this large cache with same cache block size as conventional L1 or L2 (which typically use a block size of 32/64 bytes). Consequently, with multiple hundreds of megabytes of DRAM cache, the overhead of storing tags of conventional blocks requires tens of megabytes [3, 9, 10]. In this case, it would be impossible to naively use SRAM (which provides faster access latency at the cost of low density) as storage media.

Table 1: Tag sizes/latencies for different cache sizes.

Cache size	128MB	256MB	512MB	1024MB
Tag size (per block)	24 bits	23 bits	22 bits	21 bits
Total tag size	6MB	11.5MB	22MB	42MB
Latency/hp (cycles)	5	6	7	8
Latency/lstp (cycles)	9	10	12	13

Page-based DRAM cache: A page-based cache (i.e. a much larger cache block size of 2kB to 4kB) has been proposed in several works [8, 9]. With a larger block size, the tag overhead is reduced to hundreds of kilobytes or few megabytes depending on the cache size. With this cache design, there are two major challenges that prior works have addressed. First, a larger cache line size means more data to fetch on a miss. Generally, a DRAM cache miss needs to fetch data from a low-bandwidth off-chip memory. This downside becomes one of the performance bottlenecks of a page-based design. Second, a larger cache line might fetch a significant amount of unused data to the cache. This decreases the effective capacity of the DRAM cache compared to a conventional block size design. To solve the bandwidth problem, Jevdjic et al. [8] proposed a footprint cache which removes the unused blocks from the (page-based) cache line and shows an effective reduction in bandwidth. Their design enables paged-based DRAM cache to outperform block-based caches (tags-in-DRAM, with MissMap) in server workloads [5]. However, in their study, desktop workloads (SPEC 2006) shows about 25% of slow down compared to block-based DRAM cache (with MissMap, 256MB). This is because the spatial footprint in desktop workloads is generally lower than server workloads like web search.

Block-based DRAM cache: Loh and Hill [10, 11] proposed a DRAM cache with conventional block size (64 bytes). In their work, they embedded tags along with their data in the same row buffer (tags-in-DRAM). They architect a 29-way DRAM cache with a 2kB row buffer (29 tags and 29 blocks). One issue with this design is that DRAM cache misses have to pay the high cost of accessing the tags from the DRAM; consequently they also proposed a MissMap, which is a SRAM structure which tracks the contents of DRAM cache in order to skip miss accesses. Since MissMap consumes a reasonably large amount of SRAM (in the order of few megabytes) to maintain the required information, subsequent works [14, 15] proposed miss predictors to predict cache misses with a lower SRAM overhead (in the order of few kilobytes). Subsequently, Qureshi and Loh [14] pointed out that instead of a sequential access of the data and tag, one could use a wider data width (72 bytes) to read tag and data in parallel. In this design, the hit latency of a direct-mapped DRAM cache is close to the latency of accessing only the data. Whereas this approach works great for direct-mapped caches, the downside of this design, however, is its inflexibility in scaling to set associate caches. To support a 4-way cache, for example, every access requires additional 3 tBURSTs for reading 3 more blocks from DRAM cache; assuming a 2.5 ns tBURST for reading a 72-byte block, the additional latency overhead of supporting a 4-way cache is 23 cycles for a 3 GHz processor.

2.2 Tag size and access latency

In this section, we show how we measure the tag sizes of different DRAM cache configurations and also their latencies. Generally, a *tag* for a block refers to several SRAM bits of storage to accommodate not only the tag for that cache block, but also status bits (dirty/valid and cache coherence states) and state associated with the replacement policy. In our system model, the DRAM cache is located a level below the cache coherent shared cache. The minimum requirement of the status register is 2 bits (valid and dirty bits). The number of tag bits depends on processor's addressing capability and the associativity of cache. For a processor with 40-bit address space and 64-byte cache line, a 256MB/16-way cache requires 17 bits for the tag. Let us assume the state associated with the replacement policy requires 4 bits. The overall space requirement for each cache block then is 23 bits, which amounts to around 11.5 MB in total.

We model the latency of tag access using CACTI 6.5 [13]. Specifically, we use 32nm technology and two types of SRAM cell (itrs-hp and itrs-lstp, which represent high performance configuration and low standby power configuration respectively). We report tag latencies for a 3GHz processor cycle. The results are shown in Table 1. As we can see, it shows that even with a low standby power cell, a 11.5MB SRAM tag access (10 processor cycles) can still be faster than minimum tBURST latency (which is about 4 memory cycles which is about 5ns or 15 processor cycles) in DDR3-1600 in DRAM. Substituting the values for tag latencies computed above, it is clear (as shown in Fig. 2) that tags-in-SRAM can provide both better hit (tag + data) and miss (tag) latency than tags-in-DRAM with compound access. In this paper, we will use the above modeled parameters in Table 1 for the tags-in-SRAM approach.

3. OUR METHODOLOGY AND DESIGN

As our study in Fig. 1 shows, a tags-in-SRAM design provides an opportunity to improve both hit and miss latency. The SRAM overhead, however, is a major impediment that limits its practicality. In this paper, we propose a hybrid method in which we maintain full tags in DRAM like Loh and Hill's work [10] but also cache a small number of tags in our proposed cache structure called *ATCache.* To put it simply, similar to a conventional cache which *caches data from memory*, our ATCache *caches the tags of the DRAM cache.*

3.1 Terminology

Before describing our approach, we first define several terminologies that can help explain our design.

SetTag: A SetTag refers to the set of tags of the blocks which belong to the same cache set of a DRAM cache. For example, if the DRAM cache is 16-way associative, then SetTag refers to all the 16 tags in each set. It is worth noting that all 16 tags are necessary to check for a hit or miss in DRAM cache, which is why they are cached together in the ATCache. In other words, a SetTag of an ATCache is analogous to a word of a conventional cache.

SetID: A SetID is an identifier for ATCache to identify if the SetTag for the correct set exists in the ATCache. In other words, a SetID of an ATCache is analogous to a tag of a conventional cache.

Caching ratio: The Caching ratio of an AT cache is the ratio of the size of the ATCache to the total size of the tags required by the DRAM cache. For example, a 256MB/16-way DRAM cache requires 11.5 MB for tags. A caching ratio of 256 corresponds to an ATCache of size 46KB.

Prefetching granularity (PG): In a conventional cache, whenever a word is accessed we also (pre)fetch additional adjacent words belonging to a block to exploit spatial locality. In a similar vein, the prefetching granularity (PG) refers to the number of adjacent SetTags that the ATCache will fetch on a miss. In other words, the PG of an ATCache is analogous to a block size of a conventional cache.

3.2 Locality of tag accesses

The idea of caching is founded on the principles of spatial and temporal locality, which a conventional cache exploits. In this section, we want to examine if spatial and temporal locality exists for tag data accesses also. To this end, we conduct a quick study in which we use a 46kB ATCache (1/256 of total tag size) to store recently accessed tags. Because tags of the same cache set would be accessed together, we store these tags in units of a DRAM cache set (and call it SetTag). In addition to this, to exploit spatial locality we fetch the SetTag of adjacent sets on an ATCache miss; we use the term *prefetching granularity* (PG) to refer to the number of adjacent cache sets that the ATCache will fetch on a miss.

Fig. 3 shows the average miss ratio of the ATCache for different PGs across all single-thread benchmarks that we studied (§ 4) in SPEC 2006. From the figure, the miss ratio of PG/2 is around 63.4%. This means that more than 30% of the tag accesses can be satisfied (hit) by our ATCache for a PG of 2, which is significantly better than a uniformly distributed hit ratio (1/256 $\simeq$ 0.4%). Furthermore, as the PG is increased from 2 through 64, the miss ratio decreases from 63.4% to 20.5%. These all indicate the existence of locality amongst tag accesses.

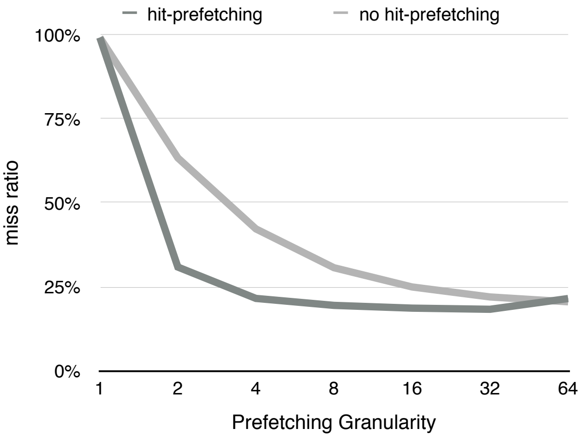

Figure 3: Miss ratio with various PG.

3.3 Hit prefetching

In our prior study, we found that the prefetching granularity (PG) is critical to the performance of ATCache. Prefetching nearby SetTags to the ATCache, generally speaking, is beneficial to miss ratio; however, a large PG can possibly pollute the ATCache which could result in a miss ratio increase, in addition to increasing miss-penalty. We provide a simple solution to achieve a balance between spatial locality, cache pollution, and miss-penalty. We start by fetching the SetTag for a relatively small number of cache sets (say PG/4). Corresponding to the SetTag of each fetched cache set, we maintain a flag that tracks whether or not the SetTag of the current cache set is accessed. Even if one of the prefetched SetTags is actually accessed, we prefetch the SetTags of the next contiguous 4 cache sets – in doing so, we achieve the effect of a larger PG. In case none of the prefetched SetTags are accessed, we do not prefetch any additional SetTags – in doing so, we avoid paying the space and time costs of a larger PG when a larger PG is not beneficial. It is worth noting that the space overhead of this technique is very small. It only requires *one additional bit* in the ATCache's SetID. We conducted a simple experiment to estimate the benefit of hit prefetching; as shown in Fig. 3, we find that with hit prefetching the miss ratio of PG/4 (21.5%) is able to match the miss ratio of PG/32 (22%) without hit prefetching. Since hit prefetching provides a significant benefit, we include it in our baseline system.

3.4 Size, hit ratio and latency

Another important factor to consider in cache design is the interplay between size and the access latency. In most cases, a larger cache size can provide a better hit ratio, but would also mean a higher access latency. In this section, we want to study how the cache capacity affects the ATCache's hit ratio and its latency. Fig. 4 shows the hit ratio for different caching ratios. For this experiment, we use PG/4 with hit prefetching turned on. As we can see, the miss ratio degrades gracefully as the caching ratio is increased. Even with an ATCache size of 11.2KB (caching ratio of 1024), the ATCache can still satisfy over 50% of tag accesses. On the other hand, an ATCache of only 11.2KB will enable it to have a faster access latency of close to 1 cycle in comparison to about 6 cycles for a full tags-in-SRAM design (the latency values are computed using CACTI as discussed earlier in the background section). The reduced tag access latency contributes to better DRAM cache performance as illustrated in Fig. 2.

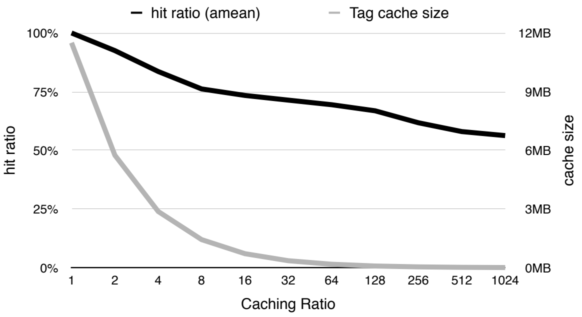

Figure 4: Hit ratio, size for different caching ratios.

3.5 Integration with miss predictor

Prior works have proposed miss predictors [10, 14, 15] which help to improve the performance of a tags-in-DRAM design by reducing the miss penalty. More specifically, they help avoid a DRAM tag access for (what is predicted to be) a DRAM cache miss. Although the latency to access a tag in our design is not as high as the tags-in-DRAM design (since the tags are now stored in the ATCache which is in the SRAM), we can still benefit from a miss predictor. In addition to this, because miss accesses can be handled by the high accuracy predictor, they can skip the ATCache. This means that the ATCache does not need to store the tags corresponding to misses anymore, which in turn increases the effective cache capacity of the ATCache. Therefore, we implement two predictors proposed in prior works – HMP [15] and MAP-I [14]. In our study, we found that both predictors can provides a very high prediction accuracy (Fig.5) with very small space overhead as prior works have observed. In our design we choose to integrate with MAP-I as it provides marginally better predictability.

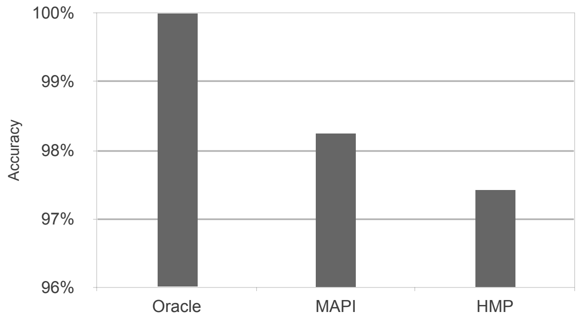

Figure 5: Accuracy of prior proposed predictors.

3.6 Design of ATCache

The design of our ATCache follows the design principles of a conventional cache: the full tags in the DRAM array represents the "main memory" and tags in the ATCache represents the "cached data". The procedure to access the ATCache is illustrated in Table 2 and the access logic is shown in Fig. 6. As we can see, the ATCache requires an additional SetID checking (Step 2A) – which is similar to a tag check for a conventional cache. Now, this represents an additional check compared to a conventional full tags-in-SRAM design. However, it is worth noting that this (step 2A) can be overlapped with step 2B (which is the tag check for DRAM cache). Therefore, the ATCache does not need additional access cycles for an ATCache hit in comparison to a full tags-in-SRAM design.

However, on an ATCache miss, the tags in DRAM have to be accessed and fetched back into the ATCache. So miss processing for ATCache is comparable to a tags-in-DRAM cache, with one small difference. Since step 2A is still required to identify if ATCache contains the correct SetTag, we incur one cycle penalty (step 2A is assumed to take one cycle) to the total access latency when there is an ATCache miss.

3.7 Putting it all together

We illustrate how the ATCache works with a spatial predictor and a miss predictor (MAP-I), with the help of an example (Table 3). This example consists of 5 consecutive DRAM cache accesses and each of them is accessing different cache sets (#sets). In the 1st access, the outcome of MAP-I is a hit which means DRAM cache might contain this data. Then, the system accesses the ATCache but it does not contain the corresponding SetTag. Therefore, similar to the tags-in-DRAM approach, an ATCache issues a DRAM compound access for tag and data. The ATCache will also fetch the adjacent cache sets after the compound access. Later, we can see these prefetched adjacent cache sets are accessed in the 3rd access. It is worth noting that a hit prefetching event (§3.3) also is generated in the 3rd access. In the 2nd access, we show a situation in which MAP-I is predicting the access as a miss. In this case, system will skip the access of DRAM cache because of the prediction result and directly fetch data from main memory. In the 4th access, a prefetched set (set 3) is accessed but the sets 4-8 is already brought back by the 3rd access. Therefore, there is no additional access required for our design. Finally, the 5th access is a MAP-I hit and an ATCache miss. Similar to 1st access, the system will issue a compound access and fetch adjacent sets.

Table 2: ATCache access procedure (refer to Fig. 6).

Steps	Description
Step 1	Locate SetID and SetTag in ATCache by a subset of #cache_set (#sub_cache_sets).
Step 2A	Check SetID to determine if ATCache set contains SetTag.
Step 2B	Check SetTag to determine if DRAM cache contains requested data (DRAM cache hit/miss).
Step 3	If step 2A is hit, use step 2B's result to determine hit/miss in DRAM cache. Otherwise, issue compound access to DRAM cache.

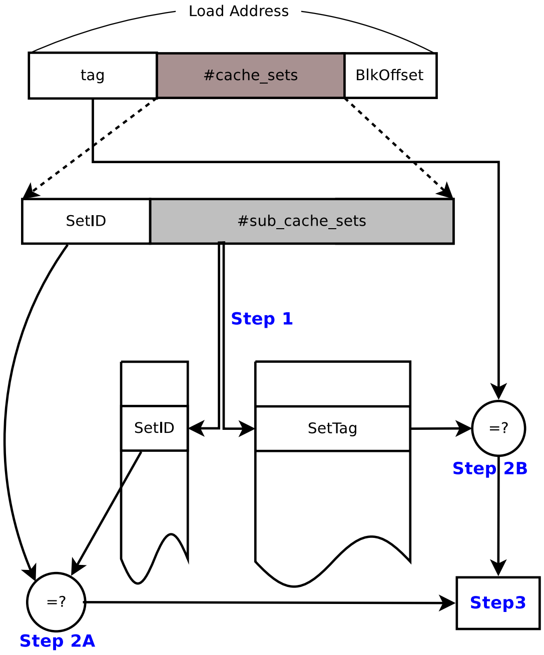

Figure 6: The access logic (also refer to Table 2).

3.8 Area overhead

In this paper, we use an ATCache with a caching ratio of 256. In other words, we use a 46kB cache for caching total

Table 3: Example showing 5 DRAM cache accesses.

#access	#set	MAP-I	ATCache	description
1	0x1	hit	miss	DRAM compound access and prefetch set 0,2,3 due to prefetching granularity (PG)
2	0x53	miss	X	predicted miss from MAP-I, skip the access
3	0x2	hit	hit	prefetch set 4-8 (hit prefetching)
4	0x3	hit	hit	access data in DRAM cache
5	0x10	hit	miss	DRAM compound access and prefetch set 0x11,0x12,0x13 (same as 1st access)

tags amounting to 11.5MB (256MB/16-way DRAM cache). As shown in Fig. 6, the additional overhead of a ATCache is a bunch of SetIDs. The size for each SetID actually depends on caching ratio and the associativity of the ATCache. Each ATCache block with caching ratio of 256 and associativity of 4 needs $8 + 2$ bits for storing SetID. In addition to SetID, to implement hit prefetching (§3.3, 1 bit per set), the overhead of SetID for each ATCache block slightly increases to 11 bits. In a 256MB/16-way cache, there are 256k cache sets. An ATCache with a caching ratio of 256 would cache only 1k ($256k/256$) sets. Therefore, the overhead of ATCache is around 1.375kB ($11 * 1024/8$ bytes). In total, the ATCache requires about 48kB SRAM space which is less than 0.5% SRAM space compared to tags-in-SRAM's 11.5MB. Table 4 shows the overhead for different cache sizes.

Table 4: Overhead for different cache sizes.

Cache Size	128MB	256MB	512MB	1024MB
Total tag size	6MB	11.5MB	22MB	42MB
ATCache space	24kB	46kB	88kB	168kB
ATCache overhead	704B	1.375kB	2.75kB	5.5kB

4. EXPERIMENTAL METHODOLOGY

4.1 Baseline system

We use the gem5 cycle-accurate simulator [1] in which we consider the L3 to be the DRAM cache; accordingly, we implement the DRAM timing model for the L3. The system parameters that we used are shown in Fig. 5. We show single-threaded results for 11 benchmarks from SPEC 2006 which are considered to be memory-intensive in prior works [14, 15]. In addition to single-threaded benchmarks, we also use the same 11 benchmarks to generate 25 multi-programmed workloads (4-core, as shown in Table 6) and evaluate their performance.

4.2 DRAM cache organizations

In this paper, we evaluate the following DRAM cache designs:

Baseline (Tags-in-DRAM): The DRAM cache design we used as baseline follows Loh an Hill's work [10]. The SetTag and their data are stored in the same row. With compound scheduling, a delay of opening a row ($\simeq$ tRCD) can be saved compared to storing them in separate rows. Our baseline system uses a 4kB row buffer which is close to a realistic DRAM organization (1kB per device, 4 devices per rank). In our system, a 4kB row buffer can store 4 15-way cache sets (4 x 15 x 64 bytes) and 4 SetTags (4 x 45 bytes). We use an open page policy to manage DRAM banks because we found open page policy gives better performance for our baseline.

Table 5: System parameters

Processor	3GHz, 4-core, 4-issue OoO, 64 ROB
L1 I/D caches	each 32kB/2way, LRU, 2-cycle, private
L2 cache	4MB/8way, 9-cycle, LRU, shared
Stacked DRAM	256MB, 2-bit SRRIP [7] tRCD-tCAS-tRAS 7-7-25 (ns) tBURST: 2.5ns 16 banks per rank, 1 rank per channel, 4 channel 4kB row buffer, open-page policy
Off-chip DRAM	800MHz (DDR3-1600), x64 interface tRCD-tCAS-tRAS 13.5-13.5-40.5 (ns) tBURST: 5ns 8 banks per rank, 2 ranks per channel, 1 channel 8kB row buffer, open-page policy
On-chip bus	3GHz, 256-bit width
System Bus	1.5GHz, 64-bit width
Miss Predictor	MAP-I [14], 256 entries
ATCache	Caching Ratio:256 (47.375kB incl. overhead) 4-way, 2-cycle latency, hit prefetching
Benchmarks	milc, soplex, omnetpp, gcc, leslie3d, GemsFDTD astar, mcf, bwaves, lbm, libquantum

Table 6: Workload groupings

1-2	soplex-astar-lbm-mcf	lbm-omnetpp-leslie3d-bwaves
3-4	milc-leslie3d-leslie3d-gcc	milc-libquantum-bwaves-gcc
5-6	libquantum-lbm-soplex-libquantum	libquantum-GemsFDTD-soplex-milc
7-8	gcc-milc-libquantum-astar	milc-soplex-bwaves-libquantum
9-10	leslie3d-omnetpp-leslie3d-mcf	lbm-astar-leslie3d-libquantum
11-12	leslie3d-leslie3d-libquantum-milc	mcf-gcc-milc-astar
13-14	omnetpp-libquantum-milc-soplex	gcc-libquantum-libquantum-soplex
15-16	soplex-GemsFDTD-omnetpp-milc	milc-soplex-leslie3d-libquantum
17-18	lbm-libquantum-omnetpp-bwaves	gcc-milc-leslie3d-milc
19-20	omnetpp-omnetpp-libquantum-leslie3d	soplex-mcf-gcc-libquantum
21-22	astar-omnetpp-astar-gcc	mcf-soplex-astar-leslie3d
23-24	bwaves-lbm-libquantum-leslie3d	astar-leslie3d-lbm-mcf
25	bwaves-soplex-bwaves-GemsFDTD	

NoDRAM (No DRAM cache): In this setting, we remove the DRAM cache from our cache hierarchy. The purpose of this setting is to examine if our system has any performance benefit from the DRAM cache in the first place.

MAP-I: As mentioned in §3.5, we use a MAP-I predictor for predicting misses. We implement a MAP-I predictor with 256 memory access counter (MAC) entries as suggested in prior work [14].

SRAM (Tags-in-SRAM) We use a tags-in-SRAM design which requires 11.5MB (§2.2) SRAM space to store all tags for 256MB DRAM cache. The access latency of the SRAM array is 6 cycles (high performance cell) as we modeled in §2.2. It is worth noting that the high SRAM overhead makes this design impractical.

ATCache: In our proposed design, we use an ATCache with *caching ratio of 256, prefetching granularity (PG) of 4, and associativity of 4*. This means we use $11.5MB/256 =$ 46 kB SRAM tag with 1.375kB overhead. We assume a single cycle latency to identify if the ATCache contains a correct cache set (step 2A in §3.6). A single cycle latency is reasonable considering that the SetID array is only 1.375kB

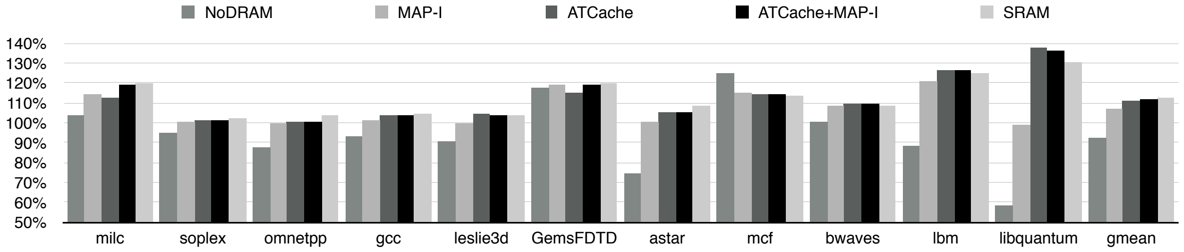

Figure 7: Performance speedup.

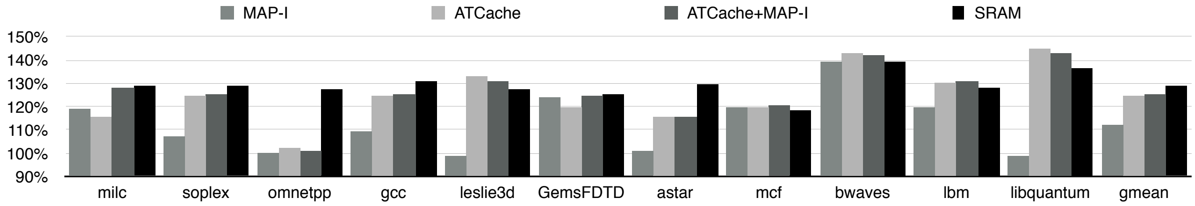

Figure 8: L2 miss latency reduction (higher the better).

in our design. Since the SRAM array for the ATCache is only 46 kB and step 2A and step 2B in §3.6 can be overlapped, we use a 2 cycles for the ATCache hit latency. For a miss in the ATCache, we add a *a 1 cycle lookup latency* (Step 2A) to the total access latency. It is worth noting that similar to miss-predictors in prior works [14], the *ATCache does not cache SetTags for DRAM cache writes*. This is because such writes are not in the critical path of the performance. In summary, in this design we use less than 0.5% of the SRAM space used by tags-in-SRAM.

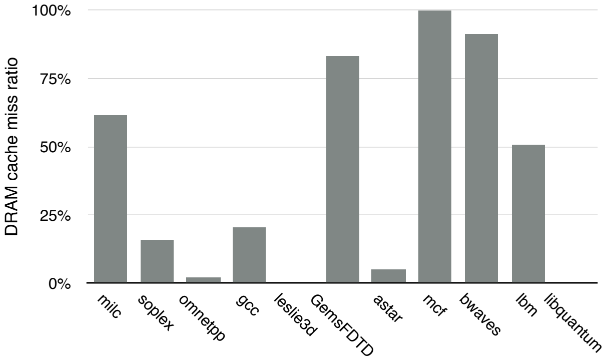

Figure 9: L3 miss ratio.

5. RESULTS

5.1 Performance

The IPC speedups (normalized to the baseline) are shown in Figure 7. From the results, we can see that a system without DRAM cache is 8.6% slower than the tags-in-DRAM baseline; from this we can conclude that the DRAM cache is effective for our system configuration.

We can also observe that a prior proposed miss predictor (MAP-I) improves over the baseline by 6.8%. In contrast, a tags-in-SRAM approach which consumes an impractically large SRAM space (11.5MB) provides a speedup of 12.2%.

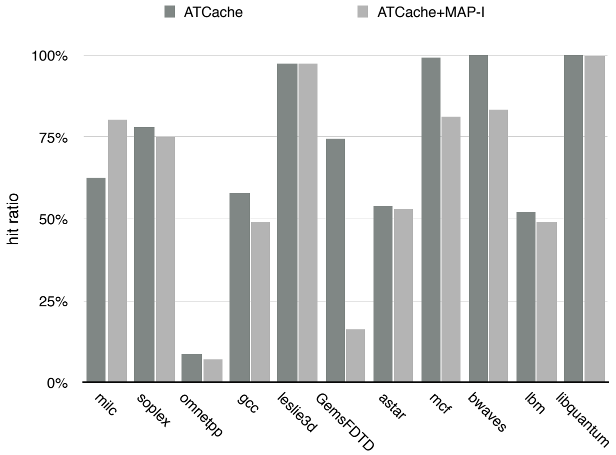

Figure 10: ATCache hit ratio.

It is worth noting that this is the gap that the ATCache tries to bridge, while consuming much smaller SRAM space.

We can see that the ATCache achieves 10.3% speedup without MAP-I predictor and 11.9% with the predictor – while consuming only 47.375kB (without MAP-I) and 48kB with MAP-I (256 entries). In other words, we are able to do almost as well as tags-in-SRAM while consuming 0.5% SRAM overhead compared to a tags-in-SRAM design.

In order to understand the speedups in individual benchmarks, we show the miss ratio of each benchmark program in Fig. 9. One interesting aspect to note here is that, by skipping miss accesses, MAP-I can provide a good speedup and even outperforms a tags-in-SRAM design for high miss ratio workloads such as *mcf* and *bwaves*. This is because the latency of accessing MAP-I structure is only one cycle (in comparison to 6 cycles access latency we used for a full tags-in-SRAM cache). However, for benchmarks with a low miss ratio (such as *libquantum*), MAP-I is not as effective and even shows a small slowdown due to its cycle lookup penalty.

On the other hand, with our ATCache we can observe benefit on both high and low miss ratio benchmarks. This is because ATCache reduces both hit latency and miss penalty. For example, for *libquantum* which has a low miss ratio, we are able to achieve a speedup of 38% over the baseline.

When we integrate MAP-I and ATCache together, we can see a speedup boost in moderate miss ratio workloads such as lbm and milc. This is because ATCache's average hit ratio is around 60% (as shown in Fig. 10), but the high accuracy of the MAP-I predictor (with a predictability of close to 98%) can help us further in reducing the DRAM miss penalty. In addition to this, with MAP-I the ATCache can be more effectively used for caching only hit tags. This is exemplified by *milc* benchmark where using an ATCache with MAP-I results in a speedup of 28.9% in comparison with a speedup of 18.9% (MAP-I only) and 11.8% (ATCache only).

However, for low miss ratio benchmarks such as *libquantum*, ATCache+MAP-I shows a small 1% slowdown compared to only ATCache design. This can be attributed to the increased hit access latency (extra cycle) for looking up the MAP-I table.

5.2 L2 miss latency

Fig. 8 shows the L2 miss latency reduction (normalized to the baseline) for all benchmarks. The L2 miss latency here refers to the time it takes for a cache block to be transferred to the L2 from the lower levels upon an L2 miss; in other words, this only includes the L3 (DRAM cache) latency and possibly the main memory latency (in case of a DRAM cache miss). This latency provides us with a more transparent indicator of the benefit of our proposal as it avoids the obfuscating effects of other parameters (such as the effects of the our-of-order processor). As we can see from Fig. 8, with our ATCache we can see up to 45.3% (21.2% on average) reduction in L2 miss latency compared to the baseline. It is worth noting that this is approximately double the reduction provided by MAP-I, which provides a reduction of 11.9% on average. Finally, ATCache+MAP-I is able to provide the maximum reduction over the baseline of 24.5%. From these results, we believe that our technique can effectively improve the performance of the DRAM cache.

5.3 Sensitivity towards caching ratio

In this section, we want to understand the effect of varying the caching ratio. As introduced in §3.1, caching ratio represents the area gain of our technique in comparison to tags-in-SRAM. A caching ratio of 1 refers to a tags-in-SRAM design. Generally, the miss ratio of the ATCache will increase when caching ratio is increased (smaller cache). Fig. 11 shows IPC speedup (average of all benchmarks) for different caching ratios. Here, "IPC (fixed latency)" refers to a configuration where we fixed the access latency of the ATCache to 6 cycles. The speedup in terms of IPC reduces from 12.2% to 9.0% when caching ratio is increased from 1 (tags-in SRAM, 11.5MB) to 256 (≃48kB). Even without accounting for (a) faster access latency that a smaller cache could provide and (b) the miss predictor, we believe this trade-off is still interesting – a reduction in 99.5% space for 3.2% reduction in speedup. "IPC (MAP-I + actual latency)" refers to the realistic configuration in which we consider the effect of a faster SRAM latency due to a much smaller cache (Table 7 shows latencies for different caching ratios) and also the MAP-I predictor. As we can see, the peak speedup in this configuration (ATCache+MAP-I) is for a caching ratio of 64 and the speedup is able to match the speedup of a tags-in-SRAM design (12.2%).

Table 7: Latency in different caching ratios – caching ratio of 1 equals tags-in-SRAM (6 cycles).

Caching Ratio	2	4	8	16	32
latency (cycles)	5	5	4	4	3
Caching Ratio	64	128	256	512	1024
latency (cycles)	3	2	2	1	1

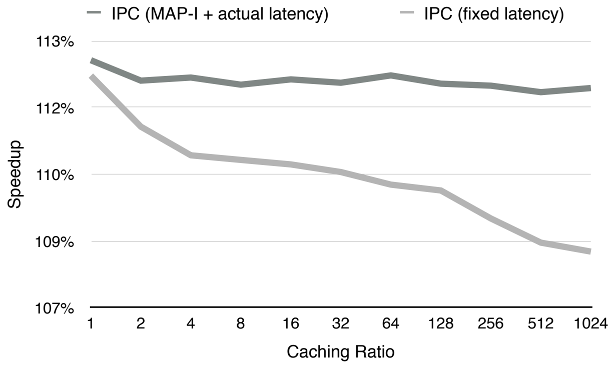

Figure 11: Speedups for different caching ratios. *Caching Ratio: 1* means a *tags-in-SRAM* design.

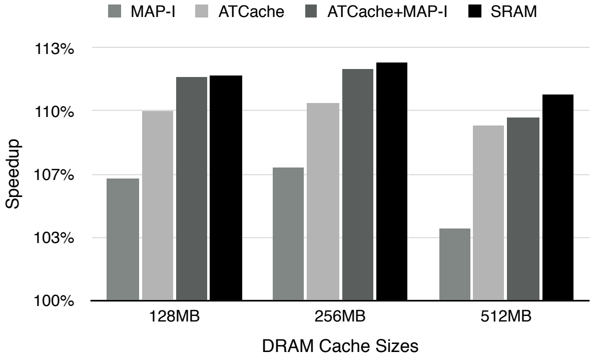

Figure 12: Sensitivity study of DRAM cache sizes.

5.4 Sensitivity towards DRAM cache size

Fig. 12 shows the effect of varying DRAM cache size. We consider 3 different sizes (128MB, 256MB, and 512MB). In this experiment, we use the latencies that we modeled in Table 1 for tags-in-SRAM. For ATCache (we fix caching ratio at 256), we use ATCache latency of 1, 2, and 3 cycles for ATCache size of 24kB (for 128MB DRAM cache), 46kB (for 256MB DRAM cache), and 88kB (for 512MB DRAM cache). From the results, we can see that the ATCache has a 9% to 10% performance improvement over its baseline (tags-in-DRAM) for all sizes. With MAP-I predictor, ATCache+MAP-I is 4% to 5% better than the MAP-I-only.

5.5 Sensitivity towards PG

As our prior studies (§3.2 and §3.3) show, the prefetching granularity (PG) is a key parameter in our design. In this section, we study the effect of varying the PG. In this study, we represent the performance of DRAM cache in terms of its average access latency, normalized to the access latency for PG/1. MAP-I is integrated with the ATCache for this study. As we can see from Fig. 13, increasing the PG from PG/1 to PG/2 , reduces the average latency of DRAM cache by 8.7%. The average latency reaches a minimum for PG/4 (10.1% reduction) and PG/8 (10.5% reduction). When we keep increasing PG, however, the ATCache starts to suffer from cache pollution and increased miss-penalty and shows a degradation in performance. We can clearly observe this with PG/64, whose average latency is 2.5% higher than PG/8. This study also vindicates our decision of choosing PG/4 for our baseline as it is close to the best performing PG.

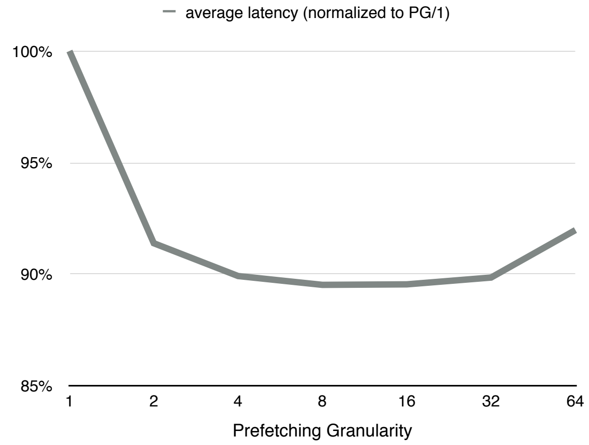

Figure 13: Sensitivity study to PG (with MAP-I).

5.6 Multiprogrammed workloads

IPC: In this section, we study how our design performs under multiprogrammed workloads. We randomly generate 25 workload groups as shown in Table 6, with each group consisting of 4 workloads. The speedup results compared to the baseline system are shown in Figure 14. Each workload group's result is a geometric mean of 4 workloads' speedup. The average is geometric mean of 4 x 25 workloads. As we can see[1], MAP-I provides 5.4% speedup and a full tags-in-SRAM design gives 11.8% speedup over the baseline. On the other hand, ATCache+MAP-I provides up to 12.2% speedup and on average 10.7% speedup over the baseline. Since this is comparable to the speedups we obtained with our uniprocessor workloads, this shows that the ATCache continues to work well under multiprogrammed workloads.

L2 miss latency: To sidestep the effect of different speedup metrics that can be potentially used to summarize the performance of multiprogrammed workloads [12, 4], we compare the L2 miss latency in Fig. 15, as it is arguably a more transparent indicator of the efficacy of ATCache. As we can see from Fig. 15, ATCache with MAP-I predictor shows a 18.4% speedup in L2 miss latency over the baseline; in comparison, tags-in-SRAM shows 21.9% speedup. It is worth noting that a tags-in-SRAM design consumes about 256 times of SRAM space than our design. Again, from these results, we can conclude that our design continues to work effectively in a more memory-intensive scenario.

6. RELATED WORK

The idea of caching tags has been explored previously. Wang et al. [16] proposed the CAT cache for reducing the area overhead of storing tags in the SRAM cache. CAT cache exploits tag value locality by removing tag value duplication – in doing so, they are able to reduce the area overhead of the tag array with a small performance overhead. However, this work is not applicable in the DRAM cache context since CAT involves accessing the data array first before accessing the tag; this is precisely what works on DRAM cache seek to avoid. Furthermore, whereas the CAT cache exploits tag value locality, the ATCache exploits spatial and temporal locality. It would be interesting to explore if ATCache can be extended to exploit value locality of tags also.

Other works [2, 17] have proposed caching tags to improve cache access latency in the generic setting of a multilevel SRAM cache hierarchy. The key difference between this and our work is the new context (DRAM cache) in which target this idea. By targeting the idea to a novel and concrete context of DRAM caches, we are able to specialize our technique in ways over and beyond the prior works. For example, hit prefetching, through which we gain a significant chunk of our performance, has not been discussed in any of the prior works.

7. CONCLUSION

The advent of the DRAM cache has posed a problem of how to efficiently manage the tags associated with the DRAM cache. One naive option is to store all the tags in SRAM; while this would ensure fast access of the tags, the associated storage cost would render this approach impractical. Consequently prior works have proposed innovative techniques to manage the tags efficiently in DRAM. Nonetheless, we observe, with a help of a study, that it is more performance efficient to manage the tags in SRAM.

Having established this, we propose a simple idea to cache the tags in SRAM so that we can achieve the effect of maintaining all tags in SRAM, without paying the prohibitive cost; we show that there is enough spatial and temporal locality amongst DRAM cache tag accesses to merit caching the tags. Our experimental results show that we achieve similar performance (within 2%) to a very fast tags-in-SRAM design (6 cycles access latency for 11.5MB), while consuming less than 1% of the SRAM space. If we integrate our caching idea with prior proposed miss prediction, we show that we can come within 0.5% of performance achieved with the tags-in-SRAM approach.

8. ACKNOWLEDGMENTS

We thank all the reviewers for their helpful comments. We also thank Bharghava Rajaram for discussions that led to the initial idea. This work is supported by UK India Education and Research Iniative Thematic Partnership Award UKUTP201100256 and the Centre for Numerical Algorithms and Intelligent Software, funded by EPSRC grant EP/G036136/1 and the Scottish Funding Council to the University of Edinburgh.

[1] It is worth noting that a system without DRAM cache is 22.0% slower than our baseline. This is because multiprogrammed workloads have significantly more L2 misses.

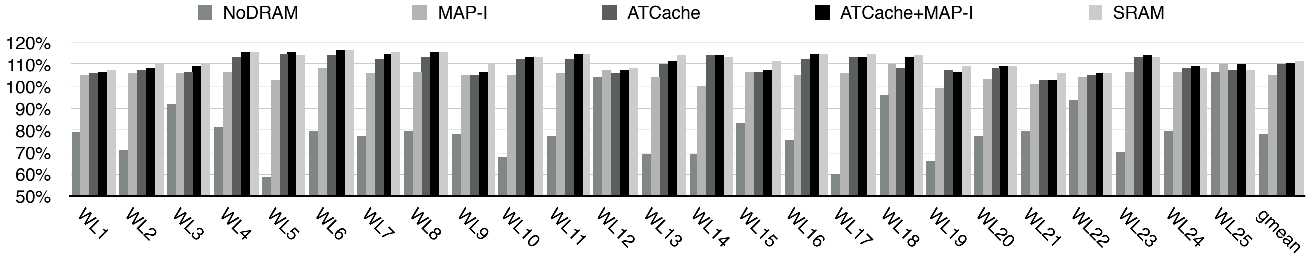

Figure 14: Performance speedup in multiprogrammed workloads.

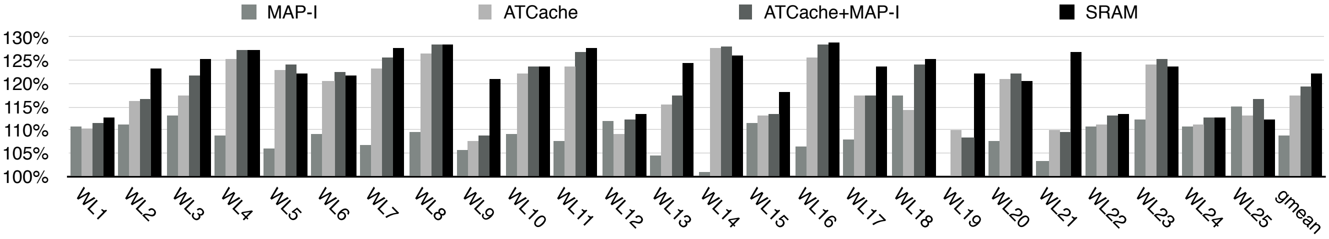

Figure 15: L2 miss latency reduction in multiprogrammed workloads (higher the better).

9. REFERENCES

[1] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[2] W. Chou, Y. Nain, H. Wei, and C. Ma, "Caching tag for a large scale cache computer memory system," Sep. 22 1998, uS Patent 5,813,031. [Online]. Available: http://www.google.co.uk/patents/US5813031

[3] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *SC*, 2010, pp. 1–11.

[4] S. Eyerman and L. Eeckhout, "Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance," *IEEE Computer Architecture Letters*, 2013.

[5] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ASPLOS*, 2012, pp. 37–48.

[6] Intel. Haswell processors. [Online]. Available: http://ark.intel.com/products/codename/42174/haswell

[7] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ISCA*, 2010, pp. 60–71.

[8] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA*, 2013, pp. 404–415.

[9] 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*, 2010, pp. 1–12.

[10] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *MICRO*, 2011, pp. 454–464.

[11] ——, "Supporting very large dram caches with compound-access scheduling and missmap," *IEEE Micro*, vol. 32, no. 3, pp. 70–78, 2012.

[12] P. Michaud, "Demystifying multicore throughput metrics," *IEEE Computer Architecture Letters*, 2013.

[13] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Architecting efficient interconnects for large caches with cacti 6.0," *IEEE Micro*, vol. 28, no. 1, pp. 69–79, 2008.

[14] 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 *MICRO*, 2012, pp. 235–246.

[15] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *MICRO*, 2012, pp. 247–257.

[16] H. Wang, T. Sun, and Q. Yang, "Cat - caching address tags: A technique for reducing area cost of on-chip caches," in *ISCA*, 1995, pp. 381–390.

[17] T. Wicki, M. Kasinathan, and R. Hetherington, "Cache tag caching," Apr. 3 2001, uS Patent 6,212,602. [Online]. Available: http://www.google.co.uk/patents/US6212602

[18] L. Zhao, R. R. Iyer, R. Illikkal, and D. Newell, "Exploring dram cache architectures for cmp server platforms," in *ICCD*, 2007, pp. 55–62.