CT-Cache: Compressed Tag-Driven Cache Architecture

Haeyoon Cho*, Joonho Kong*, Arslan Munir†, Naresh Kumar Giri†
*School of Electronics Engineering, Kyungpook National University, South Korea
†Department of Computer Science, Kansas State University, Manhattan, KS
{haeyoon.cho, joonho.kong}@knu.ac.kr, {amunir, ngiri}@ksu.edu

Abstract—**One of the most challenging problems in designing large caches is to devise efficient cache tag storage. Typical large last-level caches often require tens of megabytes of tag storage. Consequently, this large tag storage leads to high latency and energy consumption for a cache tag access, which adversely affect system performance and energy efficiency. In this paper, we propose CT-Cache that exploits the similarity in upper cache tag bits. Our proposed CT-Cache decouples a tag storage into a global tag table and delta tag arrays. The global tag table is a small fully-associative buffer that stores upper tag patterns, which can be shared between multiple cache lines, while the delta tag array stores lower tag bits for each cache line. By avoiding the redundant storage, the CT-Cache significantly reduces the cache tag storage size, which in turn reduces latency and energy consumption of a cache tag access. Evaluation results reveal that the CT-Cache reduces tag storage size by 87.8%, which significantly reduces tag access latency and energy. Owing to the tag storage reduction, the CT-Cache improves performance by 4.7%∼16.2% and reduces last-level cache and main memory energy consumption by 20.0%∼29.7% compared to the conventional caches that use non-compressed tag storage.**

Index Terms—**Tag compression, Cache architecture, Last-Level cache, Locality, Performance, Energy efficiency**

I. Introduction

To enhance the performance of modern processors, the number of cores housed in a single chip is increasing. However, the memory system has not kept pace in terms of both capacity and bandwidth to supply required amount of data to all processor cores to sustain desired performance enhancement of processors. This discrepancy between processor and memory performance is known as "memory wall" and greatly influences the overall performance of computing systems. To exacerbate the memory wall problem, modern big data applications involve data-centric computations, which further increases the pressure on memory system. However, advancements in integration and device technologies (e.g., 3D-stacking technology and embedded dynamic random access memory) have allowed to integrate large dynamic random access memory (DRAM) onto the processor chip. The integrated (e)DRAM (eDRAM stands for embedded DRAM and is a DRAM integrated on the same die as the application-specific integrated circuit (ASIC) or multi-processor) size can be in order of hundreds of megabytes (MB) to few gigabytes (GB). Although, large (e)DRAM last-level caches (LLCs) can be designed and realized by such novel integration and device technologies, there are several challenges that need to be addressed to effectively employ large caches. When the cache size increases to hundreds of megabytes, the cache metadata, such as tag bits, dirty/valid bit, coherency management bits, state information for implementing cache replacement policy, etc., also increase. For example, a cache of 256 MB would have an associated tag size of 11.5 MB. Similarly, for a cache size of 1024 MB, the tag size would be up to 42 MB [4]. Thus, it is not feasible to store tags in SRAM as it would result in high area overhead.

To resolve the tag storage overhead, several prior works [5], [10], [14], [15] aim to address cache tag storage problem for large caches. The work in [10], [14], and [15] stores cache tags in (e)DRAM so that the cache tags do not require a huge number of SRAM cells (e.g., tens of megabytes) for tag storage, alleviating the area overhead. However, these approaches typically have high latency mainly because the cache tag access entails a slow (e)DRAM access. Longer tag access latency adversely affects system performance and energy efficiency as the cache tag access lies in the critical path of the cache access time in large caches that employ sequential tag and data access. One may use large cache line size (e.g., page-size) to reduce tag storage overhead (e.g., [5]). However, the large cache line size would waste a huge memory bandwidth in order to transfer large block size data when a cache miss occurs.

In this paper, we present an efficient cache architecture (referred to as CT-Cache) for large caches. In our proposed architecture, the cache tags are split into two parts such that the upper parts contain bits that are common/same among multiple cache lines and the lower parts contain bits that are unique/different. Since there is a significant similarity in patterns of upper bits in memory addresses due to the principle of locality, our cache architecture stores the common tag bit parts in a global tag table (GTT) which is implemented in a small buffer (∼1KB). This eliminates the need for storing multiple instances of a common/same value, which helps in reducing storage size, area, energy, etc., of the cache tag storage. The lower part of the cache tag, which uniquely identifies each cache line, is stored in a separate array which is referred to as delta tag array (DTA). For a cache hit to occur, the upper part of the tag must match with a value in the GTT and the lower part of the tag must match with a value in the DTA. To efficiently manage GTT and DTA, our proposed CT-Cache employs a flexible mapping (see Section III-A2 for details) between the GTT entries and a set of cache lines. Furthermore, proactive cache management in the CT-Cache (see Section III-A4 for details) periodically rearranges a mapping between the GTT entries and a set of cache lines to guarantee fairness between the GTT entries. These features further enhance cache space utilization, resulting in improved performance and energy efficiency.

To the best of our knowledge, our work is the first one that actually reduces the tag storage for large LLCs while not depending on the type of memory cells used in the cache. This independence on memory cell types means that the CT-Cache can also be applied to large SRAM-based as well as (e)DRAM-based caches whereas several previous proposals depended on a certain memory cell-based caches (e.g., DRAM caches [5], [10], [14]). Although several works introduced cache compression schemes for large-scale caches (e.g., [11], [16]), these works have purposed to compress cache data while the cache tag size remains large.

The main contributions of this paper are as follows:

- We propose a novel cache architecture (CT-Cache) for large caches by leveraging cache tag compression.
- Our proposed CT-Cache is independent of cache memory cell technology and can be applied to any large-scale cache (e.g., SRAM-based, (e)DRAM-based, etc.).
- Our proposed CT-Cache stores tags in 87.8% lesser area, imparts a performance improvement of 4.7%∼16.2%, and reduces LLC and main memory energy consumption by 20.0%∼29.7% as compared to conventional caches.

- Due to the flexible mapping between the GTT entries and a set of cache lines and proactive cache management, the CT-Cache also shows stable and consistent performance and energy efficiency improvements compared to CT-Static (static mapping between the GTT entries and a set of cache lines) and CT-w/oPAD (CT-Cache without proactive cache management).

Remainder of this paper is organized as follows. Section II summarizes related work and Section III presents our proposed CT-Cache architecture. Evaluation methodology and results are presented in Section IV. Lastly, Section V concludes this paper.

II. Related Work

There has been work done in literature related to large cache design particularly for efficiently storing the cache metadata. Loh and Hill [10] proposed a DRAM cache with block size of 64 bytes. Since the cache tags in their proposed cache architecture were stored in DRAM, they had to address the problem of increased miss penalty when accessing tags from DRAM. In order to combat this problem, they used MissMap on SRAM to keep track of DRAM content to avoid cache misses. However, the size of MissMap was already in the range of few MB, and thus the proposed architecture was not feasible for implementing cache tags in SRAM. Qureshi and Loh [14] addressed this issue with an Alloy Cache–a direct-mapped DRAM cache which has wider data path. Their proposed cache architecture improved the access latency as data and tag could be accessed in a single fetch. Their model worked good for direct-mapped caches but was inflexible for set-associative caches.

Jevdjic et al. [5] proposed Footprint Cache which combined the benefits of both block-based and page-based cache design. They designed a footprint predictor which only fetched selected blocks from the accessed page. They opted to only store page tags in order to reduce tag size. However, their design is geared towards DRAM-based caches while it would not be suitable for SRAM-based or eDRAM-based large caches. Huang et al. [4] proposed ATCache that stores entire tags in DRAM while the recently or frequently accessed tags in a small SRAM buffer. Although, their design aimed to reduce cache hit latency, cache tag storage size was still large (e.g., a 256MB DRAM cache requires 11.5 MB tag size).

Yang et al. [15] proposed a new tag scheme for DRAM cache using eDRAM technology. In this model, the tag was stored in eDRAM to alleviate the overhead of storing tags in SRAM. However, this model was exclusive to eDRAM technology and was not suitable for other technologies. TLB index-based [6] [8] tagging approaches used TLB indexes as cache tags instead of actual tag memory; however, this approach was only suitable for L1 caches where TLB access and cache access are performed simultaneously.

There have been several studies on cache compressions to improve storage efficiency of large caches. Young et al. [16], presented the idea of compressing data in DRAM caches. This approach used extra hardware for cache index predictors to reduce access latency. In [11], the authors presented base-delta-immediate compression for data in cache, which improves cache utilization and system performance. However, the studies mentioned above introduced cache data compression while the full tag bits must be stored in the storage, incurring the huge storage overhead due to the cache tags in large caches.

Petrov and Orailoglu [12] proposed a cache tag compression technique based on static (i.e., compile time) analysis of memory layouts. Although their proposed technique reduced cache power consumption, it relied on compile-time information, which made adaptive cache management at runtime difficult. Kwak and Jeon [7] proposed a cache tag compression scheme for embedded processors. Their proposed scheme also aimed at reducing tag storage by sharing upper tag patterns in a locality buffer. However, their proposed architecture still required to store the entry number of the locality buffer in the

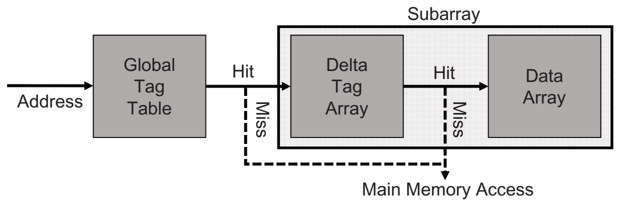

Fig. 1: Cache access flow of CT-Cache. There are many subarrays in the cache although we only show one subarray in the figure for brevity. We call a bundle of a delta tag array and corresponding data array as a subarray.

tag array, leading to lower tag compression ratio. Although this architecture would be efficient in small caches, it would not be suitable for large caches which should maintain a much larger number of upper tag patterns.

Our proposed CT-cache efficiently compresses cache tags, which makes storage of cache tags in SRAM feasible. Our approach permits saving space of multiple instances of redundant cache tag bits, efficiently reducing the tag storage.

III. Compressed Tag-Driven Cache Architecture

A. Architecture Overview

1) Decoupled Tag Storage: To avoid redundant storage of upper tag bits, we propose to decouple the conventional tag arrays into two parts: GTT and DTA. The common upper bit parts are stored in a small storage, which is referred to as GTT, while the remaining unique lower bit parts are stored in the DTA. Consequently, we need much smaller arrays for tag storages as we remove redundant storages for upper tag bits. The GTT is a small fully-associative buffer (in this work, we use 32-entry GTT) that contains the upper part of the tag bits while the DTAs contain the remaining bits of the tags that are associated with each cache line. Since we store the upper bit patterns in the GTT, a single GTT entry must be shared between multiple cache lines. As we use decoupled tag storages, the tag matching procedure of CT-Cache is also composed of two steps: GTT access and DTA access. In the case of both GTT hit and delta tag hit, it is regarded as a cache hit and we perform a data array access. If either the GTT miss or delta tag miss occurs, it is regarded as a cache miss and the request goes to the main memory.

2) Cache Management: In our design, since a single GTT entry must be shared among multiple cache lines, it is very crucial to determine the mapping rule between the GTT entries and cache lines. For efficient design, we group multiple cache lines into a single subarray and we define the mapping rule between the GTT entries and subarrays. A subarray logically contains data array and corresponding DTA[1]. Each subarray works as a direct mapped cache, which means data with a certain address can be mapped to only one place in a single subarray. Although capacity of a single subarray size can vary depending on cache design, one subarray has 2MB of data storage and associated delta tag storage in this work.

For a simple implementation, we could statically assign the cache subarrays into each GTT entry. For example, if we have a 32-entry GTT and 128 subarrays, we can statically assign four subarrays to each GTT entry[2]. However, this static assignment may worsen cache capacity utilization in the case where the required cache capacity for each GTT entry is diverse. To address this problem, we dynamically allocate and deallocate subarrays to GTT entries. As shown in Fig. 2, GTT entries can have various number of cache subarrays at runtime. Since a single subarray works as a direct-mapped cache, if N subarrays are allocated to a certain GTT entry, these subarrays work as an N-way set-associative

[1]As shown in Fig. 1, though we logically combine a data array and DTA into a single subarray, the data array and DTA can be physically far apart in the cache layout.

[2]We will also evaluate this scheme in evaluations (referred to as 'CT-Static').

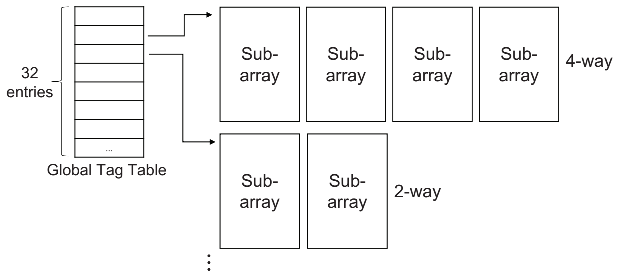

Fig. 2: A conceptual block diagram of the global tag table entries and their allocated subarrays.

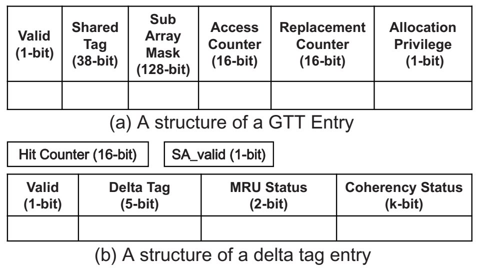

Fig. 3: Structure of the global tag table entry and delta tag entry.

cache. This dynamic allocation/deallocation enables a flexible management of the cache capacity allocated to each GTT entry, which in turn leads to better performance and energy efficiency.

3) Structure and Management of Global Tag Table and Delta Tag Array: The structure of the GTT is shown in Fig. 3(a). It can be implemented by using both CAM (content-addressable memory) and SRAM cells. A shared tag storage in the GTT is implemented by CAM cells to enable a fast fully-associative search while the remaining metadata storage uses SRAM cells. The subarray mask bits record which subarrays are currently allocated to the GTT entry. For example, if the subarray mask bit[0] and mask bit[1] are '1' while the remaining mask bits are '0', it means that the subarray[0] and subarray[1] are allocated to the corresponding GTT entry. In our design, there are total 128 subarrays (although the number of subarrays could vary depending on the cache design). Hence, we need 128 bits for subarray mask bits of each GTT entry.

For dynamic management of GTT entries, there are 16-bit access and replacement counters for each entry. The access counter maintains the number of accesses to the GTT entry within a pre-defined time interval (10 million clock cycles in this work). When replacing the GTT entry, we refer to the access counter for each entry and the entry with the least access count becomes a victim (i.e., replacement choice). The replacement counter indicates how many times the cache line replacements from the allocated subarrays occur within the time interval. An allocation privilege bit is set to '1' if the corresponding GTT entry is given a right to get a new subarray allocation. Detailed usages of the counters and allocation privilege bit will be explained in Section III-B.

To maintain the data coherency, the replacement or invalidation of a GTT entry entails bulk invalidations[3] and flushes of the data stored in the subarrays allocated to the evicted GTT entry. In the case of using write-back caches as in conventional caches, we need to perform bursty flush operations, which would issue a huge number of write-back requests to the main memory. These bursty flush operations may degrade system performance mainly due to the limited write buffer size. Thus, our proposed CT-Cache uses write-through caches so that the write requests to the main memory can be served sporadically.

An entry structure of the DTA is also shown in Fig. 3(b). The delta tag entry (array) is similar to the conventional tag entry[4] (array) while the main differences are: 1) we only store lower tag bits (5-bit in this work) for each tag entry, 2) 2 bits are added to each entry for MRU (most recently used) status bits, and 3) 16-bit hit counter and 1-bit SA_valid bit (i.e., subarray valid bit) are added to each subarray. Obviously, we only store the lower tag parts in the DTAs as we already store the upper part tags in the GTT entry. The MRU status bits are required for the victim cache line selection. In general, we can apply least recently used (LRU) policy to select a victim. However, in our proposed CT-Cache design, implementing LRU is not straightforward as there can be various associativities across sets of the subarrays associated with each GTT entry. Hence, to simplify the design, we track only two MRU (most recently used) cache lines in a set and a victim is randomly selected from the cache lines except the two MRU lines. The hit counter maintains how many cache hits occur in the subarray. This information is used for dynamic allocation and deallocation of the subarrays (see the following subsection for detailed usages). The 1-bit SA_valid bit indicates whether the corresponding subarray is currently allocated to any GTT entry or not. The SA_valid bit is used when we search for available subarrays (i.e., subarrays that are currently not allocated to any GTT entry). Please note that the 16-bit hit counter and 1-bit SA_valid bit are employed not for each entry but for each subarray.

4) Proactive Subarray Allocation and Deallocation: Our proposed CT-Cache proactively manages allocation and deallocation between GTT entries and subarrays for efficient cache utilization. For example, a certain GTT entry may require more cache subarrays to prevent conflict misses while another GTT entry may hold the subarrays that mostly contain dead cache lines (i.e., the cache lines that will not be used in future). We need to allocate more subarrays to the GTT entries for the former case while we need to deallocate subarrays from the GTT entries for the latter case.

We make the allocation and deallocation decisions by referring to the access counter and the replacement counter of each GTT entry. We go through replacement and access counters of each GTT entry every 10 million clock cycle interval. If (replacement count)/(access count) of a certain GTT entry is more than 0.1, we set the allocation privilege bit of the corresponding GTT entry to '1'. The allocation privilege bit of '1' means that the GTT entry gets a new subarray allocation whenever a conflict miss from the subarrays allocated to the GTT entry occurs during the next 10 million cycle interval.

After we determine the GTT entry of which allocation privilege bit is set to '1', if there is at least one GTT entry of which allocation privilege bit is '1', we also try to deallocate subarrays which are allocated to greedy GTT entries (i.e., considered to have much more subarrays than they actually need). For subarray deallocations, we find the GTT entries that satisfy the following condition: (replacement count)/(access count) of the GTT entry is less than 0.05. From each GTT entry that satisfies the condition[5], we deallocate one subarray which has the least hit count among the subarrays allocated to the selected GTT entries. Obviously, a deallocation includes invalidations of all the cache lines in the subarray and update of the corresponding subarray mask bit and SA_valid bit (from '1' to '0'). Please note that the deallocation only happens in the GTT entries that have more than one subarray (i.e., ≥ 2). We also reset the access and replacement counters of all GTT entries every 10 million cycle interval to cope with dynamic program behaviors. In evaluations, we will show performance and energy impact of our proactive subarray allocation and

[3]For easier bulk invalidations, we can manage all valid bits in the subarray within a single array. Since we use 2MB subarray with 64B cache line, we need 4KB valid bit array per subarray. By doing so, we can reduce latency for resetting all valid bits to '0'. Though it still require several cycles, it can be carried out in background, negligibly affecting performance.

[4]For the coherency status bits, the length of bits depends on the cache coherence protocol.

[5]In case that there is no GTT entry that satisfies the condition, we do not deallocate subarrays from any GTT entry.

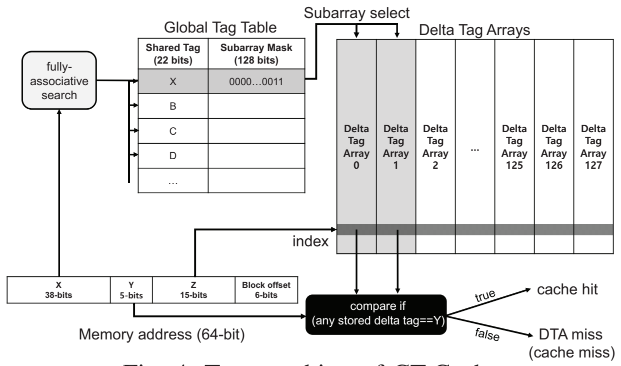

Fig. 4: Tag matching of CT-Cache.

TABLE I: The results and cache operations of tag matching in CT-Cache.

	GTT hit	GTT miss
DTA hit	Cache hit	1. Send a read (write) request to the main memory in case of a read (write) miss 2. Allocate a new GTT entry or replace a GTT entry 3. Allocate a subarray for the GTT entry
DTA miss	1. Send a read (write) request to the main memory in case of a read (write) miss 2. Need to allocate a new subarray for the GTT entry or need to replace a cache line	

deallocation by comparing the CT-Cache to the CT-Cache without proactive allocation and deallocation (referred to as 'CT-w/oPAD').

B. Cache Operations

1) Tag Matching: In this subsection, we explain the tag matching procedure of the CT-Cache in more detail. Assuming that we have a 64-bit address space, we can divide the memory address into four parts as shown in Fig. 4. In the conventional cache design, both X and Y parts correspond to the tag bit parts. In our design, we decouple the tag bits into X and Y bits each of which will be used for GTT matching and delta tag matching, respectively. When there is a cache access request, we first search for GTT entries to match X bits (the valid bit of the entry is also checked though it is not shown in Fig. 4). If there is a matched entry (GTT hit), we then access the DTAs with the index bit (Z) for lower tag bit (Y part) matching. During the delta tag matching, the subarray mask bits of the corresponding GTT entry are also accessed. As explained before, the subarray mask bits record the cache subarrays that are currently allocated to each GTT entry. The subarray mask bits inform us which DTAs must be accessed for delta tag comparison. When the accessed GTT entry holds more than one subarray, multiple DTAs must be accessed. In this case, the DTA accesses and comparisons are performed in parallel across the multiple DTAs. A tag match in any DTA is regarded as a cache hit and the data array of the corresponding subarray is accessed.

The results of the CT-Cache tag matching are summarized in Table I. In the case of both GTT hit and DTA hit, it is regarded as a cache hit and the data array is accessed. Since we use write-through caches, we also send a write request to main memory in the case of write hit. For the remaining cases, it is regarded as a cache miss since we cannot find the requested data from the cache. In the case of a cache read (write) miss, we should send a read (write) request to the main memory[6]. In CT-Cache, we classify a cache miss into two different types: GTT miss and GTT hit/DTA miss. In the following subsections, we explain the details of cache operations for GTT miss and GTT hit/DTA miss cases.

2) GTT Misses: In case of GTT misses, there are two possible cases: 1) there is an available GTT entry (i.e., there exist a GTT entry of which valid bit is '0'), and 2) no available GTT entry. In case 1, we can allocate a new GTT entry without a replacement for storing the upper tag pattern which incurred the GTT miss. In order to allocate the lower tag bits and data, we then allocate a new subarray for the newly allocated GTT entry. Please note that there may be a case where there is no available subarray for allocation. In this case, we deallocate the subarray that has the least hit count among 128 subarrays and allocate this subarray to the new GTT entry. In case 2, we need to select a victim GTT entry and replace it with the new one. In order not to evict live cache blocks (i.e., cache blocks that are likely to be accessed in near future), we choose a GTT entry that has the least access counter as a victim (i.e., replacement) GTT entry. Finally, we allocate a new subarray to the new GTT entry and allocate the data in the data array while updating the metadata in the DTA and the GTT entry.

3) GTT hit/DTA Misses: In the case of DTA misses, we first check the following conditions: 1) whether or not the allocation privilege bit of the GTT entry that caused a GTT hit is '1', and 2) whether or not the cache miss is a conflict miss. If both the conditions are satisfied, we allocate an additional subarray for the corresponding GTT entry. After the subarray allocation, the data is also stored in the newly allocated subarray and the corresponding metadata is updated. In case where there is no available subarray for the allocation, we have to select a victim cache line from N victim candidates where N is equal to the number of subarrays currently allocated to the GTT entry. The victim selection is performed in a similar manner to the pseudo-LRU (already explained in Section III-A3). After we select a victim, we replace it with new data in the data array and update the corresponding metadata in the DTA. In the opposite case where either of the two conditions is not satisfied, we also perform a cache line replacement with pseudo-LRU while not allocating an additional subarray for the GTT entry.

IV. Evaluations

A. Evaluation Framework

For evaluation of our proposed CT-Cache, we use gem5 simulator [1] with fastforwarding 2 billion instructions and actually running 1 billion instructions. We model a high-performance processor equipped with four out-of-order cores with per-core L1 data/instruction and L2 caches, a shared 8MB L3 cache, and a shared 256MB L4 cache. We employ the CT-Cache to 256MB eDRAM-based L4 cache as large caches generally suffer from the metadata storage overhead. For rest of the cache hierarchy (i.e., L1, L2, and L3), we use the conventional SRAM-based caches. Table II summarizes the parameters for our simulated processor and system. For energy and array latency evaluation, we use CACTI-P [9] and DESTINY [13] with 22nm technology nodes. We first collect cycle-level latencies of the arrays from CACTI-P and DESTINY and then use these latencies in gem5 simulation tool for performance evaluations. For energy evaluations, we also gather the eDRAM-based L4 cache access statistics from gem5, and calculate dynamic and leakage energy of L4 caches. Since the CT-Cache may increase main memory energy consumption as we use write-through policy, we also evaluate DRAM main memory energy consumption by using DRAMPower [2].

We use eight memory-intensive workloads from SPEC2006 CPU benchmark suite: 429.mcf, 433.milc, 437.leslie3d, 450.soplex, 459.GemsFDTD, 462.libquantum, 470.lbm, and 471.omnetpp. We evaluate single, quad, and mixed workload configurations. In 'single' configuration, we run a single copy of the workload in only one core while we run an identical workload in all four cores in 'quad' configuration. In 'mixed' configuration, we run a mix of four different workloads in the system as summarized in Table III.

In performance and energy results, we show four different schemes for comparisons.

- Baseline: It is a conventional cache that uses non-compressed tag storages.

[6]Please recall that we use write-through caches. In the case of a write miss, we must also update the data to the main memory as well as the CT-Cache.

TABLE II: Simulated processor and system specifications.

Categories	Specification
Processor core	Alpha 21364, ARM ISA
Clock frequency	4GHz
32KB per-core L1 data and instruction cache	1 cycle, 2-way, LRU, write-back
256KB per-core L2	3 cycles, 4-way, LRU, write-back
8MB shared L3	Sequential, 4 cycles for tag access, 7 cycles for data access, 16-way, LRU, write-back
Baseline 256MB shared L4	Sequential, 36 cycles for tag access, 81 cycles for data access, 32-way, LRU, write-back
CT-Cache 256MB shared L4	Sequential, 32 GTT entries, 128 subarrays, 1 cycle for global tag table, 7 cycles for delta tag array, 81 cycles for data access, write-through
Main memory	DDR4 2400, 1 channel, 2 ranks, 16 banks per rank, 64-entry read buffer, 128-entry write buffer

TABLE III: Eight mixed workload groups used for evaluations.

	Workloads
mixed_1	429.mcf, 437.leslie3d, 450.soplex, 471.omnetpp
mixed_2	433.milc, 459.GemsFDTD, 462.libquantum, 470.lbm
mixed_3	429.mcf, 433.milc, 437.leslie3d, 450.soplex
mixed_4	459.GemsFDTD, 462.libquantum, 470.lbm, 471.omnetpp
mixed_5	429.mcf, 450.soplex, 459.GemsFDTD, 470.lbm
mixed_6	433.milc, 437.leslie3d, 462.libquantum, 471.omnetpp
mixed_7	429.mcf, 433.milc, 459.GemsFDTD, 462.libquantum
mixed_8	437.leslie3d, 450.soplex, 470.lbm, 471.omnetpp

- CT-Static: It uses a GTT that statically maps four subarrays to each GTT entry.
- CT-w/oPAD: It is CT-Cache without the proactive subarray allocation and deallocation. In other words, the subarray allocation and deallocation only happens when a GTT miss or delta miss occurs (i.e., reactive allocation and deallocation).
- CT-Cache: It is our proposed CT-Cache with proactive subarray allocation and deallocation as explained in Section III.

B. Performance

Fig. 5 shows the performance results (weighted speedup [3] for ‘quad’ and ‘mixed’) normalized to the baseline. For single, quad, and mixed cases, our proposed CT-Cache shows performance improvements of 16.2%, 16.2%, and 4.7%, respectively, over the baseline. Compared to the single and quad cases, the mixed configurations show relatively less performance improvement. This is because the mixed workloads have higher possibility to have different patterns in upper address bits. Since we have a limited number of GTT entries, the diverse upper address patterns will lead to more GTT misses. However, the CT-Cache still shows performance improvements even in the case of the mixed workloads mainly due to the reduced latency of tag accesses.

Compared to the CT-Static and CT-w/oPAD, the CT-Cache shows better performance in case of multi-programmed workloads. In case of mixed (quad) workloads, CT-Cache shows better performance by 30.4% (33.3%) and 8.1% (5.8%) on average, compared to CT-Static and CT-w/oPAD, respectively. On the other hand, in case of single workloads, though CT-Cache exhibits performance improvement of 16.2% as compared to the baseline, CT-w/oPAD shows the best performance among four different schemes (21.8% performance improvement over the baseline, on average). This is because a single workload hardly uses all of the GTT entries. It also means the reactive subarray management would be sufficient for the workloads that have relatively small working sets. However, due to the rigidity of the subarray allocation and deallocation, the CT-Static cannot distribute the subarrays to the GTT entries depending on their capacity requirements. This rigidity leads to higher cache miss rates, which eventually results in a huge performance loss compared to the baseline (19.7% in the case of the mixed workloads). Overall, the CT-Cache shows stable performance improvements across single, quad, and mixed workloads as compared to the CT-Static and CT-w/oPAD.

C. Energy

Fig. 6 depicts the LLC and main memory energy consumption results for CT-Static, CT-w/oPAD, and CT-Cache normalized to the baseline. The CT-Cache shows unanimous energy reductions in all of the cases shown in Fig. 6 while the CT-Static and CT-w/oPAD show energy consumption more than the baseline in some cases (bars higher than 1.0 in Fig. 6). The CT-Cache reduces LLC and main memory energy consumption by 28.3%, 29.7%, and 20.0%, on average, compared to the baseline for single, quad, and mixed workloads, respectively. The main reasons of energy reduction can be summarized in two-folds: 1) the reduced tag storage consumes much less dynamic and leakage energy, and 2) reduced execution time also reduces leakage (standby) and refresh energy of LLC and main memory because the processor and main memory can be transitioned into low-power modes (e.g., power gating) earlier.

Compared to CT-Static and CT-w/oPAD, the CT-Cache shows more consistent energy reductions from the baseline across single, quad, and mixed workloads. Considering that CT-Static and CT-w/oPAD also use the reduced tag storages, the differences in energy consumptions across the four schemes are mainly attributed to the differences in the execution time (i.e., performance). Comparing the energy results (Fig. 6) with the performance results (Fig. 5), there is a strong reverse correlation. In other words, the higher performance a scheme shows, the lower energy consumption the scheme tends to exhibit.

D. Area

TABLE IV: Tag storage comparison for 256MB LLC.

	Baseline	CT-Cache
Tag storage	20.5MB	2.5MB
Global Tag Table	N/A	800B
Hit counters and SA_valid bit for CT-cache	N/A	272B
Total	20.5MB	2.501MB

Table IV summarizes tag storage comparison between the baseline and CT-Cache. We omit the metadata (e.g., valid bit, coherence status bit, etc.) for each cache line in this comparison as the metadata are included in both baseline and CT-Cache. Please note that this is a conservative comparison as the CT-Cache only maintains 2-bits for replacement policy (5-bits are required for the baseline which employs LRU policy) and does not also require a dirty bit[7]. For the baseline, we need 41-bits tag for each cache line when using 64-bit address space. Since the baseline cache has 4M cache lines, we need a total of 20.5MB for tag storage. In the case of CT-Cache, we need 5-bits delta tag storage for each cache line, leading to a total of 2.5MB delta tag storage for 256MB LLC. We also require 200-bits (25-Bytes) for each entry of the GTT storage, resulting in total 800-Bytes for the 32-entry GTT. Furthermore, the CT-Cache maintains the hit counter and SA_valid bit for each subarray, which corresponds to 272-Bytes (17*128-bits). In total, the CT-Cache reduces the tag storage by 87.8% compared to the baseline. Due to the reduced tag storages, one can obtain latency and energy reduction for tag access, which eventually results in better performance and energy efficiency as shown in Sections IV-B and IV-C.

V. Conclusion

In this paper, we have proposed CT-Cache—a cache design based on tag compression. Our proposed CT-Cache decouples

[7]Since CT-Cache uses write-through policy, we do not need to maintain a dirty bit in each cache line.

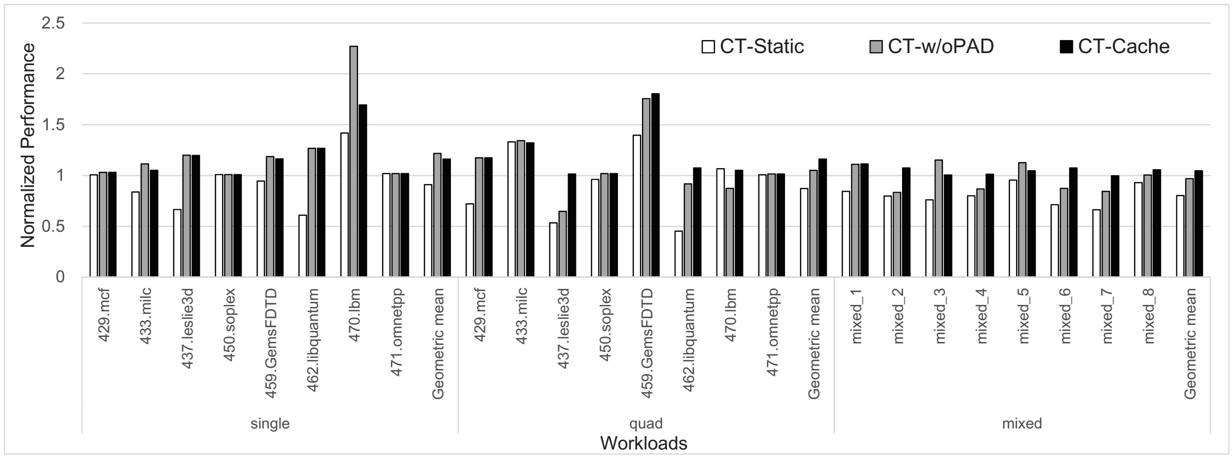

Fig. 5: Performance (weighted speedup) results of CT-Static, CT-w/oPAD, and CT-Cache normalized to the baseline.

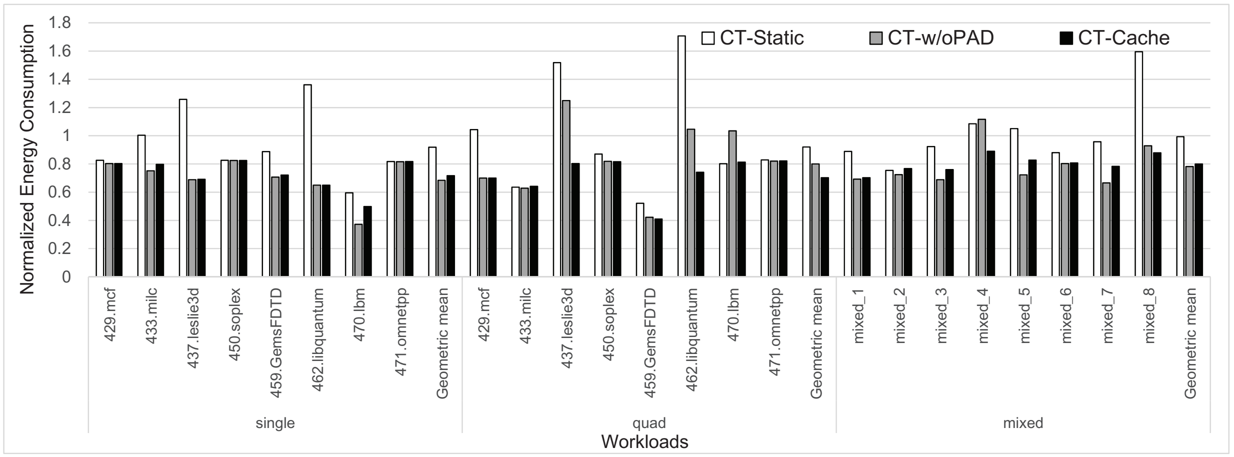

Fig. 6: Energy consumption results of CT-Static, CT-w/oPAD, and CT-Cache normalized to the baseline.

tag storages into the global tag table and delta tag arrays. By sharing the upper-bit tag storage, we can significantly reduce tag storage required for large last-level caches. Our proposed CT-Cache shows performance improvement and energy reduction of 4.7%∼16.2% and 20.0%∼29.7%, respectively, compared to the baseline (conventional caches that use non-compressed tag storage). The evaluation results verify that the CT-Cache shows stable and consistent performance improvements and energy reductions over the baseline as compared to the CT-Static and CT-w/oPAD (the two variants of our proposed CT-Cache). As our future work, we plan to (i) investigate the impact of full system environment including the operating systems on our proposed CT-Cache, (ii) perform sensitivity studies on various cache configurations (e.g., cache hierarchy and size, etc.), (iii) evaluate our proposed CT-Cache technique using write-back caches and provide performance and energy consumption comparison with the currently used write-through caches, (iv) devise a technique to reduce memory bandwidth requirements due to the write-through policy, and (v) develop formal methods to determine the design parameters in the CT-Cache.

Acknowledgment

Joonho Kong is the corresponding author of this paper. 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 (2015R1C1A1A01051836) and Samsung Electronics. We would also like to thank anonymous reviewers for their constructive comments.

References

[1] N. Binkert et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2), 2011.

[2] K. Chandrasekar et al. DRAMPower: Open-source DRAM Power & Energy Estimation Tool.

[3] S. Eyerman and L. Eeckhout. Restating the case for weighted-IPC metrics to evaluate multiprogram workload performance. *IEEE Computer Architecture Letters*, 13(2):93–96, 2014.

[4] C.-C. Huang and V. Nagarajan. ATCache: reducing DRAM cache latency via a small SRAM tag cache. In *PACT*, pages 51–60, 2014.

[5] D. Jevdjic et al. Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 404–415, 2013.

[6] J. Kim et al. TLB index-based tagging for reducing data cache and TLB energy consumption. *IEEE Transactions on Computers*, 66(7):1200–1211, 2017.

[7] J. W. Kwak and Y. T. Jeon. Compressed tag architecture for low-power embedded cache systems. *Journal of Systems Architecture - Embedded Systems Design*, 56(9):419–428, 2010.

[8] J. Lee et al. TLB index-based tagging for cache energy reduction. In *ISLPED*, pages 85–90, 2011.

[9] S. Li et al. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *ICCAD*, pages 694–701, 2011.

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

[11] G. Pekhimenko et al. Base-delta-immediate compression: practical data compression for on-chip caches. In *PACT*, pages 377–388, 2012.

[12] P. Petrov and A. Orailoglu. Tag compression for low power in dynamically customizable embedded processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(7):1031–1047, 2004.

[13] M. Poremba et al. DESTINY: A tool for modeling emerging 3D NVM and eDRAM caches. In *DATE*, pages 1543–1546, 2015.

[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*, pages 235–246, 2012.

[15] K.-H. Yang et al. eTag: Tag-comparison in memory to achieve direct data access based on eDRAM to improve energy efficiency of DRAM cache. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(4):858–868, 2017.

[16] V. Young et al. DICE: Compressing DRAM caches for bandwidth and capacity. In *ISCA*, pages 627–638, 2017.