A Fully Associative, Tagless DRAM Cache

Yongjun Lee[†] Jongwon Kim[†] Hakbeom Jang[†] Hyunggyun Yang[‡]
Jangwoo Kim[‡] Jinkyu Jeong[†] Jae W. Lee[†]
[†]Sungkyunkwan University, Suwon, Korea [‡]POSTECH, Pohang, Korea
{yongjunlee, kimjongwon, hakbeom, jinkyu, jaewlee}@skku.edu {psyjs037, jangwoo}@postech.ac.kr

Abstract

This paper introduces a tagless cache architecture for large in-package DRAM caches. The conventional die-stacked DRAM cache has both a TLB and a cache tag array, which are responsible for virtual-to-physical and physical-to-cache address translation, respectively. We propose to align the granularity of caching with OS page size and take a unified approach to address translation and cache tag management. To this end, we introduce cache-map TLB (cTLB), which stores virtual-to-cache, instead of virtual-to-physical, address mappings. At a TLB miss, the TLB miss handler allocates the requested block into the cache if it is not cached yet, and updates both the page table and cTLB with the virtual-to-cache address mapping. Assuming the availability of large in-package DRAM caches, this ensures that an access to the memory region within the TLB reach always hits in the cache with low hit latency since a TLB access immediately returns the exact location of the requested block in the cache, hence saving a tag-checking operation. The remaining cache space is used as victim cache for memory pages that are recently evicted from cTLB. By completely eliminating data structures for cache tag management, from either on-die SRAM or in-package DRAM, the proposed DRAM cache achieves best scalability and hit latency, while maintaining high hit rate of a fully associative cache. Our evaluation with 3D Through-Silicon Via (TSV)-based in-package DRAM demonstrates that the proposed cache improves the IPC and energy efficiency by 30.9% and 39.5%, respectively, compared to the baseline with no DRAM cache. These numbers translate to 4.3% and 23.8% improvements over an impractical SRAM-tag cache requiring megabytes of on-die SRAM storage, due to low hit latency and zero energy waste for cache tags.

ISCA'15, June 13–17, 2015, Portland, OR, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3402-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750383

1. Introduction

Recently, die-stacked DRAM technologies have been widely adopted to satisfy conflicting demands for capacity, throughput, and energy efficiency from the main memory system. Major processor vendors have announced their plans to integrate the 2.5D/3D die stacking technologies into their products, including Intel's Knights Landing [3], Xilinx' Virtex-7 FPGA [8], Nvidia's Volta GPU [6], to name a few. For example, Intel's 72-core Knights Landing processor will include up to 16GB in-package DRAM, backed by up to 384GB off-package DDR4 DRAM. AMD [1], IBM [19], and Sony [4] also plan to adopt die-stacked High Bandwidth Memory (HBM) [24] for their future processor SoCs.

To best utilize fast, energy-efficient in-package DRAM without burdening software writers, many researchers propose to use it as a large last-level cache [20, 21, 22, 26, 27, 29, 38]. This is justified by the fact that the in-package DRAM capacity, ranging from hundreds of megabytes to several gigabytes [27], is still not big enough to completely replace the main memory especially for emerging applications with huge memory footprints [16] such as in-memory database [7] and genome assemblies [31]. Alternatively, exposing the in-package DRAM to software by mapping it to the physical memory space would burden OS and/or application writers with the difficult task of memory page placement and migration [27].

The primary challenge in architecting large, in-package DRAM caches is to minimize the overhead of cache tag management in terms of latency, storage, and energy consumption. For example, a 1GB DRAM cache with the conventional cache block size, say 64 bytes, requires 128MB storage for tags only (assuming 48-bit physical address space). It is infeasible to have such a large SRAM array on the processor die. Instead, one might place the tags, together with cached data, into in-package DRAM, but only at the cost of increased hit latency, and hence significant performance overhead [27, 29].

To alleviate the problems associated with large tags, *page-based* DRAM caches have recently been proposed to cache at a page granularity, typically ranging from 1 to 8 kilobytes [20, 21, 22]. In addition to reducing the tag overhead, page-based caches have additional benefits of higher hit rate by better exploiting spatial locality and maximum DRAM access efficiency by amortizing row activation cost. However, the tag overhead is still significant. For 1GB in-package DRAM, most of the existing page-based DRAM caches either require multi-megabytes of on-die SRAM (i.e., 2MB for [21, 22]) or allocate 32–64MB in-package DRAM [20] just for tags. This leads to significant overhead in terms of latency, chip area, and energy consumption, which is likely to increase as the in-package DRAM size continues to scale up.

This paper introduces a *tagless* page-based cache, which completely eliminates cache tags. The key idea is to align the granularity of caching with OS page size, and take a unified approach to address translation and cache tag management. Unlike the conventional page-based cache, whose access path requires two-step address translation with both TLB (for virtual-to-physical) and cache tags (for physical-to-cache), the proposed DRAM cache consolidates them into a single-step procedure using *cache-map TLB* (cTLB). cTLB replaces the conventional TLB and stores virtual-to-cache, instead of virtual-to-physical, address mappings. At a TLB miss, the TLB miss handler performs the conventional page table walk, and allocates the requested memory page into the cache if it is not cached yet. Then it updates both the page table and the cTLB with the corresponding virtual-to-cache address mapping. Assuming the availability of large in-package DRAM caches, an access to the memory region within the TLB reach is guaranteed to hit in the cache with low hit latency due to no cache tag-checking overhead. The remaining cache space outside the TLB reach is used as victim cache for memory pages recently evicted from cTLB. Eviction from the victim cache is performed asynchronously to take write-back overhead off from the cache access path by having a small number of free blocks always available. Note that techniques to alleviate the over-fetching problem for page-based caches, such as footprint caching [21], hot/cold page tracking [20, 22], are complementary and can augment our work.

In summary, the proposed tagless DRAM cache satisfies the following, often conflicting, design goals:

- *Low hit latency* - Cache tag-checking operation is completely eliminated from the cache access path to achieve lowest hit latency known to date.
- *Zero tag storage overhead* - cTLB directly translates a virtual address to a cache address to eliminate tag storage in either on-die SRAM (for page-based caches) or in-package DRAM (mostly for block-based caches). In addition, the proposed cache does not require any auxiliary on-die SRAM structure such as MissMap [27], miss predictors [29, 33], way predictor [20], etc.
- *Low average memory access time* - The proposed cache effectively implements a fully associative cache to yield higher hit rate than existing page-based caches based on direct-mapped [29] or N-way set-associative schemes with low N [20]. Combined with low hit latency, the proposed cache has even lower average memory access time than an impractical 16-way set-associative SRAM-tag cache, which requires megabytes of on-die SRAM storage.
- *High energy efficiency* - No energy waste from tags and low average memory access time lead to high energy efficiency.
- *Scalability* - The only new data structure introduced by the proposed cache is the global inverted page table (GIPT), which maintains cache-to-physical address mappings. This table is shared by all processes in the system and has a very small overhead (2.56MB for 1GB cache, yielding <0.25% overhead). More importantly, unlike cache tags, this table is accessed infrequently (only at a TLB miss and cache block eviction) along with the page table, and is not required to be placed in package. Hence, the proposed cache features superior scalability to the existing tag-based caches.
- *Flexibility* - Most of the proposed caching mechanism is embodied in the TLB miss handler with small modifications to the page table. A caching policy (e.g., selective locking or bypassing of cache blocks) can be flexibly plugged in by modifying the TLB miss handler. This is particularly relevant to architectures with software-managed TLBs (e.g., MIPS, Alpha, UltraSPARC).

2. Background and Motivation

2.1. Die-Stacked DRAM

Die-stacking technologies have recently drawn much attention from the research community as a viable solution to the "Memory Wall" problem [37]. A typical 3D form factor stacks 4–8 DRAM dies using through-silicon vias (TSVs) with an optional logic die at the bottom [28]. Multiple industry standards such as High Bandwidth Memory [13] and Hybrid Memory Cube [28] have emerged to support this technology with die-stacked DRAM capacity ranging from 512MB to 4GB. A silicon interposer with die-stacked DRAM integration, also known as 2.5D form factor, further improves the scalability of in-package DRAM by integrating multiple, possibly 3D stacked, dies using intra-package communication mediums on silicon interposer [34]. Processors with silicon interposer-based integration is likely to hit the market before full 3D stacked processors for their advantages in yield (and hence cost) [8, 11].

There are two main ways to exploit fast, in-package DRAM: software-managed fast main memory and software-transparent last-level cache. The former typically maps the in-package DRAM region to physical address space, to create a *heterogeneous* main memory composed of small, fast in-package region and slow, big off-package region [12, 30]. The software-managed heterogeneous memory approach has zero tag overhead associated with cache management and maximizes useable capacity with no duplication of a memory page. However, OS (and possibly applications as well) should handle the difficult task of page placement and migration. This is particularly challenging on commodity platforms since OS cannot gather per-page access statistics without hardware support [27]. Therefore, most of prior work along this avenue proposes to augment the memory controller with counters to gather access statistics [12, 30].

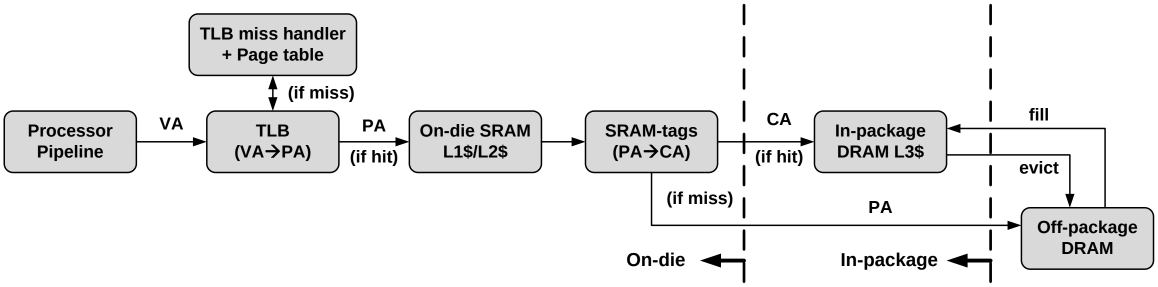


Figure 1: Access path of the baseline page-based cache with on-die SRAM tags

As an alternative, there are recent proposals to architect in-package DRAM as a large, software-transparent last-level cache [20, 21, 22, 26, 27, 29, 38]. This approach has an advantage of being readily deployable without modifying the software stack. With large, and increasing, size of in-package DRAM, minimizing the overhead associated with tag management such as access latency, storage, and energy consumption, is the primary design challenge when architecting an in-package DRAM cache.

Existing in-package DRAM caches can be divided into two classes according to the granularity of caching: (cache) block-based [17, 27, 33] and page-based [20, 21, 22]. A block-based cache stores data and tags at a conventional cache line granularity, say 64 bytes, to maximize effective capacity of caching (by minimizing over-fetching) and efficiently exploit temporal locality. However, the storage requirement for tags is very large, accounting for 12.5% of the total in-package DRAM capacity (e.g., 128MB per 1GB DRAM cache) in a state-of-the-art design [29]. Besides, block-based caches do not effectively exploit DRAM row buffer locality, and have low hit rate for server workloads which have low temporal locality and high spatial locality [21].

Overcoming these limitations, page-based caches have been recently proposed to alleviate the problem of large tag overhead by caching at a page granularity, whose size ranges from 1-8 kilobytes [20, 21, 22]. This proportionally reduces the tag overhead, hence improving latency and energy efficiency. Most of the existing proposals for page-based caching tackle the over-fetching problem, to reduce off-package bandwidth pollution and increase effective capacity. However, reducing the tag overhead is still an important design goal with ever increasing in-package DRAM size as the state-of-the-art page-based caches require either multi-megabyte on-die SRAM storage or a significant fraction of the total in-package DRAM capacity (e.g., 3–6% in [20]).

2.2. Cache Tag Overhead in Page-Based Caches

Figure 1 illustrates the access path of a page-based DRAM cache with on-die SRAM tags, which serves as the common baseline for two state-of-the-art page-based caches before being augmented with footprint caching [21], and hot page filtering [22]. To locate the requested cache block, the TLB first translates its virtual address to the physical address, which in turn gets translated to the cache address using the cache tags. Therefore, the average memory access time ($AMAT_{SRAM-tag}$) with this translation cost taken into account can be expressed as follows:

$$\begin{aligned} AMAT_{\text{SRAM-tag}} &= MissRate_{\text{TLB}} * MissPenalty_{\text{TLB}} \\ &\quad + AMAT_{\text{TLB-hit}} \end{aligned} \tag{1}$$

where

$$\begin{aligned} AMAT_{\text{TLB-hit}} &= HitTime_{\text{L1/L2}} \\ &\quad + MissRate_{\text{L1/L2}} * AvgL3Latency \end{aligned} \tag{2}$$

and

$$\begin{aligned} AvgL3Latency &= AccessTime_{\text{SRAM-tag}} \\ &\quad + BlockAccessTime_{\text{in-pkg}} \\ &\quad + MissRate_{\text{L3}} * PageAccessTime_{\text{off-pkg}} \end{aligned} \tag{3}$$

The average memory access time ($AMAT_{SRAM-tag}$) includes the cost for both virtual-to-physical address translation by the TLB (the first term in Equation 1) and physical-to-cache address translation by the cache tags (the first term in Equation 3). The cache tags (labeled "SRAM-tags" in Figure 1) provide a membership check (i.e., hit or miss) for the requested physical address as well as a translation to the cache address in case of a hit. Our modeling with CACTI 6.5 [2] indicates that the SRAM tag access latency takes up a significant fraction of the L3 hit latency (i.e., 11 cycles out of 65 cycles for 3GHz CPU). Note that, the tag access latency is placed on the critical path of an L3 cache access regardless of hit or miss.

In addition, a 1GB in-package DRAM cache with 4KB page requires a 2MB on-die SRAM storage for tags, which incurs significant area and energy overhead. This overhead will be more pronounced as in-package DRAM size continue to scale up in the future as integration of 4GB [28] and 16GB [3] in-package DRAM will soon be introduced into the market.

Therefore, reducing the tag overhead is a primary design goal even for paged-based in-package DRAM caches. In this paper we introduce the *first* tagless cache architecture, which fundamentally resolves the future scaling problem of large in-package DRAM caches.

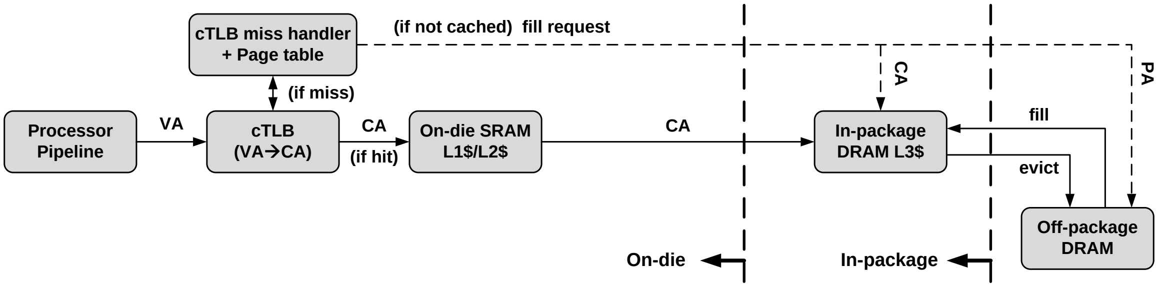


Figure 2: Access path of the proposed tagless cache

3. Tagless DRAM Cache

3.1. Overview

The proposed tagless DRAM cache aligns the granularity of caching to an OS page size, say 4KB, and consolidates the two address translation mechanisms in the conventional DRAM cache into one. The TLB miss handler performs not only page table walk but also cache block allocation/fetch. We introduce *cache-map TLB (cTLB)*, which has the identical hardware organization to the original TLB but stores virtual-to-cache, instead of virtual-to-physical, address mappings. In this way we can completely eliminate on-die SRAM tags in Figure 1 and their latency/storage/energy overheads.

Figure 2 illustrates the access path of the tagless DRAM cache in comparison to the SRAM-tag cache in Figure 1. The operations of the cache access path are sketched as follows:

- Upon a miss, the TLB miss hander first performs the conventional page table walk to find the right page table entry (PTE). If the memory page being accessed currently resides in the cache, the TLB miss handler simply returns to update cTLB with the physical-to-cache address mapping. If not cached, it allocates a free cache block and generates a cache fill request to copy the page from off-package DRAM to the allocated cache block. It also updates the global inverted page table (GIPT), which maintains cache-to-physical mappings for cached pages, with the new entry. Once completed, the TLB miss handler returns to update cTLB with the newly created virtual-to-cache address mapping.
- At a TLB hit cTLB returns the cache address (CA), instead of the physical address (PA). On-die SRAM caches, say L1 and L2 caches in Figure 2, are now addressed and tagged by (in-package) cache addresses instead of (off-package) physical addresses.
- In case of an on-die cache miss, the cache address is used to access the in-package DRAM cache. Note that, with a TLB hit, it is guaranteed for the access to hit in the cache without paying the cost of a tag-checking operation in existing DRAM caches.
- Cache block eviction is done asynchronously using the free queue. To take the write-back overhead off from the cache access path, the tagless cache ensures that a small number of free blocks, say α blocks, are always available to accommodate a cache fill request in a similar way to [12]. There is a globally shared variable, called header pointer, which points to the next free cache block to be allocated. A cache block to be freed is enqueued into the free queue, which is serviced asynchronously, possibly by a background process. If a cache block is evicted, its PTE is updated to replace the cache address with the physical address by consulting the GIPT.

Then the average memory access time ($AMAT_{Tagless}$) can be expressed as follows:

$$\begin{aligned} AMAT_{\text{Tagless}} = {} & MissRate_{\text{cTLB}} * MissPenalty_{\text{cTLB}} \\ & + HitTime_{\text{L1/L2}} \\ & + MissRate_{\text{L1/L2}} * BlockAccessTime_{\text{in-pkg}} \end{aligned} \tag{4}$$

where

$$\begin{aligned} MissPenalty_{\text{cTLB}} = {} & MissPenalty_{\text{TLB}} \\ & + MissRate_{\text{Victim}} * \big[AccessTime_{\text{GIPT}} \\ & + PageAccessTime_{\text{off-pkg}}\big] \end{aligned} \tag{5}$$

According to our evaluation, the average memory access time ($AMAT_{Tagless}$) is consistently lower than that of the SRAM-tag cache ($AMAT_{SRAM-tag}$) in Section 2.2 by eliminating the tag-checking latency. Detailed results will be presented in Section 5.1.

3.2. Overall Structure

Figure 3 depicts the overall structure of the proposed tagless cache and data flows among components. The three major components are cTLB, modified page table, and global inverted page table (GIPT). Hardware modifications may or may not be required depending on how TLB misses and page faults are handled–either in hardware or software. Our technique is applicable to both cases. The rest of this section discusses the organization and operation of each component.

Page Table: The page table is modified to accommodate three additional bits. Note that most of the popular architectures have some unused bits available in PTEs. For example, the x86_64 architecture has 14 unused bits (11 in high bits and three in low bits) [9, 18]. The three flag bits are summarized as follows:

- *Valid-in-Cache (VC)*: This bit indicates whether the page is currently cached in the DRAM cache or not.

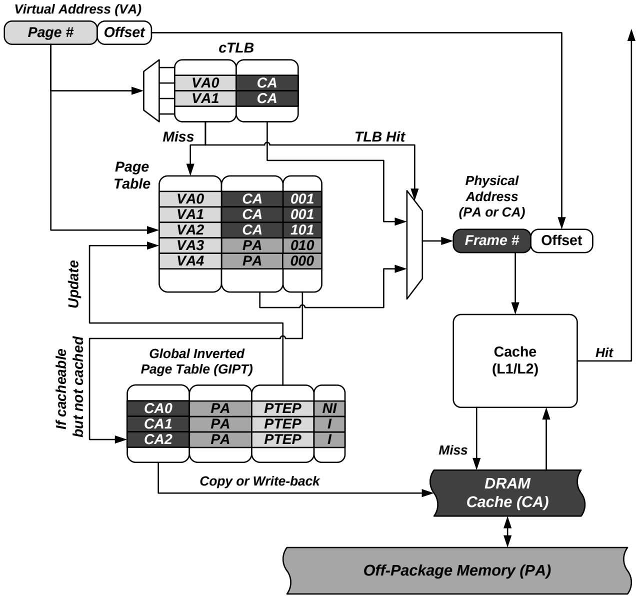


Figure 3: The overall structure of the tagless DRAM cache

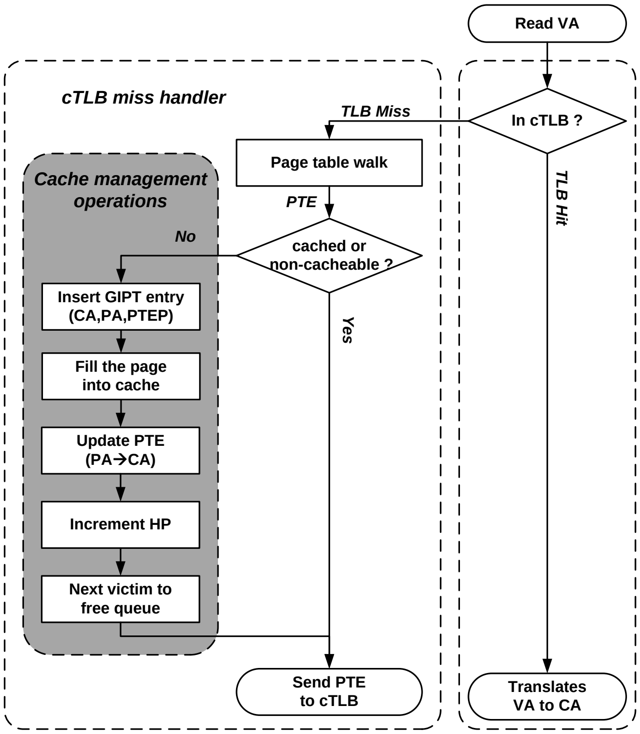


Figure 4: Flow chart of cTLB operations

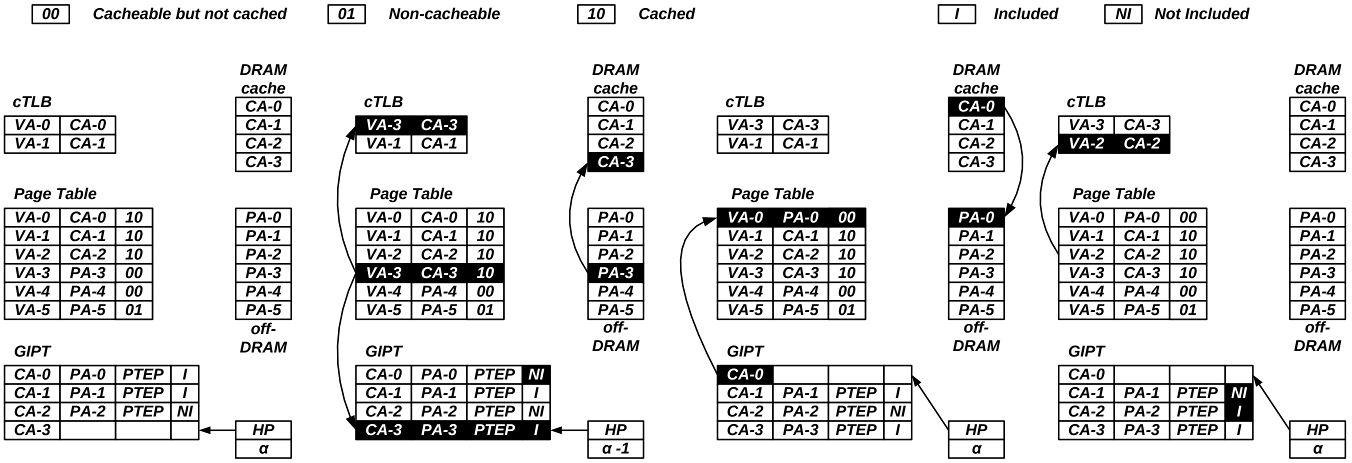
- *Non-Cacheable (NC)*: This bit indicates whether the page bypasses the DRAM cache or not. Note that, even if this bit is set to one, the page does not bypass on-die SRAM caches (e.g., L1 and L2 caches in Figure 2), hence effectively enabling block-level caching. More detailed usage of this bit is discussed in Section 3.5.
- *Pending Update (PU)*: This bit indicates that the PTE will soon be updated (e.g., upon completion of an on-going page copy for a cache fill). This bit is included to eliminate duplicate cache fill requests to the same page if requested by multiple threads.

cTLB: The hardware organization of cTLB is identical to that of the original TLB except for inclusion of a Non-Cacheable (NC) bit from the new PTE. If the NC bit is set to zero, which is the common case, the TLB entry maintains a virtual-to-cache address mapping; otherwise, it maintains a conventional virtual-to-physical address mapping for a non-cacheable page.

Figure 4 shows the flow chart of the TLB miss handler with cTLB. The shaded path illustrates the extra work done by the handler for cache management in addition to the conventional page table walk. Note that this path is activated only if the page is cacheable but not cached yet (i.e., (VC, NC) = (0, 0)).

Once activated, the handler first sets the PU bit in the PTE using an atomic instruction such as compare-and-swap [14]. Note that, as a result of the page table walk, the PTE is already loaded into the processor cache, so it will hit in the cache. If the PU bit is already set to one by another thread, it will busy-wait until the bit is cleared. Then the hander gets allocated a free cache block (pointed to by the *header pointer* (HP)), and inserts an entry to the GIPT to establish a cache-to-physical address mapping. It is followed by a cache fill request using both cache and physical addresses already available. Upon completion the hander updates the PTE to replace the original (off-package) physical address with the (in-package) cache address. Assuming a small number of free blocks are always available and FIFO replacement policy, HP is incremented by one to point to the next free block. The next victim from the cache (practically, its CA) is enqueued into the free queue, which will be cleared asynchronously. Finally, the PU bit will be reset to zero, and the handler will return with the cache address to update cTLB.

Global Inverted Page Table (GIPT) and Free Queue: These two components collaboratively implement a cache replacement mechanism. In the same spirit of placing the replacement path off from the common cache access path as in [12], cache block eviction is done asynchronously. The free queue is a FIFO, which maintains a list of cache blocks to be evicted. The GIPT is a global (not process-private) structure indexed by cache address and has three fields: physical page number (PPN), PTE pointer (PTEP) and TLB residence bit vector. The GIPT maintains a cache-to-physical address mapping as well as a pointer to the PTE for each cached block. In addition, The GIPT maintains TLB flags to evict cached block which is not included in TLBs. Assuming 48-bit physical address space, the space overhead of GIPT is 36 bits for PPN, 42 bits for PTEP (whose 6 LSBs are always zero for alignment) and 4 bits for TLB residence bit vector (assuming a quad-core CPU), hence 82 bits per entry. For a 1GB DRAM cache, the GIPT occupies 2.56 MB of storage (0.25% overhead), which can be placed in either in-package DRAM or off-package DRAM. Thus, GIPT is much more scalable than



(a) Initial State **(b) VA-3 allocated and PTE updated** **(c) Head pointer updated and CA-0 freed** **(d)VA-2 cached**

Figure 5: Running example of tagless caching operations

the tag array of the same size.

The cache eviction algorithm dequeues the top entry from the free queue to fetch the cache address of the block to evict. Then it uses the cache address as index to look up the GIPT, to recover the off-package PPN of the block and its PTE. If dirty, the cache block is written back to the off-package DRAM. Finally, the PPN field of the PTE is replaced by the recovered PPN.

3.3. Example Walk-Through

To demonstrate the operations of the tagless cache, we walk through an example scenario shown in Figure 5. Figure 5(a) depicts the initial snapshot of the tagless cache state. The three entities on the left column are cTLB, page table, and GIPT, respectively. The right column shows DRAM cache, off-package DRAM, header pointer (HP), and the number of available free blocks (α), where α is assumed to be one. The two-bit field in the page table corresponds to a pair of VC (Valid-in-Cache) and NC (Non-Cacheable) bits; the PU (Pending Update) bit is omitted in the assumption of no concurrent TLB misses on the same VA.

Step 1 (off-package miss): Figure 5(b) shows a snapshot after a memory access to VA-3 is completed. Initially, VA-3 is cacheable but not cached and an access to it causes a TLB miss. The TLB miss handler performs a page table walk to fetch its PTE, and gets allocated a free cache block (whose address is CA-3) to initiate a cache fill. The corresponding GIPT entry is updated to back up its physical address (PPN), which is PA-3, and PTE pointer (PTEP). The page table and cTLB are also updated with a virtual-to-cache address mapping (i.e., VA-3 to CA-3) and new flag bits (i.e., (VC, NC) = (1, 0)). Although not shown, the TLB miss handler also enqueues a victim block that is currently not resident in any TLB (e.g., CA-0 in this example) to the free queue as the number of free blocks goes below α.

Step 2 (cache block eviction): Figure 5(c) captures a snapshot after an asynchronous cache block eviction process is completed. Assuming CA-0 is at the top of the free queue and is not included in any TLBs, represented NI, the GIPT is looked up to recover its corresponding PA (PA-0) and PTEP. If the block is dirty, the cached data is written back to the off-package memory at PA-0. Then the PTE pointed to by the recovered PTEP (which corresponds to VA-0 in this example) is updated with the PPN to complete the eviction process. Finally, HP is updated to point to the new free block in the cache, now available for future allocations.

Step 3 (in-package victim hit): Figure 5(d) shows a snapshot after a memory access to VA-2 is completed, which causes an in-package victim hit. Again, a TLB miss triggers a page table walk for VA-2, and the PTE indicates that the block is currently placed in package. Then the TLB miss handler simply returns with the cache address (CA-2) to establish a virtual-to-cache address mapping (i.e., VA-2 to CA-2) at cTLB.

3.4. Cache Hit and Miss Latency

Table 1 summarizes four possible cases leading to different memory access times for a memory access. When a memory access hits in both the TLB and the cache, it leads to the lowest access latency. The second case, in which the TLB hits but the DRAM cache misses, can only happen to non-cacheable pages. In this case, a memory request experiences miss latency as if there was no DRAM cache. The third case is a case of in-package victim hit, where there is no additional penalty except for the TLB miss handling cost (which cannot be avoided, anyway). Finally, the last one shows a cold miss case in which it costs the latency for a cache fill and GIPT update.

Figure 6 visualizes two of those cases, when (TLB, DRAM cache) = (*Hit*, *Hit*) and (*Miss*, *Miss*), for comparison. Figure 6(a) demonstrates that the tagless cache achieves lower latency as it removes a tag-checking overhead from the access path. In the cold miss case, the tagless cache saves tag-checking overhead (denoted by "SRAM-tag") but pays the cost

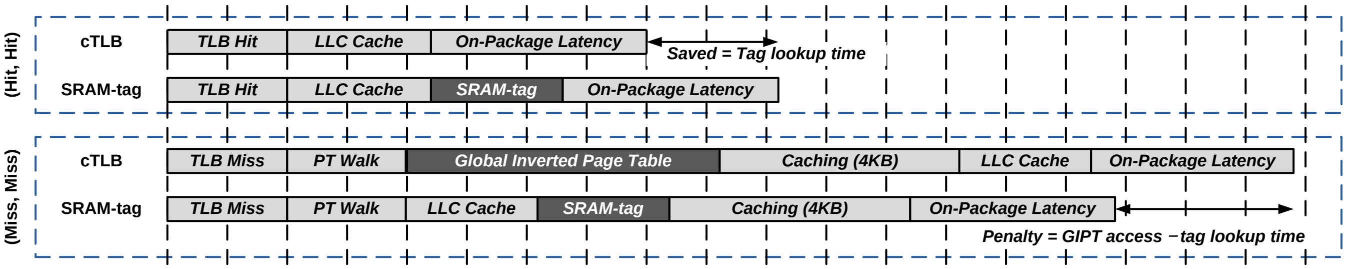


Figure 6: Compared to hit and miss penalty of SRAM-tag and cTLB cache (drawn not to scale)

TLB	DRAM cache	Descriptions
Hit	Hit	Cache hit. Zero latency penalty.
Hit	Miss	Non-cacheable page. Costs off-package block access time.
Miss	Hit	In-package victim hit. Zero latency penalty (except for TLB miss penalty).
Miss	Miss	Off-package cache miss. Costs cache fill and GIPT update latency.

Table 1: Four possible cases for a memory access

of a GIPT update (denoted by "Global Inverted Page Table"). Assuming FIFO replacement policy, the access pattern of GIPT update features very high locality as HP is incremented one by one. This makes MMU caching highly effective to reduce the cost of GIPT updates. In our evaluation, however, we calculate the penalty conservatively to pay the cost two full memory writes to off-package DRAM.

3.5. Non-Cacheable Pages

The proposed tagless cache provides a flag in the page table to specify non-cacheable pages, which bypass the in-package DRAM cache (but not the higher-level caches on die). An existing memory allocation API (e.g., `mmap`) can be readily extended for the OS or user programs to control the caching behavior of individual pages. A cache fill request to such a page directly goes to off-package DRAM to fetch a 64-byte block to be forwarded to the requester (e.g., on-die L2 cache). We identify three important use cases of this flexible caching mechanism that the tagless cache offers.

Alleviating over-fetching problem. Page-based DRAM caches suffer the over-fetching problem [20, 21, 22], which pollutes scarce off-package bandwidth and reduces effective cache size. Jevdjic et al. find that a significant fraction of pages from server workloads are *singletons*, which contain only a single useful block [21]. Moreover, some 95% of those pages show no reuse. For such pages, it would be beneficial not to allocate in-package DRAM. Exploiting the non-cacheable flag, it is feasible to alleviate the over-fetching problem by applying existing solutions, such as footprint caching [20, 21], online hot/cold page tracking [22, 30], and even offline profiling. Section 5.4 presents a case study, which demonstrates the performance potentials of non-cacheable pages with flexible placement of memory pages for the in-package DRAM cache.

Design requirement	Block-based	Page-based	Tagless (This work)
Small tag storage	bad	good	**best**
High hit ratio	bad	good	**best**
Low hit latency	bad	good	**best**
High DRAM row buf. locality	bad	**good**	**good**
Minimal over-fetching	**good**	bad	bad

Table 2: Comparison of different DRAM cache designs

Shared page support for multiple processes. If multiple page tables share the same physical page, it can cause a *page aliasing* problem, where the physical page is cached at multiple different locations. This problem occurs because the TLB miss handler does not have an easy way to check whether the requested physical page is already cached or not. A software TLB miss handler can easily avoid this problem as modern OS kernels (e.g., Linux) keep track of PTEs that map to the same page. Then the TLB miss handler can iterate over those PTEs to replace the PA with the CA at a cache fill. It can be trickier for hardware TLB miss handlers, and we will discuss this issue in more depth in Section 6. Regardless of hardware or software handlers, an easy solution to the problem is to declare all shared pages as non-cacheable to eliminate aliases, which we adopt for this work. Note that a page shared between *threads* in the same process is still cacheable as they share the same page table with no aliasing.

Superpage support. The caching granularity in the proposed tagless cache is aligned to OS page size. However, to effectively increase TLB reach, many modern architectures support superpages (e.g., 2MB, 4MB, and 1GB for x86_64 [9, 18]). Since a superpage forces coarse-grained usage of the DRAM cache, OS must carefully allocate superpages to the cache—only if there is sufficient spatial and temporal locality. If not, it would be safe to specify superpages as non-cacheable. We discuss more about this issue in Section 6.

3.6. Summary

Table 2 compares conventional block-based and page-based caches with the proposed tagless cache. The tagless cache achieves the best hit latency, in-package hit ratio (by exploiting spatial locality and minimizing conflict misses), and tag

Component	Parameters
CPU	Out-of-order, 4 cores, 3GHz
L1 TLB	32I/32D entries per core
L2 TLB	512 entries per core
L1 cache	4-way, 32KB I-cache, 32KB D-cache, 64B line, 2 cycles
L2 cache	16-way, 2MB shared cache per core, 64B line, 6 cycles
SRAM-tag Array	16-way, 256K entries
In-package DRAM (1GB)	
Bus frequency	1.6GHz (DDR 3.2GHz)
Channel and Rank	1 channel, 2 ranks
Bank	16 banks per rank
Bus width	128 bits per channel
Off-package DRAM (8GB)	
Bus frequency	800MHz (DDR 1.6GHz)
Channel and Rank	1 channel, 2 ranks
Bank	64 banks per rank
Bus width	64 bits per channel

Table 3: Architectural parameters

storage overhead (none of tags or any other auxiliary on-die SRAM structures required). It also effectively utilizes DRAM row buffer locality by fetching the entire DRAM row at a cache miss, hence amortizing the row activation overhead. The major downside of the tagless cache, actually common to most of the page-based caches, is the cost of over-fetching in terms of off-package bandwidth pollution and in-package DRAM capacity penalty. However, there are several recent proposals to reduce this overhead [20, 21, 22], which can be integrated into our framework to overcome this problem. We demonstrate its feasibility in Section 5.4 by not caching those pages with little reuse as in [21].

4. Experimental Setup

We evaluate the performance and energy efficiency of the proposed tagless DRAM cache using McSimA+ simulator [5]. Table 3 summarizes architectural parameters. The modeled system has four out-of-order cores with 1GB in-package DRAM, which is directly connected to the CPU die with TSV channels, and 8GB off-package DDR3-based DRAM. We assume the DRAM cache is deployed as L3 cache. The bandwidth of in-package DRAM is four times greater than that of off-package DRAM. The timing and energy parameters for both in-package and off-package DRAMs are adapted from recent work with die-stacked DRAM models [34], where a modified version of CACTI-3DD [10] is used. The I/O energy is reduced from 4pJ/b to 2.4pJ/b as silicon interposer-based memory channels are replaced with bumps with TSV. Table 4 shows the details of timing and power parameters for both in-package and off-package devices. We use McPAT [25] to extract power and timing parameters for the out-of-order cores and caches.

We use single-programmed, multi-programmed, and multi-threaded workloads for evaluation. For single-programmed workloads we select 11 most memory-bound programs according to their misses per kilo instructions (MPKI). We run Simpoint [32] to identify representative phases from which we choose top 4 slices with the highest weights. Each slice contains 100 million instructions. For multi-programmed workloads, we randomly mix four of the 11 memory-bound SPEC CPU 2006 programs to select eight workloads. Finally, we use four PARSEC programs that our simulation framework runs without errors for multi-threaded workloads: `swaptions`, `facesim`, `fluidanimate`, and `streamcluster`.

Parameter	In-package DRAM	Off-package DRAM
I/O energy	2.4pJ/b	20pJ/b
RD or WR energy without I/O	4pJ/b	13pJ/b
ACT+PRE energy (4KB page)	15nJ	15nJ
Activate to read delay (tRCD)	8ns	14ns
Read to first data delay (tAA)	10ns	14ns
Activate to precharge delay (tRAS)	22ns	35ns
Precharge command period (tRP)	14ns	14ns

Table 4: Parameters for 3D in-package DRAM and off-package DRAM devices (adapted from [34])

MIX1	milc-leslie3d-omnetpp-sphinx3
MIX2	milc-leslie3d-soplex-omnetpp
MIX3	milc-soplex-GemsFDTD-omnetpp
MIX4	soplex-GemsFDTD-lbm-omnetpp
MIX5	mcf-soplex-GemsFDTD-lbm
MIX6	mcf-leslie3d-lbm-sphinx3
MIX7	milc-soplex-lbm-sphinx3
MIX8	mcf-leslie3d-GemsFDTD-omnetpp

Table 5: Workload groupings (memory-bound applications)

We compare the tagless cache with the following designs:

No L3 cache (No L3): This setting serves as the baseline system, which is a conventional off-package DDR3-based memory system with no DRAM cache.

Bank-interleaving (BI): This design models a heterogeneous memory system, where the in-package DRAM region is mapped to the global physical memory space, but OS is oblivious of this heterogeneity, hence not performing any intelligent page placement or migration.

SRAM-tag (SRAM): This design models a page-based SRAM-tag array cache similar to the one in [21] with 4KB page size. Table 6 summarizes the tag size and access latency as a function of cache size, obtained from CACTI-6.5 [2].

cTLB: This is our proposed tagless cache, which is based on cTLB. A TLB hit guarantees a cache hit. The TLB miss handler not only performs the page table walk, but also fills in the requested cache block if not cached. The TLB miss penalty also includes cycles for accessing the global inverted page table (GIPT), which is consulted for cache block replacement.

Ideal: This setting models an ideal in-package caching system, where all data are stored in in-package DRAM.

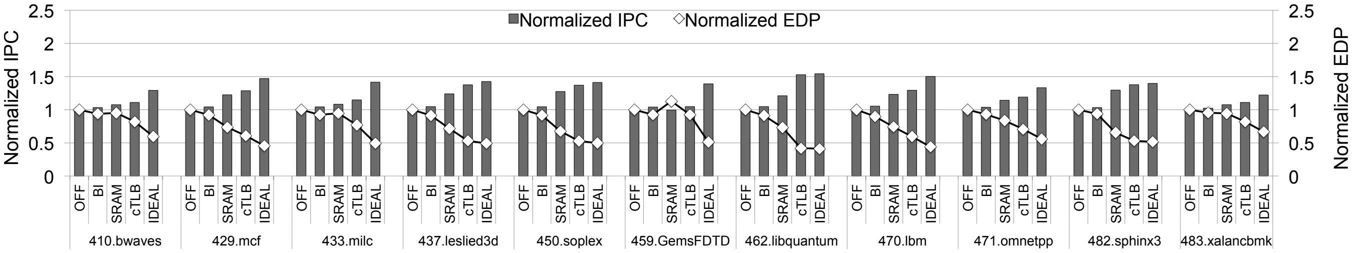


Figure 7: IPC and EDP normalized to the baseline (with no L3) for selected SPEC CPU 2006 programs

Cache size	128MB	256MB	512MB	1GB
Tags size	0.5MB	1MB	2MB	4MB
Latency (cycles)	5	6	9	11

Table 6: Cache parameters

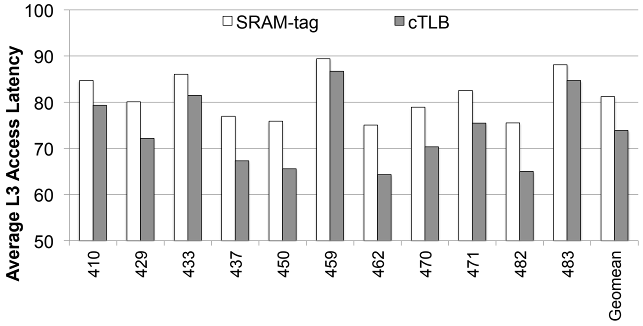


Figure 8: The average L3 access latency of SRAM-tag and tagless caches (the lower, the better)

5. Evaluation

5.1. Single-Programmed Workloads

IPC and EDP. Figure 7 shows the IPC and EDP of 11 memory-bound SPEC CPU 2006 programs. The numbers are normalized to the baseline (with no L3 cache). The heterogeneity-oblivious bank-interleaving scheme shows only 4.0% of IPC improvement. In contrast, the SRAM-tag and tagless caches, effectively exploiting fast in-package DRAM, show 16.4% and 24.9% of IPC improvements, respectively. The latter falls within 11.8% of the IPC of the "ideal" cache. In terms of energy-delay product (EDP), the tagless cache outperforms the SRAM-tag cache by 26.5%. `459.GemsFDTD` and `433.milc` have relatively large IPC gaps from the ideal cache, by 32.8% and 23.3%, respectively. This is because a large fraction of memory pages in these programs have low reuse rates for the DRAM cache, and hence low hit rates.

Average L3 Access Time. Figure 8 shows average L3 access time. It is calculated based on Equations 1 and 4, but only access latency after a L2 cache miss (including TLB access time) is counted. The tagless cache consistently yields lower latency than the SRAM-tag cache due to no tag-checking overhead, by up to 16.7% for `462.libquantum` with a geomean latency reduction of 9.9%. `459.GemsFDTD` has many pages touched for the first time, which are accessed from off-package DRAM, hence yielding no significant difference between the two.

5.2. Multi-Programmed Workloads

As multi-programmed workloads quadruple the memory footprint, we can better evaluate the performance of the proposed tagless cache, reflecting the impact of cache contention and replacement policy. For this reason we use these workloads for several sensitivity analyses. Note that we set α, the number of available free blocks, to one as in [12]. A FIFO replacement policy is used for the tagless cache and LRU is used for the SRAM-tag cache.

IPC and EDP. Figure 9 shows the normalized IPC and EDP to the baseline with no L3 cache. The SRAM-tag and tagless caches show 34.9% and 38.4% of IPC improvements, respectively. The normalized EDP reductions of SRAM-tag and tagless caches are 31.5% and 43.5%, respectively. The tagless cache outperforms the SRAM-tag cache for IPC by 2.6% and energy consumption by 21.3%. We expect this gap to widen as in-package DRAM size increases, which makes latency and energy overheads from the tags more pronounced. The heterogeneity-oblivious bank-interleaving scheme achieves only an 11.2% of IPC improvement.

Sensitivity to DRAM Cache Size. Figure 10 shows the normalized IPC to the baseline with the bank-interleaving policy according to DRAM cache size. When the DRAM cache is small (256MB), the IPC is degraded by some 30% over the bank-interleaving scheme due to excessive contentions at the DRAM cache. This leads to frequent cache block migrations between in- and off-package DRAMs. The effect of cache block thrashing gets amplified by coarse-grained page-based caching, and there is no significant performance difference between the SRAM-tag and tagless caches. However, if the DRAM cache size increases to be 512MB or greater, the page migration rate decreases significantly for the tagless cache to benefit from fast hit time. Although the amount of IPC improvement varies depending on workload characteristics, the tagless cache consistently outperforms the SRAM-tag cache for large cache sizes.

Sensitivity to Replacement Policy. The default FIFO replacement policy simplifies victim selection and eliminates the cost of auxiliary data structure for replacement. It is known that, as the cache size and/or associativity increase, the performance impact of the replacement policy is reduced [15]. Figure 11 confirms this fact by comparing FIFO and LRU replacement policies. The LRU replacement policy outperforms the FIFO policy only marginally, by 1.6% on average. In addition,

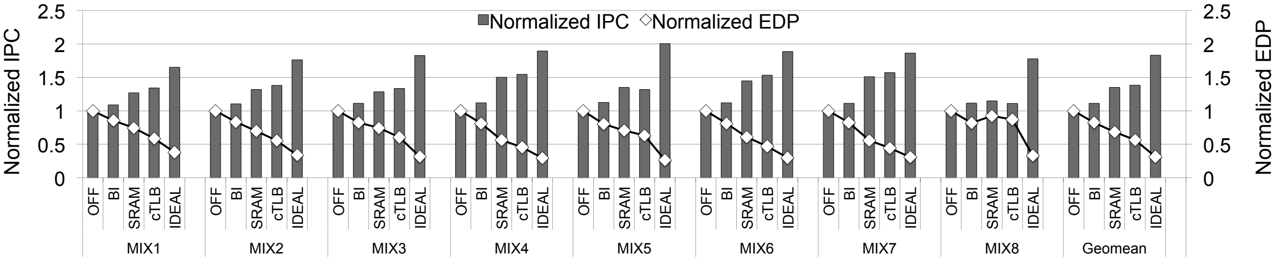


Figure 9: The normalized IPC and EDP of the multi-programmed workloads

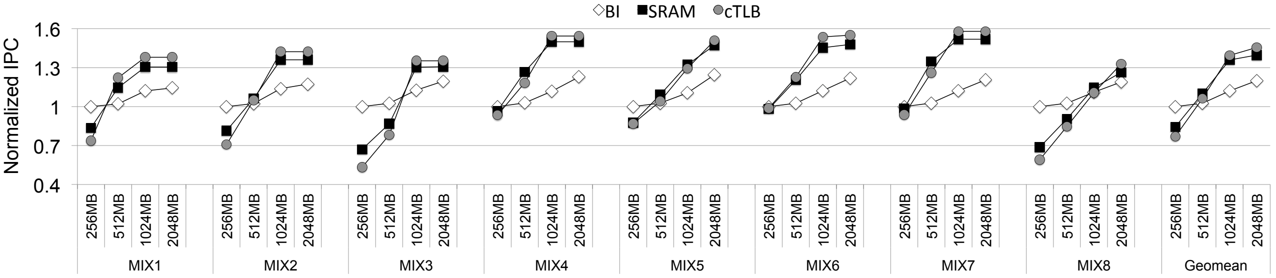


Figure 10: Sensitivity of DRAM cache size to performance in SRAM-tag and tagless caches

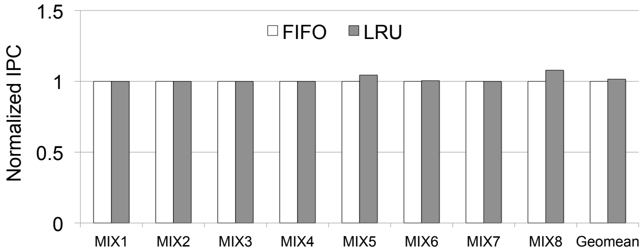


Figure 11: The performance impact of replacement policy

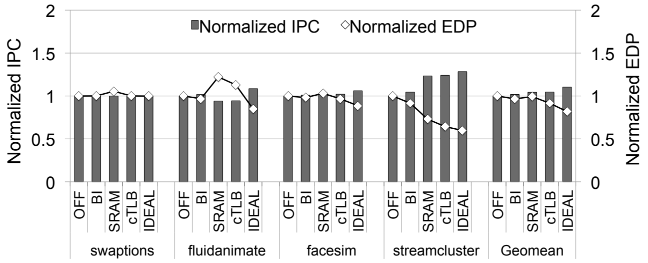


Figure 12: The IPC speedup and normalized EDP of the multi-threaded workloads

LRU-based policies (e.g., CLOCK [15]) are likely to increase storage overhead and/or cache miss penalty. Hence, a simple FIFO policy suffices in our setup.

5.3. Multi-Threaded Workloads

Figure 12 shows the normalized IPC and EDP of the four PARSEC benchmarks, which are *all* benchmarks successfully ported to our simulator. The tagless cache shows IPC improvement and EDP reduction for `streamcluster` and `facesim` over the bank-interleaving scheme. These two workloads show high page reuse ratio and high MPKI. Especially, the IPC speedup of `streamcluster` is 24.0% over the baseline with no cache (0.6% over SRAM-tag). While yielding comparable IPCs, the tagless cache reduces EDP for `facesim` over the SRAM-tag due to no energy waste for the tags. The other two workloads, `swaptions` and `fluidanimate`, show either no IPC improvement or even a little bit of degradation. These workloads have a large portion of singleton pages with low MPKI for the simulated slices. Accordingly, the performance overhead of caching negates the benefits of using fast in-package DRAM.

5.4. Case Study: Potential Benefits of Non-Cacheables

According to Jevdjic et al., a significant fraction of pages from server workloads are *singletons*, which contain only a single useful block with little reuse [21]. We apply this insight to `459.GemsFDTD`, which has the large IPC gap from the ideal cache as shown in Figure 7. This program features high MPKI but has a large number of pages with little reuse in the DRAM cache. In order to sort out such pages, we set the non-cacheable flags for those pages with access count smaller than 32. This number is based on the observation that a page has 64 64-byte blocks and more than a half of those blocks are expected to be touched if there is significant spatial locality.

Figure 13 shows the normalized IPC of `459.GemsFDTD`. The tagless cache leverages non-cacheable pages to further improve the IPC by 7.1% over that without non-cacheables. It is due to reduction in bandwidth pollution between in- and off-package DRAMs and increase in the hit ratio of the DRAM cache. This demonstrates that it can be beneficial to implement flexible, software-managed page classification schemes on top of the tagless cache to further improve the performance and energy efficiency, by adopting similar techniques as in [12, 30], for example.

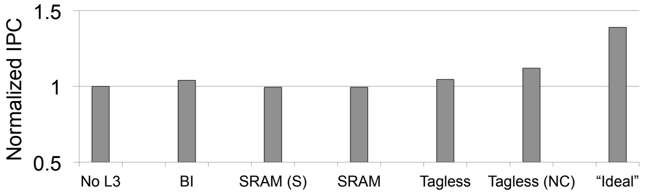


Figure 13: Performance impact of non-cacheable pages on `459.GemsFDTD`

6. Discussion

Shared page support. As an alternative to the solution in Section 3.5, one can introduce another table that maintains PA to CA mappings. Then the TLB miss handler quickly looks up the table to find any existing alias already in the cache. If such an alias exists, the handler can simply change the address mapping to point to the alias' address in the cache instead of copying the page from off-package. However, this approach incurs a latency penalty in handling a TLB miss as well as storage overhead. Although kernel pages are also shared among multiple processes, there is no aliasing problem since all processes actually share a single kernel page table. Consequently, a PTE update in the kernel address space by one process becomes immediately visible to all the other processes.

Superpage support. There is no fundamental limitation for the proposed tagless cache to support superpages as we can change the granularity of caching accordingly. However, if not judiciously applied, superpages can exacerbate the over-fetching problem to increase bandwidth pollution and in-package miss rate. Superpages might be beneficial, if there is high locality, as they improve the efficiency of DRAM access and channel transfers with bulk accesses. To support superpages, the GIPT entry must be extended to include page type information (e.g., 2-bit fields to select one of the 4KB, 2MB, 4MB, and 1GB pages). If there are a small set of hot subpages within a superpage, the OS can split a superpage into smaller pages (e.g., one 2MB page into 512 4KB pages). The hierarchical page table structure facilitates this breakdown by expanding a superpage table entry to one or more next lower-level page tables.

Multi-socket support and coherence. In a multi-socket system the last-level DRAM caches can be organized as a globally shared NUCA cache or socket-private caches. In the socket-private organization, one page may be cached in multiple sockets with different (socket-private) cache addresses. This makes coherence protocol operations more complicated (e.g., requiring another level of address translation) and imposes additional challenges in maintaining shared data structures (e.g., PTEs mapped to cache addresses), which requires further research. In contrast, the shared cache eliminates this problem and makes it easier to design cache-coherence protocols. In the shared cache, if a cached page is evicted, TLB shootdown must be done to guarantee the consistency between cached pages and TLB entries.

7. Related Work

In-package DRAM caches. DRAM caches can be classified into two categories by caching granularity: block-based and page-based. First, most of the block-based cache designs aim to reduce tag costs in terms of latency, area and energy consumption. Loh et al. [27] propose to embed cache tags in the same row of the block in DRAM so that just one row buffer open can serve both tag lookup and data access. Likewise, Alloy Cache [29] co-locates tag and data into a single row and is configured as a direct-mapped cache to further reduce cache hit latency. Huang et al. [17] propose ATCache to reduce SRAM tag overhead and achieve tag matching latency close to SRAM-tag by architecting two-level tag caches; a small SRAM-based tag cache and a large DRAM-based tag store. Woo et al. [36] propose a die-stacked DRAM cache that prefetches in page granularity but provides cache accesses in block granularity in order to reduce false sharing. Sim et al. [33] propose multi-level hit-miss predictor to reduce hardware cost while reducing cache miss latency. These block-based cache designs, however, have inherent drawbacks of storage and energy overheads from tags, which can severely limits scalability for future, very large in-package DRAM.

Page-based caches overcome this limitation by caching at a coarser (page) granularity. Most of the page-based caches aim to alleviate the over-fetching problem. Footprint cache [21] and Unison cache [20] reduce off-package memory traffic by predicting blocks with high reuse and caching them selectively. Jiang et al. [22] propose filter-based DRAM caching to cache only hot pages. The tagless cache is complementary to these proposals–primarily aiming to eliminate the tag overhead by consolidating two-phase address translation into single phase. In addition, our scheme implements a fully associative cache which is not viable in other tag-based caches.

Heterogeneous memory systems. The main challenge in architecting software-managed heterogeneous memory system with fast and slow regions is to efficiently identify and migrate hot/cold pages [12, 23, 30, 35]. Dong et al. [12] propose a sophisticated remapping structure and algorithms to transparently migrate and map critical pages. This approach, however, can have inherent disadvantage of increasing memory access latency as every memory accesses should pass through the remapping table. Ramos et al. [30] propose a ranking algorithm to identify critical pages in hardware by exploiting multi-queue replacement policy. They implement the classification algorithm in the memory controller, which can cost area and latency penalties. Also, our scheme provides additional flexibility to easily support various caching policies.

8. Conclusion

This paper introduces a fully associative, tagless page-based DRAM cache that minimizes latency/storage/energy overheads associated with large cache tags. The key idea is to align the granularity of caching with OS page size and consolidate the mechanisms for address translation and cache management into the TLB miss handler. The resulting tagless cache design achieves multiple, conflicting design goals such as low hit latency, high hit rate, high energy efficiency, and flexibility. Moreover, the tagless cache fundamentally resolves the scaling problem of future large in-package DRAM caches by completely eliminating cache tags as well as any auxiliary on-die SRAM structures.

Acknowledgments

We would like to thank Jung Ho Ahn and Young Hoon Son for their help with die-stacked DRAM modeling. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2014R1A1A1005894 and NRF-2014R1A1A1003746) and the Ministry of Education (NRF-2014R1A1A2054658).

References

[1] "AMD Working With Hynix For Development of High-Bandwidth 3D Stacked Memory." [Online]. Available: http://wccftech.com/amd-working-hynix-development-highbandwidth-3d-stacked-memory

[2] "CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model." [Online]. Available: http://www.hpl.hp.com/research/cacti

[3] "Intel unveils 72-core x86 Knights Landing CPU for exascale supercomputing." [Online]. Available: http://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing

[4] "Interview: Masaaki Tsuruta, Sony Computer Entertainment." [Online]. Available: http://eandt.theiet.org/magazine/2011/12/maasaki-tsu-interview.cfm

[5] "McSim Simulator." [Online]. Available: http://scale.snu.ac.kr/mcsim

[6] "Nvidia to Stack up DRAM on Future Volta GPUs." [Online]. Available: http://www.theregister.co.uk/2013/03/19

[7] "The SAP HANA Database." [Online]. Available: http://www.sap.com/HANA

[8] "Xilinx SSI Technology." [Online]. Available: http://www.hotchips.org/archives/hc24

[9] AMD, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, May 2013.

[10] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Mar 2012.

[11] J. R. Cubillo, R. Weerasekera, Z. Z. Oo, E.-X. Liu, B. Conn, S. Bhattacharya, and R. Patti, "Interconnect design and analysis for through silicon interposers (TSIs)," in *Proceedings of the 2011 IEEE International 3D Systems Integration Conference (3DIC)*, Jan/Feb 2012.

[12] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2010.

[13] P. Gillingham and B. Millar, "High bandwidth memory interface," Jan. 21 2003, US Patent 6,510,503.

[14] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.

[15] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[16] J. L. Henning, "SPEC CPU2006 Memory Footprint," *Computer Architecture News*, vol. 35, no. 1, Mar. 2007.

[17] C.-C. Huang and V. Nagarajan, "ATCache: reducing DRAM cache latency via a small SRAM tag cache," in *Proceedings of the 23rd international conference on Parallel Architectures and Compilation Techniques (PACT)*, Aug 2014.

[18] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, September 2014.

[19] S. S. Iyer, "The Evolution of Dense Embedded Memory in High Performance Logic Technologies," in *Proceedings of the IEEE International Electron Devices Meeting (IEDM)*, Dec 2012.

[20] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked DRAM cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2014.

[21] 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 *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, Jun 2013.

[22] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian, "CHOP: Adaptive filter-based DRAM caching for CMP server platforms," in *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, Jan 2010.

[23] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power Aware Page Allocation," in *Proceedings of the 9th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov 2000.

[24] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2 A 1.2 V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," in *Proceedings of 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014.

[25] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009.

[26] G. H. Loh, "Extending the effectiveness of 3d-stacked DRAM caches with an adaptive multi-queue policy," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009.

[27] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2011.

[28] J. T. Pawlowski, "Hybrid Memory Cube," in *Hot Chips*, Aug 2011.

[29] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2012.

[30] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the International Conference on Supercomputing (ICS)*, Jun 2011.

[31] S. L. Salzberg, A. M. Phillippy, A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, G. Marais, M. Pop, and J. A. Yorke, "GAGE: A critical evaluation of genome assemblies and assembly algorithms," *Genome Research*, Dec 2011.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2002.

[33] 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 *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2012.

[34] Y. H. Son, O. Seongil, H. Yang, D. Jung, J. H. Ahn, J. Kim, J. Kim, and J. W. Lee, "Microbank: architecting through-silicon interposer-based main memory systems," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Dec 2014.

[35] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie, "Adaptive placement and migration policy for an STT-RAM-based hybrid cache," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014.

[36] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. Lee, "An optimized 3d-stacked memory architecture by exploiting excessive, high-density TSV bandwidth," in *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, Jan 2010.

[37] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, Mar 1995.

[38] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *Proceedings of the 25th International Conference on Computer Design (ICCD)*, Oct 2007.