Tag Tables

Sean Franey & Mikko Lipasti
University of Wisconsin - Madison
sfraney@wisc.edu, mikko@engr.wisc.edu

Abstract

Tag Tables enable storage of tags for very large set-associative caches - such as those afforded by 3D DRAM integration - with fine-grained block sizes (e.g. 64B) with low enough overhead to be feasibly implemented on the processor die in SRAM. This approach differs from previous proposals utilizing small block sizes which have assumed that on-chip tag arrays for DRAM caches are too expensive and have consequently stored them with the data in the DRAM itself. Tag Tables are able to avoid the costly overhead of traditional tag arrays by exploiting the natural spatial locality of applications to track the location of data in the cache via a compact "base-plus-offset" encoding. Further, Tag Tables leverage the on-demand nature of a forward page table structure to only allocate storage for those entries that correspond to data currently present in the cache, as opposed to the static cost imposed by a traditional tag array. Through high associativity, we show that Tag Tables provide an average performance improvement of more than 10% over the prior state-of-the-art - Alloy Cache - 44% more than the Loh-Hill Cache due to fast on-chip lookups, and 58% over a no-L4 system through a range of multithreaded and multiprogrammed workloads with high L3 miss rates.

1. Introduction

Recent technological advances incorporating dense DRAM "close to" the processing cores of a computing system - namely embedded DRAM (eDRAM) and stacked DRAM - provide unprecedented low-latency, high-bandwidth data storage. While the capacities are large given their proximity to the core, they are unlikely to provide sufficient storage to replace main memory for all but certain embedded applications [5]. As such, recent products [2, 8, 12] and research proposals [9, 10, 14] have advocated their use as additional levels of cache. Unfortunately, traditional tag tracking mechanisms are not well-suited to these large capacities for most applications.

First of all, the small-allocation-unit cache represents the mechanism of choice for most of the history of hardware caches. While very efficient at utilizing cache capacity by storing relatively small blocks of data, it is prohibitively expensive to store the necessary number of unique tags for these high-capacity caches which are expected to reach between hundreds of megabytes to tens of gigabytes in the near future. For example, considering a 1GB cache with 64B blocks and 6B of tag per block, these traditional tag arrays require 96MB of storage - infeasible on the SRAM-based logic die. As such, recent small-block-based approaches have proposed storing the tags in the DRAM array themselves [10, 14], necessitating novel techniques for addressing the fundamental issue of high-latency DRAM accesses for tag checks. Two prior tags-in-DRAM approaches for these large caches, the Loh-Hill Cache [10] and the Alloy Cache [14] both attempt to mitigate this issue by completely avoiding DRAM cache accesses on misses, either by an additional tracking structure on the logic die as in the Loh-Hill Cache's MissMap or through prediction as in the Alloy Cache. The Alloy Cache further addresses DRAM access latency by also optimizing hit latency. While the Loh-Hill Cache preserves a high level of associativity for its stacked DRAM cache, requiring the acquisition of multiple DRAM blocks for all the tags in a set, the Alloy Cache advocates a direct-mapped approach, requiring the acquisition of only an additional burst of data across the bus to acquire the co-located tag information for a data block. Therefore, these two competing proposals take different stances on the importance of hit rate and hit latency, the importance of which is largely a function of application characteristics.

On the other end of the tag tracking spectrum, one that optimizes for tag storage and can thus avoid the fundamental issue of accessing DRAM for tags, is the large-allocation-unit cache. Whether simply a tag array with large blocks or a sectored approach allowing small block fetch within the large allocation units (the sector), these mechanisms achieve much lower tag storage overhead by reducing the number of tags, potentially allowing them to exist in fast SRAM on the logic die. The drawback for these approaches however, stems from much higher miss rates from false conflicts created by these large allocations. While simply increasing the block size is generally regarded as a poor design point due to the high bandwidth demand (a whole large block must be fetched on every miss), sectored cache approaches have recently been advocated as a viable solution to tag tracking for large DRAM-based caches. The recent Footprint Cache proposal utilizes a sectored tag array augmented with a footprint predictor that predicts the subset of sectors that are likely to be accessed, and populates the sectored cache entry by prefetching only the predicted sub-blocks [9]. This approach still suffers from conflict misses due to the large block size, but has relatively low overhead for tag storage (< 2MB for a 256MB cache, but growing to 8MB for a 1GB DRAM cache) and, subject to an accurate footprint predictor, is able to mitigate the increase in conflict misses with useful prefetches. Our evaluation in Section 8.6 shows that footprint-based prefetching can be

easily adapted to our proposal for tag tables, and has the potential to provide significant performance gains.

As a mechanism to provide many of the key features of the prior techniques, this paper presents Tag Tables. Tag Tables provide low latency tag check facilities on-par with the Alloy Cache while maintaining the associativity - and thus hit rate - of the Loh-Hill cache. Further, Tag Tables provide a small storage overhead for tags in line with previously proposed sectored-like approaches such as the Footprint Cache, allowing the tags to be stored on the fast SRAM-based logic die while maintaining traditional small block sizes (64B as evaluated). Tag Tables achieve these seemingly contradictory goals by leveraging the naturally-present spatial locality of application working sets that is enhanced by the longer residency times of data in these high-capacity DRAM caches by utilizing a base-plus-offset encoding of "chunks" of contiguous data in the cache. Further, Tag Tables are implemented as a forward page table stored in the existing L3 cache resources of the chip, allowing tag storage overhead to be dynamically balanced with on-chip caching resources. The remainder of the paper will elaborate on the specific mechanisms that achieve these features.

2. Motivation

As a means to place these competing ideologies into perspective, Figure 1 presents the "Bandwidth-Delay product" (BDP) achieved versus the tag storage overhead of a broad range of DRAM cache tag storage configurations. The compound BDP metric attempts to quantify the major benefits sought with caching: reduced off-chip bandwidth and improved application performance through high speed data access, where a lower value is better. The shaded area at the bottom left of the figure captures designs that have lower BDP than a baseline system without a DRAM cache, and have a reasonable (defined here as 6MB) SRAM cost for tags.

Along with the Loh-Hill Cache (L-H), the Alloy Cache (A), and Tag Table (TT) configurations, a family of curves is presented in the figure for a large-allocation-unit mechanism that attempts to reduce tag overhead by increasing block size. The graph clearly shows that the large allocation approach - which evaluates block sizes from 64B to 512B - fails to achieve good performance, either consuming too much off-chip bandwidth to fetch its large blocks and/or incurring too many misses due to false conflicts. In contrast, the Loh-Hill Cache, the Alloy Cache, and Tag Tables all reside within the desirable portion of the graph with the Tag Table configuration providing the best BDP (Section 8.4 will discuss the main features that distinguish Tag Tables from both the Alloy Cache and Loh-Hill Cache).

Further, while the Tag Table configuration exists to the right of the Alloy Cache, indicating increased SRAM storage requirement, this is the complete tag storage overhead (i.e., no tag information needs to consume DRAM cache capacity). In addition, the Tag Table achieves its performance without any

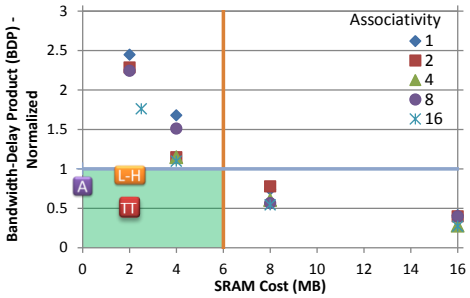

Figure 1: ***Performance of various cache configurations for a 256 MB L4 cache. Shaded area indicates desirable region of improved bandwidth-delay with practical SRAM cost. Results shown for the PARSEC canneal benchmark normalized to a baseline with no L4.***

prediction or SRAM-based hit/miss determination structures as the other three state-of-the-art techniques rely on. In fact prediction is an orthogonal technique that is equally applicable to Tag Tables thus they can be extended by any one of a number of prediction schemes to achieve even better performance (Section 8.6 will investigate the opportunity footprint prediction might have for Tag Tables).

2.1. Additional Related Work

The structure of Tag Tables are partly inspired by prior work in memory protection and address translation. Mondrian Memory Protection (MMP) argues for fine-grained protection domains and proposes a practical implementation scheme that exploits spatial locality similar to that exhibited by application cache accesses and exploited by our Tag Table. MMP exploits these observations through both a forward page table structure and base-plus-offset encoding of entries [19]. In a similar vein, Coalesced Large-Reach TLBs (CoLT) uses a base-plus-offset encoding to track many contiguous virtual-to-physical page mappings [13].

3. Page Walk

This section presents the fundamental data structure of Tag Tables, which is based on the tree structure used to implement a forward page table.

3.1. Page Tables

Motivated by Figure 1, we begin to develop a more compact storage mechanism for large cache tags by arguing that any extension of a traditional tag array will be unable to exploit runtime features of the system to achieve storage savings because they impose a static storage requirement. Regardless of the state of the cache, the one-to-one mapping of cache blocks to tags in a traditional tag array means that it must provision storage equal to the product of the number of blocks in the cache and the storage required per tag. This feature is analogous to another mechanism for tracking memory metadata - the inverted page table (IPT). Similar to the tag array, the IPT has a one-to-one mapping of physical pages to entries and imposes a static storage requirement proportional to the number of physical pages. Unlike a traditional tag array however, a

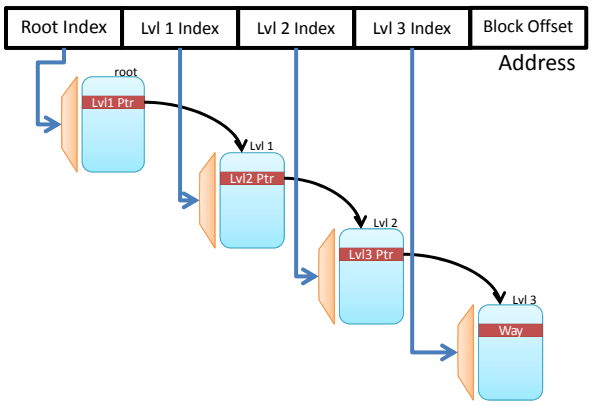

Figure 2: ***High level operation of a forward-page-table-based implementation of cache tags.***

page table has an alternate implementation to the IPT - the forward page table (FPT) - that only requires storage relative to the number of *mapped* pages. In other words, no storage overhead is incurred for a memory page that is unmapped in an FPT. Therefore, as a starting point for developing a more robust solution to tag storage for large caches, we investigate opportunities for adapting the FPT mechanism which provides us with the flexibility to adapt to application behavior and frees us from an inflexible requirement that cannot be adapted on-demand to system needs.

At a high level, the replacement of a traditional tag array by a page table implementation is straightforward and follows the operation outlined by Figure 2. Upon access, a tag check occurs as a virtual address translation does in a forward page table through indexing the various levels of the table with appropriate bits of the access's address. The "walk" terminates when either all bits are exhausted and a leaf entry is found - storing the way associated with the tag, if the cache is associative - or the branch terminates prematurely indicating a miss. In the end, all bits of the address above the block offset have been used to arrive to a unique location in the table for the address. This operation is analogous to the page table where all of the bits of the address above the *page* offset are used to traverse to a unique leaf. A miss would then trigger insertion of the data in the cache and tracking of the tag in the table by extending the branch to a leaf and storing the appropriate way in the leaf as identified by an eviction mechanism which is unchanged from a traditional cache (e.g., selection of the least recently used member of the set). In and of itself, the choice of this data structure in Tag Tables is not a fundamental improvement as the combination of multiple levels and way storage in the leaf is not inherently less expensive or faster than a traditional tag structure. It does however, provide a starting point that frees Tag Tables from inflexible storage requirements and represents a structure that is able to adapt to system operating characteristics.

3.2. Locating Metadata

As with any tag tracking structure - especially one for high-capacity caches - the storage location of the metadata is a primary concern. By metadata, we are referring to the data that is required in order to logically create the Tag Table: the leaf entries storing way information and the pointers to subsequent levels. In a traditional tag array, this metadata is the tags themselves. While the Loh-Hill Cache and Alloy Cache choose to store their metadata (cache tags) in the stacked DRAM array itself, Tag Tables adopt the page table approach and store its metadata in memory, specifically the L3 cache. While it would be possible to dedicate a structure by "carving" out a portion of the L3 cache as is proposed for the "MissMap" access predictor in the Loh-Hill cache [10], locating Tag Table metadata dynamically in the L3 allows a system with Tag Tables to quickly perform table walks and balance metadata with application data through the existing cache replacement policy.

This competition of metadata with application data is allowed without restriction up to a "high watermark." This high watermark is enforced on a per-set basis and is necessary to limit the number of ways in the set that can be occupied at any given time with metadata. This restriction is placed to prevent excessive pollution by metadata that could otherwise severely harm application performance. This is particularly a concern in the situation where the L3 cache is capable of storing the major portions - if not all - of the application's working set (a situation not that uncommon for typical L3 cache sizes and many applications). Since metadata effectively reduces the capacity of the L3 for application data, such pollution could result in the working set no longer fitting, severely impacting performance. Section 8 will evaluate the effect of different high watermark settings on Tag Table performance.

4. Reducing Levels Traversed

Although a traditional page walk can work correctly to perform a tag check as described, this section describes two opportunities afforded Tag Tables to accelerate these walks. These opportunities arise from the different needs of a tag check relative to an address translation.

4.1. Walk Bit Selection

First of all, unlike a traditional page table, Tag Tables can reverse the order of the bits used to index various levels in order to utilize the high entropy bits to index the upper level entries. This is useful since Tag Tables, like traditional tag arrays, would like to use these high entropy, low-order bits to define the set associated with the data to more evenly utilize cache capacity (as labeled in Figure 3 as "Row Selection Bits"). While utilizing high order bits for upper levels of a page table is advantageous for translating virtual to physical addresses by allowing page size to be implied by a translation's location in the table, Tag Tables do not benefit from such support and

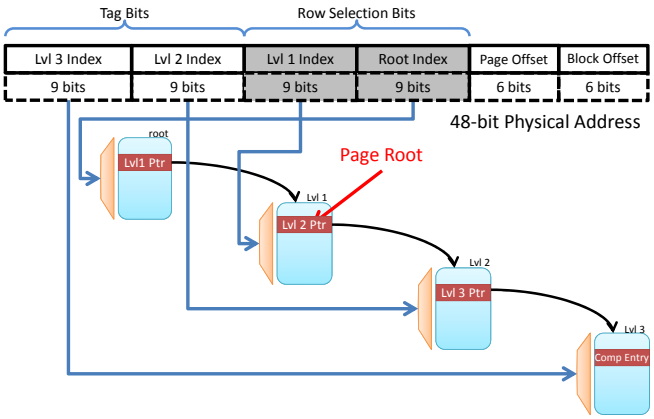

Figure 3: ***Modified high level operation, specific to a 1GB DRAM cache with 4KB rows and 64B blocks, accessed by 48-bit addresses.***

indeed, can gain significant advantage by reversing the order of the bits selected. As shown in Figure 3, this reversed selection indexes the root level with the bits of the address just above the page offset (Section 5 will discuss how the page offset bits are used in the tag check).

One important implication of this reverse walk is the ability to infer data location based on the path traversed through the tree. As Figure 3 shows, by evenly dividing the number of bits required to uniquely identify a row - $log_2(Number\ of\ Rows)$ - across some number of upper levels of the table, we can completely isolate all of the entries associated with a given row of the cache to a specific subtree of the Tag Table. We call the roots of these subtrees "page roots," corresponding to the pointers in the first level of the table (Lvl 1) in Figure 3, and are uniquely identified through the two sections of the address highlighted as the "Row Selection Bits" in the figure. Given that all the blocks within a main memory row map to the same row in the DRAM cache as discussed above, these bits are analogous to the set selection bits of a traditional tag array where each DRAM cache row is equivalent to a set. As we will discuss in the next section, this analogy extends to placement in the DRAM cache in that Tag Tables allow blocks to be placed flexibly anywhere within their appropriate DRAM cache row.

4.2. Upper Level Entries

The second mechanism Tag Tables employs to to reduce the average number of levels traversed is the allowance of flexible placement of leaf entries. Instead of requiring a tag check to traverse all levels of the tree, if there is no ambiguity, leaf entries can be placed in any other intermediate node on the correct path. Such an ambiguity guarantee is encountered whenever a traversal terminates on an incomplete path prior to reaching a leaf node. In such a situation, it is guaranteed that no other data exists in the cache that takes the same path through the tree, so there is no need to extend the current branch to a leaf, instead it is acceptable to create a leaf entry, encoding the way at the current level for the new data and maintaining a tag in the entry to disambiguate any subsequent accesses.

As an example, consider the first accesses to a cache upon initialization. Since no entries currently exist in the table, the first access will miss in the root. Therefore, upon insertion there is no need to fully traverse to a leaf, instead the leaf can be created at the associated root index. This in turn, necessitates the tracking of tag data to allow subsequent accesses that index the same entry to disambiguate themselves with the existing entry. This means that bits of the address that must be maintained as a tag correspond to those portions of the address that have not yet been used to traverse the table. In the case of a collision at the root, this corresponds to the three fields of the address associated with the L1, L2, and L3 indices as shown in Figure 3.

5. Compressed Entries

5.1. Sectored Caches

While a page table structure is useful for allowing Tag Tables to dynamically adapt to system operating characteristics, the structure does not inherently reduce the tag storage required. Therefore, as a first step toward realizing compressed tag storage and making storage of metadata in the L3 feasible, this section investigates sectored caches, a tag tracking mechanism for some of the earliest cache implementations. Sectored caches are a simple mechanism that rely on spatial locality of data in the cache to effectively reduce the overhead of storing tags for large contiguous regions of data by storing only one full tag for a region - a sector - of the cache along with a bitvector indicating which of the blocks represented by the tag are actually present [1].

As mentioned previously, such an approach is particularly beneficial for applications that exhibit a very high degree of spatial locality or "page access density" [9]. As long as a very high proportion of the blocks covered by a tag are actually present, the cache achieves most of the performance available with more fine-grained tags with much less overhead. The problem arises when there is not enough locality to densely populate the region covered by a tag, leading to a much higher rate of evictions when blocks from conflicting tags are referenced. Indeed, most cache designs in the decades since sectored caches were first introduced have eschewed their use because many applications do not exhibit the density necessary to realize their benefit.

In order to investigate the feasibility of sectored cache designs for large DRAM caches for systems running modern applications, we measure the number of unique tags in a given DRAM cache row as a proxy for the locality available for a sectored cache's large tags. We determine the number of unique tags through simulation of multi-threaded PARSEC and multi-programmed SPEC workloads on a direct-mapped Alloy Cache [14], capturing a snapshot of the unique tags present in a row at the end of simulation (details of the simula-

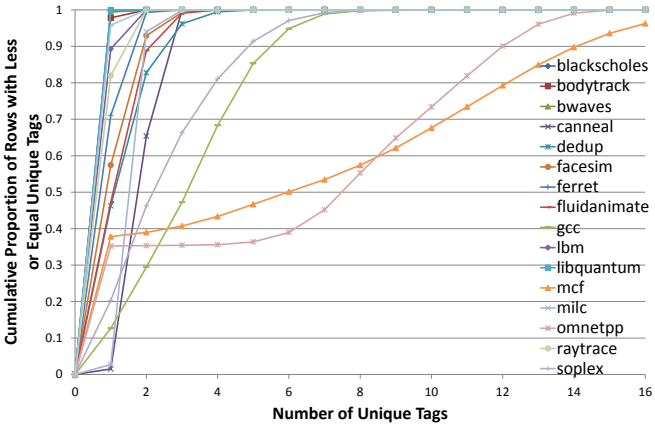

Figure 4: ***Cumulative proportion of DRAM cache rows with less than or equal to a given number of unique tags.***

tion parameters are given in Section 8). Presented in Figure 4, this evaluation indicates that modern applications exhibit a wide range of locality characteristics. The figure relates the cumulative proportion of the cache's rows that exhibit a given x-value's number of unique tags. For instance, *mcf* has nearly 40% of its cache's rows with more than 8 unique tags (60% of the rows have less than or equal to 8 tags). The page size investigated is 4 KB and it can be seen that for many applications, most rows in the cache have very few unique tags as evidenced by the sharp slope in the graph for low values of unique tags, identifying them as potentially good candidates for a sectored cache. A significant number of other applications however, such as *mcf* and *omnetpp*, unfortunately exhibit few rows with high locality as evidenced by their graph's relatively flat growth at low unique tag counts. Finally, despite indications that perhaps many applications may benefit from a sectored tag approach, i.e. those with steep slopes to the left end of the graph, it should be noted that for a 256MB cache, 64B blocks and 256B per sector (i.e., 4 blocks per sector) the storage overhead of the tag array is still 5MB. This penalty grows near-linearly with cache size (neglecting the minor change in tag size per sector due to larger caches) resulting in much higher cost as cache size increases. Together, this data justifies the absence of sectored cache designs in current cache implementations due to its severe penalty of several important application types and motivates us to identify a more robust mechanism for reducing the storage requirement of cache tags.

Motivated by the results of Figure 4 which indicate a range of spatial locality within a cache both over rows within the cache and from application to application, we seek to create a space-efficient entry type that can robustly adapt to these different scenarios. While the figure shows that the rigidly imposed spatial locality required by a sectored cache does not map well to many applications, it also indicates that there is significant opportunity if a sectored-tag-like approach were available for some applications and regions of the cache. In order to realize such a robust tracking mechanism, we propose the use of a "base-plus-offset" encoding for tracking regions of memory in the cache. Rather than imposing static region

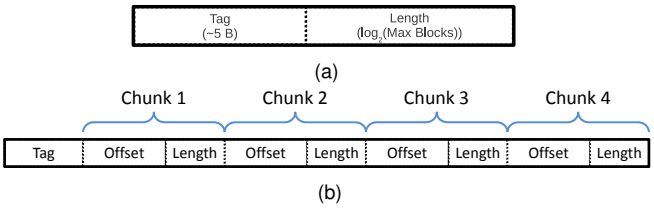

Figure 5: ***Basic & Expanded Entry Formats.***

sizes and boundaries and tracking the blocks within through the sectored approach's bitvector, we propose instead storing the tag associated with the bottom block of a "chunk" of data and indicating the number of contiguous blocks from that base block with a count value[1]. Such an encoding can be seen in Figure 5a, showing a field for a simple tag followed by a field that indicates the number of contiguous blocks existing beyond that base.

Unlike a bitvector which requires a bit for every block, this length indication only requires $log_2(Max.\ no.\ blocks)$ bits. The tradeoff of course, is that this encoding cannot track a range with "holes" (i.e., non-present blocks within the chunk) in it. To address this issue, we propose a hybrid approach by introducing some number of chunk representations greater than one as shown in Figure 5b, allowing us to represent holes implicitly in the gaps between chunks. In this format, in addition to the tag and length previously discussed, we also must utilize an "offset" specification that relates the first block of the chunk to the tag of the entry. For example, an "offset" of '4' indicates that the first block of the chunk exists four blocks beyond the base block identified by the tag.

When combined with the forward page table structure proposed in Section 3.1, by replacing the simple way-identifying entries proposed at the leaf of the page table with these compressed entries, we free ourselves from a one-to-one mapping of blocks present in the cache to leaf entries in the page table structure. This allows us to amortize our inherently larger entries over a greater range of blocks. Section 8 will quantify this amortization by showing that the average number of blocks tracked per entry is actually quite high for those applications in our benchmark set.

6. Operation

This section provides discussion of the operation of Tag Tables, utilizing all of the components so far described. The basic Tag Table structure tracks tags in a large cache and operates - at a high level - as a forward page table with tag checks implicit with a walk of the table. The exact location of data is stored in compressed form at leaf entries with base-plus-offset encoding and misses are either implied by failure to find a complete path to a leaf or by a block not residing in any of the leaf entry's chunks.

[1] Specifically, for the remainder of the paper, when we refer to a "chunk," we mean a set of blocks present in the cache that are contiguous with one another in main memory.

6.1. Insertion

As with any cache, insertion of data is a fundamental operational issue for Tag Tables. Due to the structure of the compressed entries, Tag Table efficiency is significantly impacted by the location of data. If placed intelligently, large contiguous chunks can be created, amortizing the cost of an entry as additional blocks can be tracked by merely incrementing the "length." If placed poorly, this opportunity is missed and the inherently larger entries relative to a traditional tag can result in a structure that is much larger than even a traditional tag array.

Procedurally, insertion involves first building a bitvector representing all of the blocks present in the current block's DRAM cache row. Thanks to the page roots discussed in Section 4.1, building this bitvector is isolated to the leaves associated with the current block's page root where traversal to each leaf and setting of all bits in the vector associated with the blocks tracked in the leaf's chunks is sufficient to populate the bitvector. While this may seem like a costly event, the optimization presented in Section 4.2 that allows leaf entries to exist in intermediate levels can reduce this cost when it is known that there is only one entry associated with the DRAM cache row. In such a case, building the bitvector only requires inspecting the chunks of this single entry as opposed to potentially multiple traversals. Indeed, as will be shown in Section 8, relatively few entries are frequently necessary per DRAM cache row, limiting the traversals needed to create the bitvector.

Following population of the bitvector, the Tag Table attempts to find a location for insertion by emphasizing extension of existing chunks if such a chunk exists (i.e., either the base or top of the chunk is contiguous with the inserted block) and the bitvector indicates the appropriate position is free in the row. If an existing chunk cannot be identified to extend, the Tag Table falls back to either randomly choosing an existing empty location if one exists, or randomly selecting a victim from the existing blocks. While there are potentially many improvements that can be made to this mechanism (more sophisticated replacement algorithms, etc.), simulation has shown this simple approach works reasonably well, thus we leave investigation into more sophisticated approaches for future work.

Further, while insertion can be a high latency event, particularly in the situation where a page root has many leaf entries, it occurs off of the critical path (i.e., data from the request is passed on the L3 in parallel with the DRAM cache insertion) and follows the already high latency miss event that triggered the fill, thus its impact on performance is negligible. In addition, while not evaluated for this proposal, it can be envisioned - for increased storage cost - that each page root entry can maintain its own persistent bitvector, simply updating it on insertions and evictions, removing the need to dynamically re-build it from scratch on each insertion.

6.2. Associativity

Prior work on DRAM cache tags have taken varying approaches on the topic of associativity. Loh and Hill for one, choose to allow associativity up to the number of blocks in a DRAM cache row, less those blocks required to store tag information (three blocks as presented) [10]. This is a key design point for their proposal as they rely on the guaranteed page-open state the tag check provides them to limit hit latency. Since the row is already open, there is no advantage in their design to limit the associativity, and indeed there is benefit to maintaining associativity, even in such large caches, as Figure 1 highlights. The Alloy Cache on the other hand, explicitly eliminates associativity to optimize hit latency. Since the relative cost of accessing the tags necessary for supporting associativity is so high in the Loh-Hill cache, the Alloy cache argues there is substantial benefit to be realized by limiting the number of tag checks. Given our Tag Table's location on-chip in SRAM however, it suffers from a tag access penalty much more in-line with other on-chip caches which have determined that the cost of associativity is justified (an observation substantiated by Figure 1).

Therefore, for the insertion of data in a Tag-Table-administered cache, associativity similar to the Loh-Hill cache is maintained in that placement of data is valid anywhere within a DRAM cache row. While there is no benefit of a guaranteed open row access, there is essentially no advantage, latency-wise, by restricting data to any particular location within the row. Instead, high associativity is particularly important in an inclusive cache system beyond the inherent hit rate benefits of associativity which will be discussed in more detail in Section 8.4.

In order to support this associativity, Tag Table compressed entries are provisioned to be able to fully track the blocks in a DRAM cache row. This means, that for 4KB pages and 64B blocks, that each offset and length field must be 6 bits wide. This leads to the updated entry format presented in Figure 6 whose new fields will be described fully in the summary of this section (Section 6.3), but notably convey the 6-bit offset and length fields and introduce the "Row Offset" field which relates the page offset of the base of the chunk to the actual location in the DRAM cache row for that block. For example, consider the situation where a block at page offset of 0x4 (relative to the base tag of the entry) is placed at location 0x8 in the actual DRAM row (i.e., 8 blocks from the base of the cache row). In this case, the chunk associated with the page offset will have the value '0x4' in the "Page Offset" field (assuming it is the base of the chunk) and the value '0x8' in the "Row Offset" field. This way, when a subsequent walk traverses to this entry with a page offset of 0x4, it will know that its data is physically located at block 0x8 of the appropriate DRAM cache row. Note, that if the page offset was 0x5 for an access, assuming it was present in the cache (implying a "length" of the chunk >1), the row offset returned

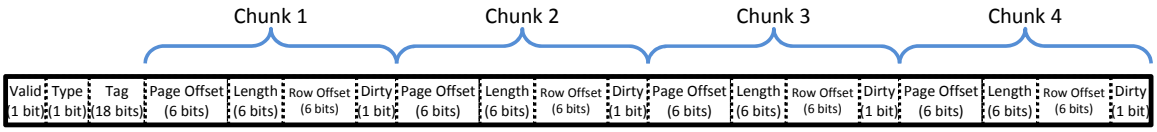

Figure 6: ***Detailed diagram of a compressed entry capable of tracking 4 contiguous chunks of data associated with a tag. Bits correspond to 1GB cache with 4KB rows and 64B blocks associated with a 4-level Tag Table that utilizes 9 bits to index each level.***

from the query would be 0x9.

6.3. Summary and Example of Operation

In common operation, the Tag Table as proposed is referenced upon every cache access as is done with a traditional tag array. Similar to traditional tag arrays, this reference can be done serially or in parallel with data access. Upon access, the Tag Table is traversed as a forward page table with pre-defined portions of the address used to index the various levels of the table as shown in Figure 3. If the traversal encounters an invalid entry at any point, the access is a miss. A miss can also be indicated if the traversal does not hit at the valid entry that matches the access's tag.

Functionally, hit determination operates first through the use of a 6-bit adder which computes the upper value of a chunk. Next, two comparators check if the access's block is greater than the base of the chunk and less than the upper block. Finally, the output of the comparators are fed to a 2-input AND gate that indicates a hit on the chunk when both comparator's operations are true. Similar to tag checks in traditional arrays, multiple chunks can be checked in parallel by trading off area and power.

Upon a hit, the Tag Table returns the row offset associated with the access by subtracting the access's page offset from the page offset field of the chunk and adding it to the row offset field. This row offset is analogous to the way specification in the simplified operation presented in Section 3.1 and identifies the specific location of the data in the row. Finally, by concatenating this row offset with the access's "Row Selection Bits," the exact location of the data in the DRAM cache is determined and associativity to the degree of the number of blocks in a cache row (e.g., 64-way associativity for a cache with 4KB rows and 64B blocks) is realized. Given the many tens to hundreds of core cycles already expended on the given access, the few additional cycles for these adder and comparator operations do not contribute significantly to performance as shown in Section 8 that takes these into account. This direct data access is in contrast to tags-in-DRAM approaches that require the DRAM itself to be accessed before being able to know unambiguously whether and/or where the data is in the cache, providing our Tag Table with significant power and performance advantages.

Finally, the dirty bit of the compressed entries is unchanged from the dirty bits in other implementations and serves to identify which chunks contain data that is modified from main memory and need to be written back on eviction.

7. Optimizations

7.1. Prefetching

The first optimization involves a prefetching mechanism that can be used to accommodate the limited chunk specification imposed by the compressed entries. As a baseline, Tag Tables utilize a mechanism that evicts the shortest chunk to make room for a new chunk specification. Leveraging a unique property of Tag Table's compressed entries that allows them to consume *less* storage space by tracking *more* data however, Tag Tables can instead utilize a reactive prefetching mechanism to maintain no greater than four chunks per entry. In situations where insertion of data would otherwise create a fifth distinct chunk in the associated compressed entry, this mechanism can prefetch data to combine existing chunks in order to free a chunk for the new data.

By way of example, consider the DRAM row presented in Figure 7 when the cache attempts to insert the "New Block" which maps to Entry 1 (the entry corresponding to the green/light blocks). Since Entry 1 is already tracking four chunks, there is no room to add the new block. The prefetch mechanism addresses this problem by identifying the least gap between Entry 1's segments and prefetches data to fill it. In this example, that gap exists between Chunks 3 and 4. Therefore, the blocks with offsets 0x2E and 0x2F associated with the entry are fetched from main memory and inserted to allow Chunk 4 to merge with Chunk 3. The end result is an increment of Chunk 3's 'length' field by 0xA (to 0x12) to accommodate the two prefetched blocks and the eight blocks previously tracked by Chunk 4, leaving Chunk 4 free to track the newly inserted data.

The procedure to determine this least gap occurs following the bitvector population and victim determination of insertion as described in Section 6.1 and consists of a number of 6-bit adders, one for each chunk, to determine the upper value of each chunk. Following this, the output of the adders (the top block of each chunk) is subtracted, again using 6-bit subtractors, from the base of the next chunk and fed into a 3-input comparator which chooses the smallest input (from the three chunk gaps). From this identification of the least gap, the actual blocks to prefetch can be easily determined by fetching a number of blocks beyond the top of the lower chunk equal to the value of the least gap.

As with insertion, although this operation consumes additional cycles it occurs off the critical path, following an already long latency miss to acquire the fill data, and only involves serializing two addition operations and a 6-bit comparison and thus has negligible effect on performance. An alternate method

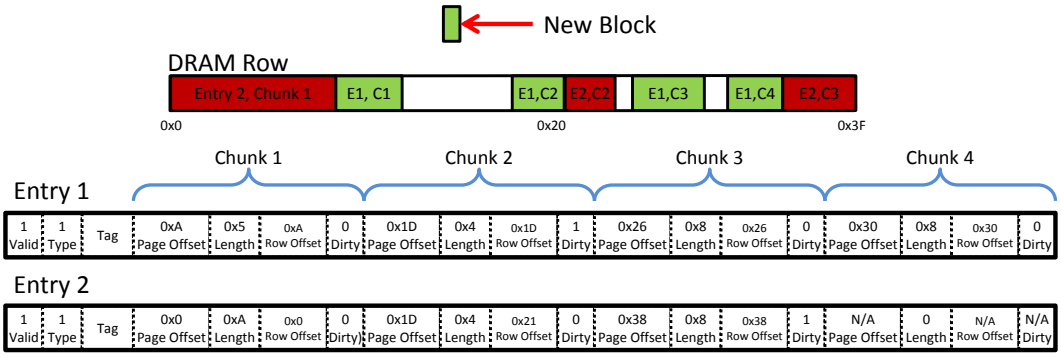

Figure 7: ***Example compressed entries associated with a DRAM cache row.***

to maintain correctness however, which we will evaluate in Section 8.5, is to evict the shortest chunk instead.

7.2. Colocating Metadata

In addition to reducing the average number of L3 accesses for metadata by allowing compressed entries to exist in intermediate levels as discussed in Section 4.2, Tag Table's structure provides an additional means for optimizing metadata retrieval. Assuming a system with a shared L3 that is physically composed of multiple, distributed slices, it is possible with our Tag Table's tree structure to partition it in such a way that the metadata associated with a partition of the table be co-located with the L3 slice that would trigger its access. If the bit interleavings associated with each slice of the L3 is static and known at initialization, it is a simple matter to maintain those portions of the Tag Table on the paths associated with a given slice at the L3 slice itself. This opportunity frees our Tag Table structure from the long latency accesses previously assumed for DRAM cache metadata, specifically the 24 cycles assumed for the MissMap [10]. Instead, considering a ring-based, 8-core CMP system, with one L3 slice per core, L3 accesses consist of 1) communicating a request to and returning data from the proper slice, 2) accessing the tag information in that slice, and 3) accessing the associated data (on a hit). Therefore, utilizing a uni-directional ring for inter-core communication, the communication portion of the latency comes to 16 core cycles round-trip on average assuming the network operates at core frequency and only requires two cycles per hop (1 cycle for switch traversal and 1 for link traversal). Following communication, accessing tags for a 1 MB bank of SRAM takes 2 core clock cycles, while data access requires 6 core cycles as determined by CACTI [18] for a 32 nm SRAM process, rounded up to the nearest whole cycle. In total, these latencies lead us to the 24 cycles assumed in prior proposals. However, each access of the Tag Table for metadata takes only the 8 cycle serial tag and data access time.

7.3. Protecting Metadata

The final optimization involves protection of Tag Table metadata. After initial evaluations, along with usefulness of protecting application data from metadata, it was determined that the metadata can benefit from protection as well (i.e., preventing application data from causing excessive metadata evictions). This is a situation commonly encountered with "streaming" applications (i.e., those with little data re-use after cache insertion). Therefore, along with enforcing a "high watermark" setting, Tag Tables also selectively enforce a "low watermark" as well. Utilizing a "set dueling" mechanism to determine when to enforce this setting [15] the amount of *metadata* that can be evicted by *application data* can be limited. Evaluation of the effect of this dueling is presented in Section 8.5.

8. Evaluation

We evaluate the performance of our Tag Table structure by comparing it against a baseline configuration of a recent server chip (Intel Gainestown based on the Nehalem architecture) with configuration details provided in Table 1. We further evaluate the performance of two prior state-of-the-art tags-in-dram approaches - the Alloy Cache and the Loh-Hill Cache - to place Tag Tables in perspective. From these analyses, we are able to illustrate the main contributors to Tag Table's improved performance over the prior proposals. Finally, we perform initial exploration into the opportunities available to Tag Tables when incorporating a prediction mechanism to proactively prefetch blocks (in contrast to the reactive prefetching presented in Section 7.1).

8.1. Simulation Infrastructure

In order to simulate sufficiently large regions of applications to exercise such large DRAM caches, we utilize a trace-based simulator that implements an abstract core model along with detailed models of the memory hierarchy above the L3 cache (i.e., the L3 cache, the L4 DRAM cache, and main memory). We generate our traces using the Pin-based Sniper simulator [6] utilizing a Gainestown configuration with private L1 and L2 caches and a shared L3 that interfaces with main memory. Traces consist of those accesses seen by the L3 cache, grouped into epochs for coarse-grain dependency tracking as described by Chou, et. al. [7]. These traces are then consumed in a second phase by a simulator that incorporates the abstract core model utilizing the trace's epoch notations for proper issue cadence, issuing requests directly to a detailed L3

Processors & SRAM Caches	
Number of Cores	8
Frequency	3.2 GHz
Width	4
L1 Icache (Private)	32 KB, 4-way, 4 cycles
L1 Dcache (Private)	32 KB, 8-way, 4 cycles
L2 (Private)	256 KB, 8-way, 11 cycles
L3 (shared)	8 MB, 16-way, 24 cycles
Stacked DRAM	
Size	*64MB*, 256MB, *1GB*
Block Size	64 B
Page Size	4 KB
Tag Table Associativity	64-way (4KB pages / 64 B blocks)
Bus frequency	1.6GHz (DDR 3.2 GHz)
Channels	4
Banks	16 per Rank
Bus width	128 bits per Channel
Page Policy	Close Page
Metadata Access Lat.	8 cycles
PRE/ACT Latency	36 cycles (18 ACT + 18 CAS)
Data Transfer	4 cycles
Off-chip DRAM	
Bus frequency	800 MHz (DDR 1.6 GHz)
Channels	2
Ranks	1 per Channel
Banks	8 per Rank
Row buffer size	4 KB
Bus width	64 bits per Channel
tCAS-tRCD-tRP-tRAS	9-9-9-36

Table 1: ***System Configuration.***

cache. Misses and dirty writebacks from the L3 are then fed to either a DRAMSim2 [16] interface that performs detailed main memory modeling - when simulating the baseline, no L4, system - or to a detailed L4 DRAM cache for DRAM cache configurations. When implementing an Alloy Cache, a perfect 0-cycle memory access predictor is assumed, allowing DRAM cache misses to be issued to main memory in parallel with the DRAM cache tag access. Further, we do not penalize the Alloy Cache for its additional burst of traffic for tag data over the DRAM cache data bus.

8.2. Workloads and Methodology

We evaluate our cache configurations on the applications of both the PARSEC benchmark suite utilizing the native input sets [4] and the SPEC 2006 suite utilizing the reference input set executing in rate mode that exhibit better than 2x performance improvement with a perfect L3 cache. While we do not present results for those applications that have less than 2x improvement with a perfect L3 due to space constraints, our evaluations of the whole PARSEC benchmark suite (except *facesim* which encountered trace generation difficulties) found

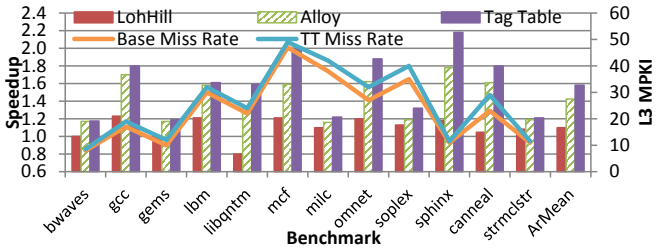

Figure 8: ***Results for 256MB DRAM Cache.***

that they all achieve modest performance improvement with both the Alloy Cache and Tag Table implementations.

We simulate the first 10 million L3 accesses in the regions of interest (ROI) for all benchmarks. For SPEC, the ROI is the primary 1 billion instruction simpoint [17] (i.e., the 1 billion instruction slice that contributes the most to overall execution), while PARSEC utilizes the hooks in the integrated benchmarks of Sniper to identify the beginning of the parallel region of execution as the ROI.

Virtual to physical address mapping for the workloads is accomplished through a random, first-touch translation mechanism to simulate long-running system characteristics. We ensure warm cache state for our evaluations by restoring L3 and DRAM cache state through a memory timestamp record mechanism [3]. For PARSEC this structure is created during trace generation and dumped immediately prior to entering the ROI. For SPEC this structure is created by simulating the 1 billion instruction simpoint immediately prior to the evaluated simpoint.

8.3. Overall Results

Figure 8 presents the speedups achieved by system configurations with either the Tag Table, Alloy Cache, or Loh-Hill Cache managing a 256MB L4 DRAM cache relative to a baseline system that interfaces the 8MB L3 directly with main memory (i.e., no DRAM cache). Overall, this graph shows an average speedup of 56% for Tag Tables relative to the 41.5% achieved by the Alloy Cache and 10.6% for the Loh-Hill Cache. Further, it shows the *L3* miss rates on the secondary y-axis that both a baseline system achieves with full access to the L3 and a Tag Table system achieves with metadata pollution in the L3, showing that the impact is not so great. Notably though, the miss rate increases can be seen to frequently track the workloads where Tag Tables achieve lesser speedup relative to the Alloy Cahe, leading to the conclusion that metadata pollution can be a non-trivial factor for some workloads, causing them to access the DRAM cache more than they otherwise would. As we will show later in Figure 10 with the DRAM cache miss rates however, these L3 miss rates are compensated for by substantially improved DRAM cache miss rates.

In addition to providing the average depth of the tree as mentioned previously, Table 2 provides further high-level details, summarizing many key metrics for evaluating Tag Tables. From left to right, these metrics are the average number of

Bench-mark	*Blocks /Entry*	*L3 Occu-pancy*	*Tree Depth*	*Lvls Acc'd*
bwaves	47.6	16.6%	2.01	1.13
gcc	43.8	16.7%	2.03	1.71
gems	33.9	22.7%	2.09	1.85
lbm	56.7	13.4%	2.01	1.57
libqntm	28.0	24.9%	2.03	1.35
mcf	30.6	24.9%	2.02	1.65
milc	45.7	16.7%	2.19	1.45
omnet	25.0	24.4%	2.04	1.96
soplex	44.2	17.6%	2.05	1.7
sphinx	46.5	16.5%	2.02	1.76
canneal	38.7	20.4%	2.16	1.99
strmclstr	28.6	23.7%	2.03	1.53

Table 2: ***Impact of Design Decisions on a 256 MB DRAM Cache.***

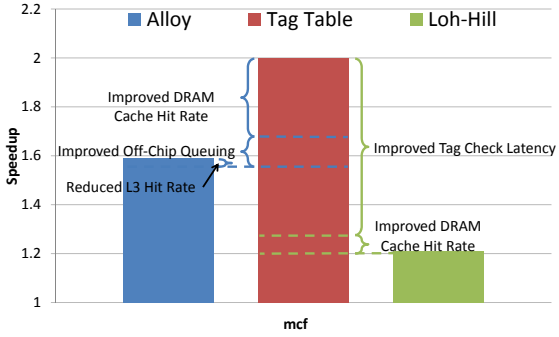

Figure 9: ***Illustrative breakdown of key characteristics resulting in Tag Table performance relative to Alloy and Loh-Hill caches.***

blocks tracked per entry, the percentage of the L3 occupied by Tag Table metadata, the average depth of the Tag Table tree, the average number of L3 accesses for metadata required by each DRAM cache access (i.e., number of levels traversed per access), and the increase in L3 misses per 1,000 instructions due to metadata pollution. Together these statistics show that applications exhibit a range of behaviors that a Tag Table is able to dynamically adapt to without consuming an excessive amount of L3 capacity (less than 25% in all cases).

8.4. Distinguishing Tag Table Features

In order to understand the key characteristics driving the improved performance of Tag Tables relative to the Alloy and Loh-Hill Caches, Figure 9 presents a rough breakdown of significant effects leading to the improvement in the *mcf* benchmark[2]. While *mcf* is chosen as the illustrative example, other benchmarks are significantly similar in terms of relative importance of the various factors.

As the figure shows, the primary differentiator between Tag Tables and the Alloy Cache is the reduced DRAM Cache miss rate which improves from a harmonic mean of 27 MPKI for the Alloy Cache down to just 17 MPKI for Tag Tables over the evaluated workloads, accounting for about three-quarters of the final speedup improvement. This result is particularly important given the fact that the Alloy Cache actually has improved hit and miss latencies compared to Tag Tables. The hit latency of the Alloy Cache is 43 cycles compared to Tag Table's 50 cycles, while the 0-cycle misses afforded the Alloy Cache with its modeled perfect predictor compares to about 14 cycles for Tag Tables. Thanks to this much lower miss rate however, Tag Tables are able to service L3 misses much faster on average since many fewer must access off-chip DRAM. In fact, this reduction in off-chip DRAM accesses provides a final, non-trivial additional factor with improved off-chip DRAM service time. Finally, the figure also takes into account the small effect of reduced L3 capacity of a Tag-Table-based system relative to Alloy due to metadata pollution. The low value of this negative contribution highlights the robustness of using the L3 replacement mechanism and set dueling to store Tag Table metadata as opposed to a dedicated structure.

For the Loh-Hill Cache, the primary differentiator is easily the difference in tag check latency created by the need to access additional blocks of tag data in order to determine location, accounting for about nine-tenths of the difference. Compared to the average 107-cycle hit latency of the Loh-Hill Cache, Tag Tables are substantially lower at 50 cycles. Miss latencies are similarly improved from a 24-cycle MissMap access in the Loh-Hill Cache to the 14-cycle lookup of Tag Tables. The additional tag blocks are also responsible for the remaining effect, the somewhat decreased hit rate due to lower associativity available by occupying a number of ways with tag information.

To quantify the hit rate component, which is a major factor for both Alloy Cache and Loh-Hill Cache comparisons, Figure 10 shows the miss rates observed in terms of misses per 1,000 instructions (MPKI) in the DRAM cache for the three configurations. As the figure shows, the associativity maintained by both the Loh-Hill Cache and Tag Tables results in very similar miss rates that are effective at avoiding a significant number of cache misses relative to the direct-mapped Alloy Cache.

Finally, Table 3 provides data related to average memory access times (AMAT) of the various cache configurations over the benchmark set. As shown, while Tag Tables exhibit a higher hit latency relative to the Alloy Cache, the improved hit rate results in improved AMAT. Further, the second order effect of improved off-chip access time provided by the reduced off-chip traffic of the associative caches is reflected in the average miss latencies of Tag Tables and the Loh-Hill Cache. While this effect is not large enough to overcome the significant hit latency increases of the Loh-Hill Cache relative to the Alloy Cache, it serves as an important additional benefit of Tag Tables.

8.5. Sensitivity Analysis

Figure 11 presents the performance achieved by Tag Tables with and without optimizations presented previously. Specif-

[2]The *mcf* benchmark is chosen simply due to the fact that it shows relatively large improvements from Loh-Hill to Alloy to Tag Tables, providing more graphical room to present the changes

Bench-mark	Loh-Hill			Alloy			Tag Tables		
	Hit Rate	Miss Lat	AMAT	Hit Rate	Miss Lat	AMAT	Hit Rate	Miss Lat	AMAT
canneal	58.1%	161.2	129.7	47.0%	185.9	118.2	65.0%	168.8	94.8
strmclstr	36.6%	155.9	138.1	33.5%	176.5	131.5	39.4%	160.2	117.5
bwaves	41.6%	139.6	126.1	35.5%	158.6	117.2	42.3%	122.7	91.8
gcc	74.9%	186.9	127.1	60.7%	225.4	114.1	80.7%	237.3	88.3
gems	46.1%	310.2	216.6	40.1%	322.9	210.2	54.4%	310.8	170.8
lbm	58.1%	307.8	191.2	49.5%	338.8	191.9	68.7%	225.9	106.3
libqntm	23.4%	176.6	160.3	20.0%	199.9	168.3	26.9%	184.0	148.1
mcf	52.5%	171.3	137.5	52.8%	196.4	114.9	62.9%	174.6	97.6
milc	20.9%	211.7	189.8	21.5%	232.7	191.6	24.2%	208.6	170.5
omnet	98.4%	149.4	107.7	75.8%	168.1	72.5	98.8%	151.9	55.0
soplex	59.9%	205.8	146.6	54.6%	251.6	137.1	61.2%	214.7	115.5
sphinx	53.3%	169.8	136.3	57.0%	206.1	112.6	59.7%	178.9	103.6
Average	*52.0%*	*195.5*	*150.6*	*45.7%*	*221.9*	*140.0*	*56.4%*	*197.3*	*114.1*

Table 3: ***Average Memory Access Times (AMAT). Hit Latencies: Loh-Hill 107 cycles, Alloy 42 cycles, Tag Tables between 49 and 54 cycles (dependent on number of L3 lookups required for metadata - "Lvls Acc'd" in Table 2)***

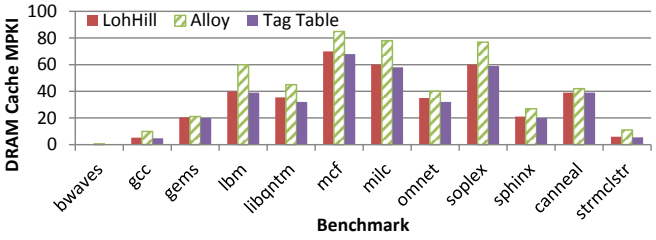

Figure 10: ***Miss Rates of Various Tag Tracking Mechanisms.***

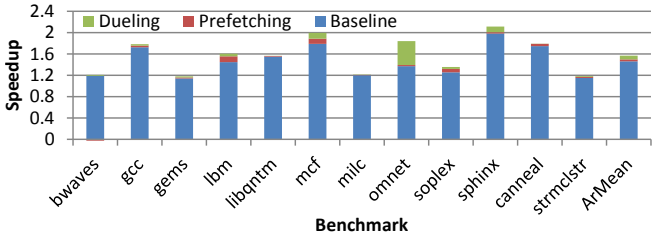

Figure 11: ***Sensitivity Analysis of Tag Table Optimizations.***

ically, these optimizations are 1.) the use of prefetching to maintain no more than the maximum number of supported chunks (Section 7.1) and 2.) the use of set dueling to selectively protect Tag Table metadata from being evicted by application data (Section 3.2). As the figure shows, the majority of the benefit of Tag Tables is achieved with a simple system where the appropriate number of chunks (less than four for our evaluation) is maintained by evicting the shortest existing chunk and where metadata is evicted without restraint by application data in the L3 cache. When selectively enabling the optimizations however, it can be seen that certain applications can be significantly affected. Specifically, *omnetpp* exhibits a significant performance improvement when set dueling is enabled. As was the motivation for including metadata protection, *omnetpp* exhibits excessive DRAM cache evictions due to L3 metadata evictions when not protected. By enabling set dueling, once it is observed that hit rates have degraded sufficiently in the non-protected sets versus the protected sets (by the Tag Table frequently missing on metadata accesses), a "low watermark" setting is enforced globally that prevents application data from evicting metadata below a certain threshold (four ways in our evaluation).

8.6. Prediction Opportunities

As alluded to previously, Tag Tables may benefit from previously proposed prediction techniques, including the Fooprint Cache proposal (FPC), which maintains an associative table of recently evicted sectors that records the subblocks that were demanded during its last residence in the cache [9]. This table is accessed on a miss using the PC and sector offset and is used to determine the set of blocks that are fetched into the sector. Figure 12 shows the potential benefit of prediction with a Footprint-Cache-like mechanism on our evaluated workloads by mimicking a perfect prediction mechanism where all referenced blocks are fetched, as observed from a prior execution trace. The figure presents the same prediction mechanism applied to Tag Tables, using chunks as the sector analogy. In other words, when a miss is encountered in the Tag Table, a number of contiguous blocks are fetched that will be accessed prior to the eviction of the chunk. With a few exceptions, Tag Tables without a predictor outperforms the Footprint Cache design, since it avoids the conflict misses caused by the sectored organization of the Footprint Cache tag array. FPC prediction shows great potential in many cases, though our results - which are based on workloads consistent with those evaluated with the Alloy Cache [14] - do not directly correlate with the workloads used in the prior work on FPC [9]. Combining FPC prediction with Tag Tables provides additional gains, and integrates easily with the Tag Table organization; we leave exploration of a detailed prediction scheme to future work.

9. Related Work

In addition to the Alloy Cache, Loh-Hill Cache, and Footprint Cache; TIMBER is another proposal - while presented as a DRAM cache inserted between the logic die and phase-change main memory - that could similarly be adapted to tracking

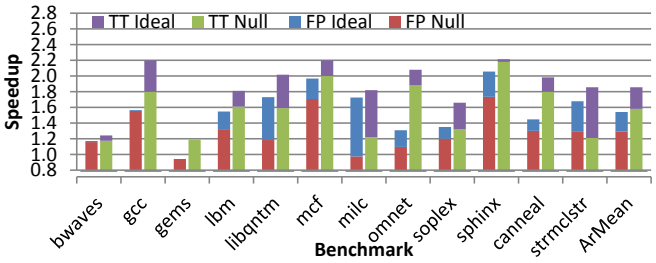

Figure 12: ***Opportunity of Prediction on Footprint Cache and Tag Tables. Tag Tables without footprint fetch (TT null) and ideal footprint prefetch (TT ideal), and Footprint Cache without prefetch (FP null) and with ideal prefetch (FP ideal).***

tags for die-stacked DRAM or eDRAM caches [11]. TIMBER augments tags-in-DRAM with an SRAM cache of recently-accessed DRAM cache blocks. On a hit to this small cache, the exact location of data is provided, as with Tag Tables. On a TIMBER cache miss however, the DRAM cache must be accessed to determine definitively whether or not the block is present. In this way, it can outperform the MissMap for blocks with temporal locality at the expense of increasing miss latency. Hit latency to blocks that miss the SRAM cache is unaffected. Unlike Tag Tables, TIMBER's cache does not provide the full tag information in SRAM - limiting knowledge to recently-accessed blocks - however it could potentially be adapted to avoid Tag Table walks by querying it first on an access, possibly improving the performance of applications that exhibit temporal locality at the DRAM cache.

10. Conclusion

This paper proposes Tag Tables, a robust and dynamically adaptable solution to the tag tracking problem for large capacity caches with traditional block sizes that allows a system to utilize a large capacity DRAM cache to achieve an average speedup of greater than 58% for a range of multithreaded and multiprogrammed workloads. Unlike previous proposals for small-block DRAM caches, Tag Tables are realizable with a storage requirement suitable for implementation on the speed-optimized SRAM logic die. This small storage footprint is accomplished through the combination of the on-demand nature of a forward page table and compressed base-plus-offset entry encoding. Relative to prior state-of-the-art approaches, Tag Tables outperform the Alloy Cache by 10% and the Loh-Hill Cache by 44%. In addition, we have presented initial investigation into the feasibility of incorporating the orthogonal footprint prediction method utilized to great effect in the Footprint Cache.

11. Acknowledgments

This work was supported in part by the National Science Foundation under grants CCF-1116450 and CCF-1318298 and a gift from Qualcomm Research.

References

[1] G. Anderson and J.-L. Baer, "Design and evaluation of a subblock cache coherence protocol for bus-based multiprocessors," University of Washington, Tech. Rep. UW CSE 94-05-02, 1994.

[2] J. Andrews and N. Baker, "Xbox 360 system architecture," *Micro, IEEE*, vol. 26, no. 2, pp. 25–37, March 2006.

[3] K. Barr, H. Pan, M. Zhang, and K. Asanovic, "Accelerating multiprocessor simulation with a memory timestamp record," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, 2005, pp. 66–77.

[4] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[5] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. Loh, D. McCauley, P. Morrow, D. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die stacking (3d) microarchitecture," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 2006, pp. 469–479.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 52.

[7] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, 2004, pp. 76–87.

[8] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation intel core processor," *Micro, IEEE*, vol. 34, no. 2, pp. 6–20, Mar 2014.

[9] 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 ISCA-40*, 2013, pp. 404–415.

[10] 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*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 454–464.

[11] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, July 2012.

[12] M. Ohmacht, D. Hoenicke, R. Haring, and A. Gara, "The edram based l3-cache of the bluegene/l supercomputer processor node," in *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, Oct 2004, pp. 18–22.

[13] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 258–269. Available: http://dx.doi.org/10.1109/MICRO.2012.32

[14] M. Qureshi and G. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, 2012, pp. 235–246.

[15] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391.

[16] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5. ACM, 2002, pp. 45–57.

[18] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi, "Cacti 5.3," *HP Laboratories, Palo Alto, CA*, 2008.

[19] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 304–316, Oct. 2002.