FusionCache: using LLC Tags for DRAM Cache

Evangelos Vasilakis*, Vassilis Papaefstathiou†, Pedro Trancoso*, Ioannis Sourdis*

*Chalmers University of Technology, CSE Dept, Gothenburg, Sweden
†Foundation for Research and Technology – Hellas (FORTH), Heraklion, Crete, Greece
Email: {evavas,sourdis,ppedro}@chalmers.se, papaef@ics.forth.gr

***Abstract*—DRAM caches have been shown to be an effective way to utilize the bandwidth and capacity of 3D stacked DRAM. Although they can capture the spatial and temporal data locality of applications, their access latency is still substantially higher than conventional on-chip SRAM caches. Moreover, their tag access latency and storage overheads are excessive. Storing tags for a large DRAM cache in SRAM is impractical as it would occupy a significant fraction of the processor chip. Storing them in the DRAM itself incurs high access overheads. Attempting to cache the DRAM tags on the processor adds a constant delay to the access time. In this paper, we introduce FusionCache, a DRAM cache that offers more efficient tag accesses by fusing DRAM cache tags with the tags of the on-chip Last Level Cache (LLC). We observe that, in an inclusive cache model where the DRAM cachelines are multiples of on-chip SRAM cachelines, LLC tags could be re-purposed to access a large part of the DRAM cache contents. Then, accessing DRAM cache tags incurs zero additional latency in the common case.**

I. Introduction

Recent emerging of 3D-stacking technology enables the integration of DRAM and processor on the same package. On the one hand, compared to the narrower-channels of traditional off-chip DRAMs, 3D stacked DRAMs offer 2-4 times higher throughput at a reduced energy per bit cost [6]. On the other hand, they do not necessarily improve access latency and have limited capacity. As a consequence, for most systems traditional DRAMs are still the first choice for the design of main memory. Nevertheless, 3D-stacked DRAM can be an attractive option for hosting a cache as 3D-stacked DRAM caches exploit the coarse grain spatial locality of applications reducing the number of accesses to main memory.

There is a fundamental tradeoff in the design of a DRAM cache. Smaller cachelines benefit bringing just the data that is needed, but result in a large number of tags, which increase the management and storage overheads.

DRAM cache tags can be stored either in on-chip SRAM or in the DRAM itself. Although using on-chip SRAM allows for faster tag access, it is a more expensive choice. Using DRAM to store the tags is more space efficient thus allowing for smaller cachelines. Nevertheless, it results in slower access time and higher bus contention. In order to alleviate the above cost and avoid the extra DRAM access for retrieving the tags, researchers suggested the use of an on-chip SRAM cache of the DRAM cache tags [4]. Another attempt to hide the tag accesses suggested placing the DRAM cache address directly in the TLB [5], but this requires fixing the DRAM cacheline size to the OS page size.

FusionCache addresses the problem of DRAM cache tags management aiming at zero access latency in the common case, while offering variable (at boot-time) cacheline granularity decoupled from the OS page-size. Our design considers DRAM caches with a cacheline size (henceforth denoted as DC-cacheline) a power-of-two multiple of the cachelines used by the on-chip SRAM Last Level Cache (henceforth denoted as LLC and its cachelines as LLC-cachelines). In this case, considering an inclusive cache model, each LLC-cacheline stored in the LLC is always part of a DC-cacheline stored in the DRAM cache. At a LLC miss, a LLC-cacheline will be requested from the DRAM cache. As expected when the LLC cache miss is finally served, the location (DRAM cache set and way) of the corresponding DC-cacheline is known to the DRAM cache controller. This information could be forwarded to the LLC and stored along with the LLC tag of the respective LLC-cacheline. Thereby, after a LLC miss, DRAM cache tag lookups could be potentially avoided if the LLC stores at least one other LLC-cacheline of the requested DC-cacheline. In effect, a tag lookup in the DRAM cache is necessary only once per DC-cacheline, when its first LLC-cacheline is requested and generates a LLC miss. In the past, similar approaches have been considered to save tag space, but none of them targeted DRAM-caches. Seznec reused tags in on-chip SRAM caches [11] and Sardashti and Wood in compressed caches where more tag space is needed for compressed cachelines [10].

The rest of the paper is structured as follows: Section II describes our design followed by our evaluation results in Section III and conclusions in Section IV.

II. FusionCache Design

FusionCache is a new DRAM cache (DC) design that reuses the on-chip LLC tags and augments them with fields to keep on-chip a subset of the DC tags. The full DC tags are still stored in the off-chip DRAM.

The proposed FusionCache design assumes inclusive caches and that DC-cachelines use larger block sizes than the LLC-cachelines (power-of-two multiples). The intuition behind *fusing* the DC tags with the LLC tags is that if at least one of the LLC-cachelines that belong to the same DC-cacheline is present in the LLC, then we have direct information about the *location* (way) of the associated DC-cacheline inside the DRAM Cache without additional DC tag lookups. The FusionCache design for an 8MB 16-way LLC requires only 200KB of additional tag space, which is only 2% additional storage overhead for the LLC.

FusionCache has to address the following challenge. All LLC-cachelines that belong to the same DC-cacheline must be

978-3-9819263-0-9/DATE18/©2018 EDAA

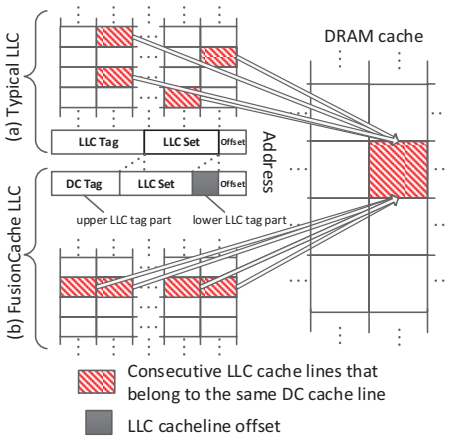


Fig. 1: LLC-cacheline placement of (a) typical LLC, and (b) FusionCache.

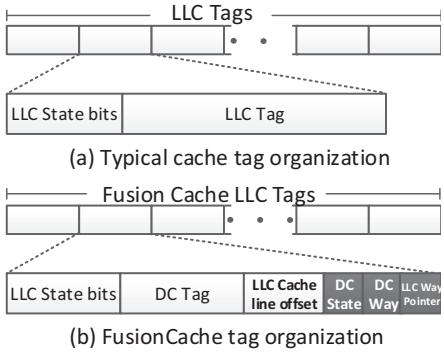


Fig. 2: Tag organization for (a) typical LLC cache, and (b) FusionCache. The darker marked bits are the extra bits needed for FusionCache support.

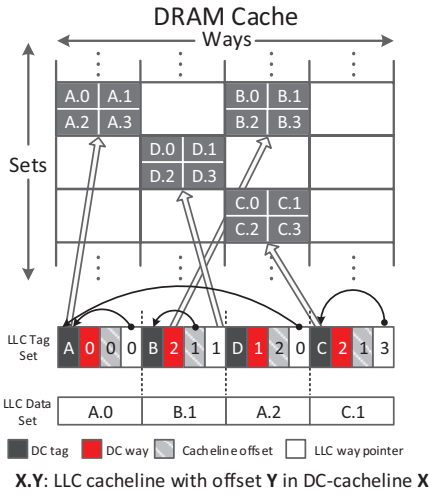


Fig. 3: A FusionCache example.

placed in the same LLC set to ensure that a single LLC access gives a definite answer regarding the DRAM cache tags. The typical cache organization, shown in Figure 1a, would require the FusionCache to access all LLC sets that potentially store LLC-cachelines of the same DC-cacheline to know whether the LLC has (or not) information for the associated DC-cacheline. On the contrary, if LLC is indexed considering the DC-cacheline size, then all above LLC-cachelines reside in the same set as depicted in Figure 1b. More precisely, the LLC set bits are extracted from the requested address after skipping the first n bits after the byte offset, where 2^n is the number of LLC-cachelines that fit in a DC-cacheline. The proposed address breakdown for accessing the FusionCache is shown in Figure 1b. The new LLC tag is then formed by concatenating the first n bits after the byte offset (lower tag part) and the remaining most significant bits after the LLC set bits (upper tag part). In essence, the upper LLC tag part is the *DC-cacheline tag* and the lower LLC tag part is its *LLC-cacheline offset*.

Now that consecutive LLC-cachelines which belong to the same DC-cacheline map to the same set, we observe that their upper tag parts (which is the DC tag) are common. This gives us the opportunity to store only one upper tag part for all of them, as long as we add a few extra bits to each one of them to point to the way their upper tag part is located. We call this pointer *LLC-way-pointer*. In doing so, we can save on LLC tag space and increase the number of DC tags stored in the LLC. Figure 2 shows the organization of LLC tags in the FusionCache design in comparison with the one of a typical LLC. The *LLC state bits* remain the same. The LLC tag is split in two parts: the upper part is the *DC tag* and the n least significant bits are the *LLC cacheline offset*. The only additional bits required for the FusionCache support are the *DC state bits*, the *DC way*, which stores the respective DC-cacheline, as well as a pointer to the LLC way which stores the correct DC tag (*LLC-way-pointer*).

Figure 3 shows an example of the FusionCache design. At the bottom of the figure, an LLC data set and its tag array are depicted. LLC-cachelines $A.0$, $B.1$, $A.2$, and $C.1$ are stored in this LLC set, which belong to the DC-cachelines A, B, A, and C, respectively. A number of fields in the tag array correspond to each of these LLC-cachelines as described in Figure 2.b. For clarity, in this example we omit the LLC and DC state bits. The remaining bits in the LLC tag set are for each LLC-cacheline the following: the DC tag, the DC way, the LLC-cacheline offset, and the LLC-way-pointer. $A.0$ cacheline points to the DC-cacheline A and indicates that A is in way 0 of the DC. It further indicates that it is the first LLC-cacheline in A because the LLC-cacheline offset is zero. Finally, we know that the DC tag belongs to the LLC-cacheline stored in this particular LLC way (way 0) because the LLC-way-pointer is zero.

This decoupling of DC tags saves space in the LLC tag array for more DC tags. Also, the number of sets and ways does not have to be the same for the LLC and the DC.

Details about FusionCache functionality are presented next:

LLC Cache indexing and tag matching: Cache lookups are comprised of 2 steps: *Indexing* and *Tag matching*. In FusionCache, LLC is indexed with regards to the DC-cacheline size, and tag matching is performed using both the DC tag and the LLC cacheline offset. For locating the correct DC tag of an LLC-cacheline the LLC-way-pointer is used.

DRAM cache tag matching: In case the access is a miss in the requested LLC-cacheline but a hit in the DC tag, then the DC data can be accessed without further tag matching. Otherwise, the DRAM cache tags need to be read from DRAM and checked first.

Cacheline replacement: Replacement both in the LLC and the DRAM cache is performed using Least Recently Used (LRU) policy, however, when adding a new LLC cacheline in the LLC there are two different cases. Before adding an LLC-cacheline it is checked whether its DC tag is already stored in the LLC set due to another LLC-cacheline currently or previously stored in the set. If so, then the allocation is simple as there needs to be no change in the DC tags that reside in the LLC set; the new LLC-cacheline will point to the already stored DC tag and DC way. Otherwise, if the corresponding DC tag is not in the set, another DC tag should be evicted and written back to the DRAM cache tag array in DRAM.

TABLE I: System configuration.

Cores	Out-of-order 4-way issue/commit 3.2 GHz
L1 Cache	Private, 64 KB, 4-way, 1 cycle access latency
L2 Cache	Private, 256 KB, 8-way, 9 cycles access latency
L3 Cache	Shared 8MB, 16-way, 14 cycles access latency
DRAM Cache	512 MB, 2 channels, 128 bit bus, 60 cycles access latency, 1.6 GHz DDR
Tag cache	256 KB, 1 cycle access latency, 4 cycles read latency
Main DRAM	8 GB, 2 channels, 64 bit bus, 92 cycles access latency, 800 MHz DDR

Cache Eviction: For any LLC-cacheline in a set we already have information about the location of the corresponding cacheline in the DRAM cache. So, an eviction and possible write-back of a LLC-cacheline is always done directly to the DRAM cache without the need for a lookup.

Configurable DC-cacheline size: The DC-cacheline size is configurable at boot time to accommodate the size preferred by the workloads at hand. To achieve this configurability, our design considers the finest possible DC-cacheline granularity (128B) for the DC-Tag, and the coarser (4KB) for the LLC-cacheline offset.

In summary, FusionCache indexes LLC considering the DC-cacheline size rather than the LLC-cacheline one. Then, all LLC-cachelines of the same DC-cacheline are constrained to reside on the same LLC set. This enforces that a single LLC access can determine whether LLC can index the DRAM cache. It also allows sharing the upper part of the LLC tags effectively increasing the number of DC tags stored in LLC.

III. Evaluation

We present our evaluation results for four different design points: (i) **Baseline (B):** No DRAM cache; (ii) **DRAM Cache (DC):** DRAM cache with tags-in-DRAM; (iii) **DRAM Cache with Tag-Cache (DCTC):** DRAM cache with tags in DRAM and an on-chip SRAM cache of the DC-tags similar to [4], the DCTC SRAM tag cache size is equal to the SRAM overhead incurred by FusionCache; and (iv) **FusionCache (FC)**.

Other techniques such as the Tagless Cache [5] are not included in our comparison as they are restricted to a single DC-cacheline size (4KB for agless Cache). As shown below, different applications achieve their best performance for different DC-cacheline sizes.

For the evaluation an in-house simulator based on *PIN* [8] is used following the interval-based simulation methodology [2] for the processor and cycle-accurate modelling of the cache and the off-chip memory system. A four core processor is simulated with private L1 and L2 caches and a shared last level cache (LLC). Table I presents the configuration of our system. *Cacti* is employed to determine the access times for the caches and tag arrays [13]. For main memory and DRAM timings the delays provided by [5] are considered.

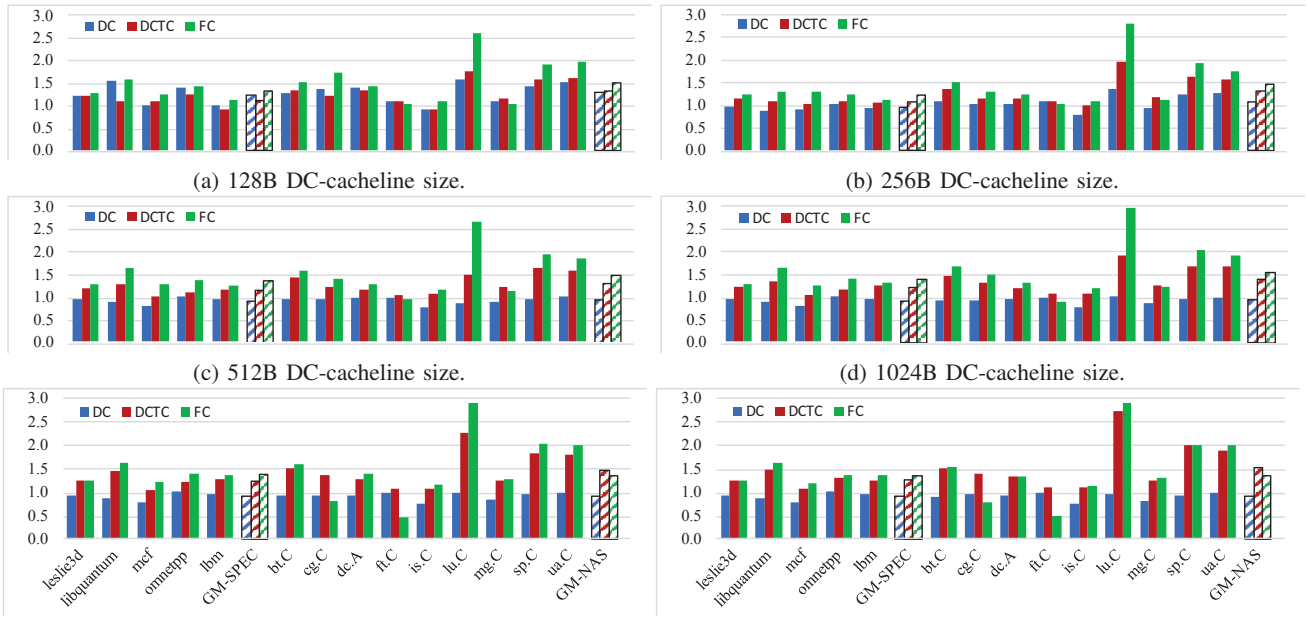
We evaluate our design with both single- and multi-threaded workloads. For single-threaded benchmarks, we selected, based on Phansalkar et al. [9], a representative subset of the SPEC2006 [3]. For multi-threaded benchmarks we used the *OpenMP* version of the *NAS Parallel Benchmark* suite [1]. One billion instructions are simulated for each thread after a cache warmup of 100 million instructions. For NAS we select the simulated period to be right after the initialization phase of each benchmark, while for the SPEC benchmarks we use simpoints to select a representative slice [12].

Figure 4 shows the speedup for each design compared to the Baseline (without DRAM cache). For this part of our evaluation we consider DC-cacheline sizes ranging from 128B to 4KB; smaller sizes are not supported by the FusionCache design as it requires the DC-cacheline to be at least 2 times the size of the LLC-cacheline, which is 64B. The dashed bars represent the geometric mean results across for all SPEC and NAS benchmarks (*GM-SPEC* and *GM-NAS*, respectively). Each different figure represents a scenario with different DC-cacheline (128B to 4KB).

For the single-threaded SPEC benchmarks, FC performs better or as good as the other designs across all DRAM cache-line sizes. For the multi-threaded benchmarks, FC achieves again better or equal performance than the rest of the designs except for the *cg.C* and *ft.C* at larger DRAM cache block sizes. The underperformance of FC in these benchmarks has an impact on its overall performance for DC-cacheline sizes 2KB and 4KB where the FC mean results (GM-NAS) are actually lower than DCTC. This is due to the constraints posed by the FusionCache design to LLC using higher order address bits for selecting the set. Depending on the memory access pattern, LLC in FusionCache may have more set-conflicts and thus more LLC evictions reducing its efficiency. This effect is more evident in the large DC-cacheline sizes (e.g. 2KB and 4KB) for workloads with high spatial locality. In such cases, an entire LLC set may be dedicated to a single DC-cacheline reducing the effective LLC associativity.

DC has better performance over DCTC in smaller DC-cacheline sizes because it exploits compound accesses better than the DCTC design where a tag-cache miss will stall future accesses to the DC set until it is served. We support compound accesses for DC-cacheline sizes only up to 128B as it would be too wasteful in DRAM space to place tags and data in the same DRAM row for bigger DC-cachelines [7]. The overall better performance of FC over DCTC can be attributed to the higher number of DRAM cache accesses without a tag lookup (87% for FC vs 67% for DCTC on average). This also affects the total DRAM cache traffic which is 8% lower on average across all our benchmarks.

We summarize the above results by showing the benefit of the FC design when compared to the DC and DCTC designs, independent of the DC-cacheline size. We select the best speedup for each design in each benchmark at any DC-cacheline size. For DC and DCTC we also take into account their performance for 64B sizes, which was not included in the previous comparison as it is not supported by FC. In addition, we present the average benefit across SPEC and NAS benchmarks. These results are depicted in Figure 5a and 5b, respectively. They show that the benefit of FC over DC is 14% for the single- and 29% for the multi-threaded benchmarks. When comparing to DCTC, the benefits of FC are 9% for the single- and 7% for the multi-threaded benchmarks.



(a) 128B DC-cacheline size.

(b) 256B DC-cacheline size.

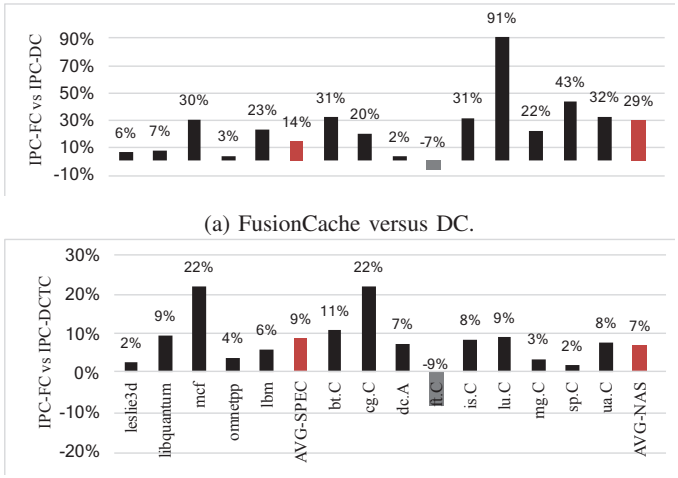
(c) 512B DC-cacheline size.

(d) 1024B DC-cacheline size.

(e) 2KB DC-cacheline size.

(f) 4KB DC-cacheline size.

Fig. 4: Speedup vs. the baseline for DC, DC with Tag-Cache (DCTC), and FusionCache (FC).



(a) FusionCache versus DC.

(b) FusionCache versus DCTC.

Fig. 5: Performance benefit of FusionCache.

IV. Conclusions

We presented *FusionCache*, a novel approach for accessing a DRAM cache (DC) by reusing the LLC tags. FusionCache exploits the fact that a LLC-cacheline that resides in LLC is always part of a DC-cacheline stored in the DRAM cache. Our design re-purposes the LLC tag array to additionally store information for accessing such DC-cachelines. In practice, it reuses the LLC tag space as a cache for the DRAM cache tags without the need of a separate on-chip SRAM tag-cache. For NAS benchmarks, FusionCache achieves an average speedup of 7% against a DRAM cache with SRAM tag-cache and 29% against a DRAM cache with tags in DRAM. For SPEC benchmarks, these speedups are 9% and 14%, respectively.

Acknowledgements

This work was partially supported by the European Commission under the Horizon 2020 Program through the ECOSCALE project (grant agreement 671632).

References

[1] David H Bailey et al. “The NAS parallel benchmarks”. In: *Int. J. of Supercomp. Appl.* 5.3 (1991), pp. 63–73.

[2] D. Genbrugge, S. Eyerman, and L. Eeckhout. “Interval simulation: Raising the level of abstraction in architectural simulation”. In: *Int. Conf. HPCA*. 2010, pp. 1–12.

[3] John L. Henning. “SPEC CPU2006 Benchmark Descriptions”. In: *SIGARCH Comput. Archit. News* 34.4 (2006).

[4] Cheng-Chieh Huang and Vijay Nagarajan. “ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache”. In: *PACT*. 2014, pp. 51–60.

[5] Y. Lee et al. “A fully associative, tagless DRAM cache”. In: *ISCA*. 2015, pp. 211–222.

[6] G. H. Loh. “3D-Stacked Memory Architectures for Multi-core Processors”. In: *ISCA*. 2008, pp. 453–464.

[7] Gabriel H. Loh and Mark D. Hill. “Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches”. In: *MICRO-44*. 2011, pp. 454–464.

[8] Chi-Keung Luk et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: *ACM SIGPLAN PLDI*. 2005, pp. 190–200.

[9] A. Phansalkar, A. Joshi, and L.K. John. “Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite”. In: *SIGARCH Comput. Archit. News* 35.2 (2007), pp. 412–423.

[10] S. Sardashti and D. A. Wood. “Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching”. In: *MICRO*. 2013.

[11] A. Seznec. “Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio”. In: *ISCA*. 1994.

[12] T. Sherwood et al. “Automatically Characterizing Large Scale Program Behavior”. In: *SIGOPS Oper. Syst. Rev.* 36.5 (2002).

[13] S.J.E. Wilton and N.P. Jouppi. “CACTI: an enhanced cache access and cycle time model”. In: *IEEE Journal of Solid-State Circuits* 31.5 (May 1996), pp. 677–688.