Tag Tables

By

Sean Benton Franey

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2014

Date of final oral examination: 08/04/14

The dissertation is approved by the following members of the Final Oral Committee:

Mikko H. Lipasti, Professor, Electrical and Computer Engineering
Parameswaran Ramanathan, Professor, Electrical and Computer Engineering
Mark D. Hill, Professor, Computer Science
Nam Sung Kim, Associate Professor, Electrical and Computer Engineering
Xinyu Zhang, Assistant Professor, Electrical and Computer Engineering

UMI Number: 3635337

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 3635337

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

© Copyright by Sean Benton Franey 2014

All Rights Reserved

To "my girls," Shari and Grace

Acknowledgments

Although mine is the only name credited on the first page of this document, it would be silly to think that my contribution represented any more than a plurality of the work that was necessary to make it, and what it represents, possible. As they say "it takes a village" in the context of raising children, but it is no less the case when completing a Ph.D. To that end, here are some of the residents of my village. By no means is this an official census, but I hope it encompasses the primary individuals that were necessary for its smooth administration.

First and foremost, I would like to thank my wife Shari and my daughter Grace, whose support and personal sacrifices were the single most important contributions to my achieving the goal of receiving a Ph.D. While my daughter might not realize her sacrifices now, I cannot help but think how her life would have been different if I would have taken the simpler path more than 10 years ago and just entered the workforce as so many of my peers did after finishing their naval service. Although I believe it would have resulted in a more stable school and friend situation for her, not to mention not having to live in four different states before she was 4 and six different houses - including 6 weeks in a hotel - before she was 7, I am hopeful that this experience will serve her well in the future, teaching her to adapt to new situations and giving her an appreciation for the experiences of others. I further hope this experience helps teach her the importance of hard work, perseverance, and the pursuit of her goals. While I would like to think that my example will be a major

contributor to teaching her that lesson, no less of a contribution will come from my wife, whose own sacrifices have been significant over the years. Despite my pursuit causing her to live in various parts of the country that she was not initially too excited about and has more importantly kept her from spending the time she would like to with her family, she has never suggested I abandon my goal or failed to give me anything but the utmost support in order for me to put forth my best effort. Indeed she was accommodating and supportive of me even when she herself was struggling with the long hours of busy work and studying required to earn her own college degree. Fortunately now however, I think she might go ahead and try to cash in on her equity as we move on to the next phase of our life, back in her hometown with her family, where she can hopefully devote more of her time to the things that are most important to her by supporting her mother and more fully involving herself in our daughter's activities.

Second, I would like to thank my parents. Through their example, I learned the importance of hard work, making your own luck through preparation and persistence, and following through with a goal. Further, I do not think I would have necessarily had the stomach to take some of the necessary risks I did without the knowledge that they would be there for me to help if I failed. Also, their attentive ear and encouraging words were helpful for putting my trials and stresses into perspective, allowing me to deal with them rationally and appropriately.

Third, I would like to thank my advisor, Mikko Lipasti whom I feel exceptionally lucky to have met and had the opportunity to work with. If I was to list the most important features of an advisor before starting my journey, Mikko would have certainly met all of the most important ones. For one, his insight and intelligence are extraordinary and were invaluable in pointing out the opportunities and potential pitfalls of various research directions and were no small help in identifying root causes of complex issues that are inevitably encountered when complex systems such as processors are studied and modeled.

Second, frequent discussions with him, particularly early on, were extremely useful in keeping me on the right path, not venturing helplessly into the weeds, not dwelling too much on failures, and highlighting the valuable lessons and opportunities even failures revealed. Third, he is well-balanced and understands the external pressures we all face, particularly concerning marriage and children, never making me feel like I was failing in my duties when family demands dictated I spend less time on research. Finally, I sincerely appreciate the sense of ownership he allowed me to develop for my work. Rather than imposing his will throughout the investigation and writing process, Mikko did an excellent job at guiding my research along, providing advice but never making me feel like my work was simply an extension of his consciousness. Instead, I feel like our time together has truly been a collaboration.

Finally, I would like to thank my fellow students. I want to thank Arslan Zulfiqar for the comraderie during the many shared hours of thesis writing and generation of defense presentation slides. I want to admonish Mitch Hayenga for spoiling episodes of *Breaking Bad* and distracting me from my work, but thank him for engaging me in conversations that helped me think critically - not just pessimistically - about the research proposals of others. I want to thank Dibakar Gope for making me glad I was always done with the next unpleasant thing he had to do in pursuit of his degree. I want to thank Andy Nere for involving me in the extension of one of his papers for publication in a journal. Without that experience, I would not have had the chance for meaningful collaboration with a peer on a technical paper during my graduate school career. Last, and certainly least, I want to thank Dave Palframan for his habit of innocently ruining my day by casually pointing out fatal flaws in my experimental methodology or configuration and for being an example of a well-rounded individual whose status as a successful researcher is only one small facet of his overall being.

Contents

List of Tables

List of Figures

Abstract

The emergence of tightly integrated DRAM through embedded DRAM (eDRAM) or stacked DRAM, provides a high-capacity data store near the high-speed logic die that many have envisioned as an additional level of cache. Unfortunately, the large capacities afforded by these caches create a tag tracking problem. Either small blocks require too many tags to store them on the high-speed logic die or large blocks incur excessive conflicts.

This thesis presents Tag Tables, a technique for tracking tags that is especially suited to large last-level caches. Tag Tables solve the tag tracking problem while achieving the seemingly contradictory goals of maintaining small blocks, high associativity, and storage of tags on the high-speed logic die. They accomplish these goals by exploiting the high spatial locality exhibited by these high-capacity caches through a highly efficient mechanism for tracking contiguous chunks of data allowing an average tag cost on the order of just a few bits per block. Further, Tag Tables employ a dynamic page table structure to perform tag checks, allowing them to adapt their structure to system operating characteristics. By utilizing a Tag Table structure to administer a large DRAM-based cache, this thesis shows that a system is able to achieve 56% speedup on a mix of mulitprogrammed and

multithreaded workloads relative to a baseline system without a DRAM cache, beating prior state-of-the-art that stores the tags in the DRAM itself.

Beyond presenting the basic Tag Table mechanism, this thesis further explores unique opportunities for optimizing tag checks in Tag Tables. These opportunities include the use of Set Dueling to balance metadata and application data in memory, the application of a unique *reactive* prefetching technique along with more traditional *proactive* techniques, and the adaptation of techniques for accelerating page table walks such as lookaside buffers and translation caches. Through the combination of these additional opportunities, Tag Tables are capable of achieving better than 65% speedup over a baseline, no-DRAM-cache system and 18% over the prior-state-of-the art.

1 Introduction

Recent technological advances incorporating dense DRAM close to the processing cores of a computing system - namely embedded DRAM (eDRAM) and stacked DRAM (to be discussed in more detail in Chapter 2) - provide unprecedented low-latency, high-bandwidth data storage. While the capacities are large given their proximity to the core, they are nevertheless unlikely to provide sufficient storage to replace main memory for all but certain embedded applications as argued by Black et al. [9]. As such, recent products such as IBM's POWER7 [39], Intel's Haswell [20], and Microsoft's XBOX 360 [2] along with research proposals such as Loh and Hill's Cache [33], Qureshi and Loh's Alloy Cache [41], and the Footprint Cache by Jevdjic, Volos, and Falsafi [27] 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.

1.1 Previous Approaches

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 (typically 32B to 128B), 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, 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, to be discussed in more detail in Chapter 2, Loh and Hill's Cache [33] and Qureshi and Loh's Alloy Cache [41], both attempt to mitigate this issue by removing DRAM cache access from the critical path on misses, either through 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.

Alternatively, a tag tracking mechanism can be optimized for tag storage and thus avoid the fundamental issue of accessing DRAM for tags by implementing large allocation units. 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 comes from much higher miss rates from cache 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 Footprint Cache proposed by Jevdjic et al. for one, adopts a sectored cache design and extends it with a predictor for a request's *footprint* [27]. Coupled with the application characteristics of emerging *scale-out workloads*, which exhibit a high degree of *page access density* (i.e., the number of subblocks accessed within a sector prior to eviction) which reduce the rate of cache conflicts, the footprint predictor is able to prefetch the subblocks expected to be accessed during the life of the sector based on past behavior of the sector when allocated by the same PC and sector offset. While shown to work well on the evaluated workloads, consuming relatively little SRAM capacity (<2 MB for a 256 MB cache), and proposing a useful prefetch mechanism, it is an open question - one that will be investigated in Chapter 6 - how well a sectored cache approach will work on more

general applications.

1.2 Tag Table Features

What the Loh-Hill Cache and Alloy Cache do not exploit and the sectored cache is unable to exploit effectively due to rigid sector boundaries, is that the capacities afforded by these large DRAM caches result in increased spatial locality of blocks in the cache through longer residency times. The effect of this spatial locality is the duplication of a significant number of tag bits between blocks in the cache due to large contiguous regions - or "chunks" - of main memory data eventually being present. In this thesis, we present Tag Tables, a mechanism which is able to exploit this increased spatial locality and repeated tag bits by storing them in a base-plus-offset encoding. Further, Tag Tables represent a dynamic tracking structure that is adaptable to changing application characteristics through the adoption of a page table data structure.

1.2.1 Compressed Block Storage

Partly inspired by Witchel, Cates, and Asanović's Mondrian Memory Protection [52], the tag encoding utilized by Tag Tables is implemented similar to a sectored cache by storing a base tag for a large region (i.e., the repeating bits) but then indicating the number of contiguous blocks present past the base tag with a "Length" field as shown in Figure 1.1 as opposed to a bitvector as in a sectored cache. With this encoding, significant savings, on the order of a sectored cache, can be achieved with Tag Tables, as will be quantified in

Figure 1.1: Base-plus offset encoding utilized by Tag Tables to greatly reduce tag storage requirement.

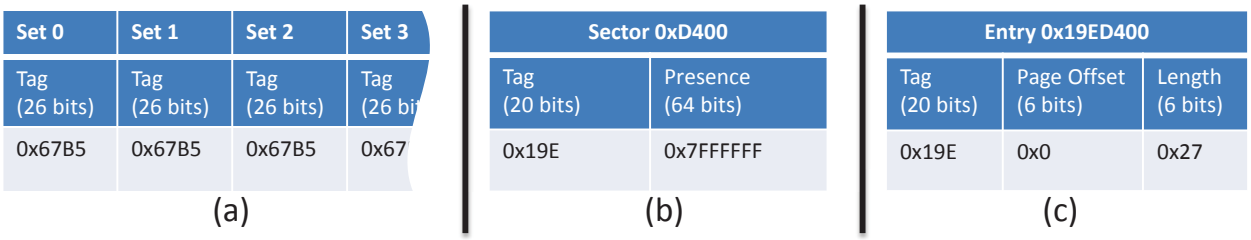


Figure 1.2: Example storage of tags for (a) traditional tag organizations, (b) a sectored cache, and (c) a Tag Table.

Chapter 3, without the internal fragmentation or increased conflict rates of sectored caches.

To illustrate the point of repeated tag bits over various configurations, refer to Figure 1.2. This figure compares the tag structures associated with a 256MB cache with 64B blocks when 27 blocks associated with addresses 0x19ED400000 to 0x19ED400980 are being tracked. In order to mimic the sort of configurations assumed for DRAM cache tag tracking, the traditional cache is 64-way set associative, the sectored cache has 4KB sectors (leading to 64 blocks per sector), and the Tag Table cache utilizes 4KB rows. Further, the figure only considers the information necessary for tracking presence, therefore no dirty bits, LRU information, etc. are considered.

As the figure highlights, the traditional cache has a significant amount of repetition in the tag bits of these contiguous blocks and indeed the figure does not even show all of the tags necessary to track the data. While the sectored cache can help consolidate these bits, it still requires a large bitvector to track the exact blocks that are present. A compressed Tag

Table entry can similarly consolidate these repeated bits but it can also more efficiently track the blocks present within the region, consuming only $\log_2(\textit{Max blocks tracked by an entry})$ bits. These savings come at a cost of flexibility in that full enumeration as to the presence of blocks is lost. Chapter 3 will describe and evaluate approaches to mitigate this effect for workloads that exhibit less contiguity in cache blocks.

The extra "Page Offset" field in the figure in a Tag Table entry relative to the sectored tags would be analogous to the sectored cache storing the sector number (0xD400) rather than implying it with the tag array index. It is important to point out that this extra field helps Tag Tables avoid the significant conflict issues encountered with sectored caches, which are exacerbated by large sectors. These conflicts stem from the rigid sector boundaries imposed by sectored caches and mean that any data from a memory region mapping to the same sector in the cache that does not match the tags of the existing data must evict the data currently present. Maintaining these bits in the entry allows Tag Tables to flexibly place data in the cache and allows data from disparate memory locations to co-exist in the same row/sector.

1.2.2 Dynamic Data Structure

Along with compressed storage of tags, Tag Tables leverage a forward page table data structure in order to dynamically adjust to system runtime characteristics. Unlike traditional tag arrays which must allocate enough storage for the tag and other metadata bits at design time, a forward page table is created and modified over time, consuming more or less

storage overhead as necessary as application characteristics evolve. The application of this structure allows Tag Tables to dynamically adapt to system characteristics during runtime and facilitates the application of existing research into accelerating page walks such as Translation Lookaside Buffers (TLBs) and translation caches, as will be explored in Chapter 5.

1.2.3 Storage of Metadata

Finally, similar to a page table which stores its metadata in main memory, Tag Tables store their metadata in the on-chip L3 cache rather than dedicated storage, further allowing them to adapt their characteristics over time. This storage in the L3 cache allows for the adaptation of caching policies to balance L3 and DRAM cache pressures to provide the best configuration for application characteristics. Rather than having to statically determine the appropriate L3 occupancy for metadata at design time, the dynamic page table structure of Tag Tables allows the system to adjust caching policies dynamically to emphasize either L3 capacity or DRAM cache capacity. Chapter 4 will adapt Set Dueling techniques to this purpose, showing significant opportunity for transferring application working sets between the L3 and DRAM caches based on the pressures they introduce. For example, a workload with a primary working set that easily fits in the L3 cache and a secondary working set that fits in the DRAM cache can afford to allocate more DRAM cache metadata in the L3 to ensure the secondary working set is covered. In contrast, a workload that pushes the boundaries of the L3 can force a policy that devotes as much capacity as possible to the L3

while sacrificing DRAM cache data tracking. These adaptations are not possible with a mechanism that statically allocates resources to tag tracking.

1.3 High Level Operation of Tag Tables

While the specific operation of Tag Tables will be explained in Chapter 3 and further refined in the remaining chapters, Figure 1.3 combines the features so far discussed and provides the high level operation. As mentioned, locating data in Tag Tables occurs as a walk in a forward page table. Starting with a root pointer that must be maintained in a physical register as with a traditional page table, the first bits of the address are utilized to index into the proper entry of the root which in turn provides a pointer to the next level of the table. The walk continues in this way until a leaf entry is found which encodes the row offset associated with the address in a compressed format as described previously. This row offset is analogous to a *way* in a traditional cache. With this information, the DRAM cache can be accessed by concatenating this offset with portions of the address that are analogous to set selection bits in a traditional cache, and represent the row associated with the data. In this way, the exact location of the data in the cache is available. Misses are indicated whenever a path to a leaf entry does not exist or the entry's offset does not match the access's.

As an example of operation, consider access to the cache for address 0x19ED5AF140. The walk will begin by indexing the root level with 0x1AF - the 9 bits just above the page offset. This index will provide a pointer to the next appropriate level of the table, which is

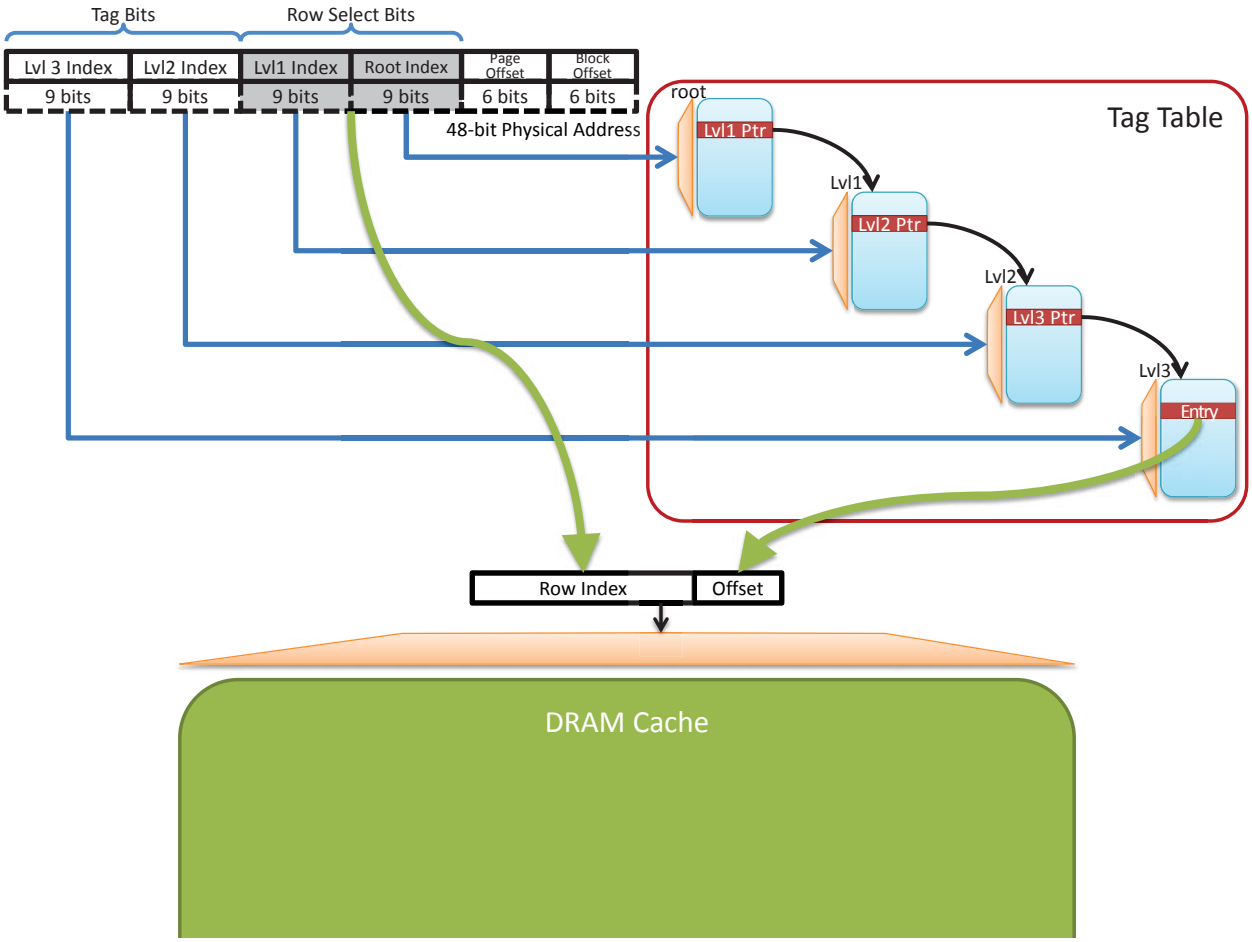


Figure 1.3: high level operation of Tag Tables.

then indexed by 0x16A. This continues until the leaf entry is found. As long as the leaf entry contains location information for offset 0x5 - the page offset of this access - it is a hit and the entry can provide the appropriate *row* offset of the data in the cache. Note, that the offset of the entry need not be 0x5, but can instead be some value less if the offset plus the length is greater than 0x5 due to the compressed storage enabled by the entries. Combined with the row selection bits 0x2D5AF, this offset identifies a unique block in the cache.

The astute reader may notice that the figure represents a page walk using bits in reverse

of a traditional walk (e.g., the bits used to index the root of the table are the lowest-order bits used in the walk, not the highest). This will be explained in Chapter 3 and is an important design decision for Tag Tables which allows them to be more shallow and require fewer traversals on average to determine block location.

The remainder of this thesis will elaborate on this high level description of Tag Tables, propose optimizations, and evaluate the performance of a Tag Table structure for tracking data in a large DRAM-based cache. Further, previous approaches for tracking data in large caches will be described and evaluated relative to Tag Tables, revealing the benefit of various design decisions and optimizations.

1.4 Thesis Contributions

The research presented in this thesis makes the following contributions:

- **Proposes Tag Tables:** Tag Tables are a novel SRAM tag design for large caches with small block sizes that provide an average performance improvement of 58% from an L4 DRAM cache versus a 42% speedup from Alloy Cache - the best prior approach - and 10% from the Loh-Hill Cache.

- **Proposes Set Scouting:** Set Scouting is a novel extension to Set Dueling which allows for selection among a range of policies not just two. Applied to the level of Tag Table metadata allowed to be stored in a system's L3 cache, Set Scouting provides a means

to set a wide range of policies, providing in excess of 5% additional speedup over a baseline Tag Table implementation.

- **Evaluates the tradeoffs of large block sizes and varying associativity for large capacity caches:** A simple first approach to supporting tags for large capacity caches, large block sizes and reduced associativity allow for reduced tag storage overhead. This thesis evaluates the effectiveness of this approach on general workloads.

- **Evaluates the amenability of sectored-cache approaches for modern applications:** A more sophisticated extension of large block sizes, sectored caches decouple fetch from allocation, allowing large sectors to be allocated and only small blocks to be fetched. This organization replaces tags for each block in the sector with a single bit, tracking tags at a sector granularity. While reducing bandwidth demand of large block designs, this approach does not affect conflict rates, therefore this thesis evaluates the effect of these conflicts on modern workloads.

- **Evaluates Footprint Prediction opportunity on current, general workloads in a last level cache:** Spatial Footprint Prediction is a mechanism for improving the performance of sectored caches by prefetching the data to a sector on allocation. While effective for applications that have strong correlation between the instructions that allocate a sector, this thesis evaluates the degree to which modern, general applications exhibit this behavior.

- **Evaluates the effectiveness of virtual address translation mechanisms on Tag Ta-**

bles: Translation lookaside buffers (TLBs) and translation caches are techniques utilized to accelerate address translations in a forward page table. This thesis evaluates their applicability to Tag Tables, revealing TLB opportunity of more than 20% over the workloads evaluated.

1.5 Thesis Organization

Chapter 2 follows to elaborate on previous work in the realm of tags for large capacity caches beginning with a brief description of historical board-level caches, following with recent technologies for creating large DRAM caches, and concluding with a survey of contemporary techniques for tag tracking motivated by stacked-DRAM caches. Beginning in Chapter 3, specific features and design decisions related to Tag Tables are introduced, contrasting them with existing techniques for large capacity caches and evaluating their effectiveness. Motivated by observations made during the performance analysis of the baseline Tag Tables, Chapter 4 evaluates a dynamic mechanism to adjust L3 policies concerning Tag Table metadata storage that utilizes the Set Dueling and novel Set Scouting techniques to control the level of metadata pollution allowed in the L3 cache. In order to further improve the operation of Tag Tables by utilizing existing techniques for accelerating virtual to physical address translations, Chapter 5 presents adaptations of existing page table techniques to Tag Tables such as translation lookaside buffers (TLBs) and translation caches. Further optimizations for improving Tag Table performance by leveraging prefetching techniques are presented in Chapter 6 to populate data in a DRAM cache prior to first

reference. Finally, Chapter 7 concludes by summarizing key insights obtained through the evaluation of Tag Tables and discusses potential avenues for future work or adaptations based on these insights.

2 Background

This chapter serves to present prior work on large capacity caches from historical on-board caches, current designs utilizing embedded DRAM, and up to recent proposals anticipating large cache storage facilitated by stacked DRAM.

2.1 Historical Board-level Caches

While large, fast DRAM-based caches have recently generated quite a bit of excitement and spawned several recent research proposals and products, large caches - relative to on-chip storage - are not without precedent. During the mid-1980's, commercial and academic computing systems made use of caches which provided significantly more storage than was available on chip by incorporating separate memory chips, off the logic die, interfaced with the system bus [23, 24]. Given their location, these board-level caches, contrary to current DRAM-based caches which are largely valuable due to immense bandwidth, provided limited bandwidth as they were frequently pin-limited and needed to share bandwidth with a system bus. Further, the availability and preference of big, "dumb" memory chips to

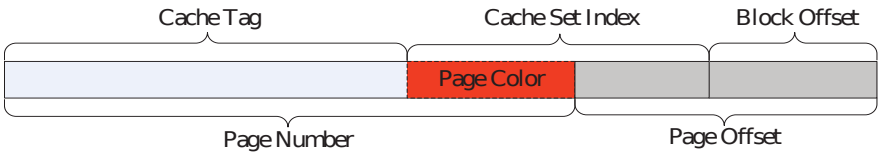


Figure 2.1: Overlap bits used for page coloring (from [14]).

implement these caches meant they were simply used to store bits, not perform tag checks or other complex logic. Instead, tag checking was performed on die resulting in the need to communicate all tags across the system bus. In response, the caches were frequently direct-mapped or implemented with some low degree of associativity to limit the amount of metadata that needed to be transferred.

This decision to maintain little or no associativity in the cache led to such research proposals as page coloring [11]. Page coloring addresses destructive cache interference between processes that can be particularly prevalent when the cache has a low degree of associativity. Coloring is performed at the operating system level by segmenting the physical pages allocated to various processes such that the bits of overlap between the physical page number and the set select bits (the page color) are consistent across all pages allocated to a process as shown in Figure 2.1. In this way, specific regions of the cache can essentially be reserved for various processes, avoiding destructive interference between unrelated processes.

However, as time progressed and technology scaling allowed these caches to move on-chip, such techniques became less relevant as the need to keep low degrees of associativity was lifted. Caches continued to grow in capacity and associativity and techniques such as parallel tag checks became relatively inexpensive and ubiquitous to allow checking of

multiple tags in a set simultaneously. Tag storage increased linearly with respect to cache capacity as block sizes remained relatively unchanged. Relatively little concern was paid to the latency of tag checks and storage overhead of tags until high-capacity caches were once again introduced with high-density DRAM-based caches.

2.2 Large DRAM Caches

Large DRAM caches in the literature come in two flavors: those based on embedded dynamic random access memory (eDRAM) and those facilitated by die stacking, called stacked DRAM. While stacked DRAM is largely an academic technology today, with commercial examples still on the horizon [36], eDRAM has existed in commercial processors since it was presented as the implementation of the L3 cache in the IBM BlueGene/L in 2004 [39] and the POWER7 released in 2010 [12, 29], and is even present in Microsoft's XBOX 360 gaming console (which is powered by an IBM processor) [2].

2.2.1 Embedded DRAM

Embedded DRAM is fundamentally no different from traditional DRAM in the sense that it stores data as charge in a capacitor with an access transistor to allow it to be read and written. However, unlike traditional DRAM implementations whose density-optimized process technology is incompatible with those used to create latency-optimized logic for processors, eDRAM fabrication utilizes compatible techniques to allow the DRAM to be *embedded* with the logic. While the two primary fabricators of eDRAM - Intel and IBM -

utilize different techniques to perform this embedding, the fundamental capacitor-with-access-transistor features remain the same, leading to storage densities 2-4x greater than existing SRAM-based memories which require 6 or 8 transistors [12]. However, due to the latency-optimized process used to create the access transistor, eDRAM cells leak at a much greater rate than traditional DRAM, requiring refreshes up to 1000x more frequently [12, 51].

As mentioned, along with IBM, Intel has begun shipping eDRAM systems with their recent Haswell processor family [20]. While a package-level offering as opposed to IBM's truly integrated, on-die approach, Haswell represents another high volume manufacturing point for eDRAM and is likely just a starting point with future designs expected to use a truly embedded approach [10, 30].

Both IBM and Intel implement their eDRAM with trench capacitors. Given the proportional relationship between capacitance and plate area as shown in Equation 2.1 in Figure 2.2, creation of a capacitor with planar plates on a silicon substrate in current process technologies would consume excessive area in order to achieve reasonable capacitance and thus retention times. Therefore, trench capacitors form their plates by creating a deep trench in a substrate and creating one capacitor plate as a doped area around the trench and the second is formed in the void of the trench with a dielectric barrier between the two as shown in Figure 2.2. The IBM eDRAM trench capacitor for instance, has a greater than 35:1 aspect ratio, meaning the trench is 35 times deeper than it is wide [12], significantly limiting its footprint on the silicon surface while achieving enough capacitance to be able

$$C = \epsilon_0 \frac{a}{d} \qquad (2.1)$$

a – Area of plates
d – Distance between plates

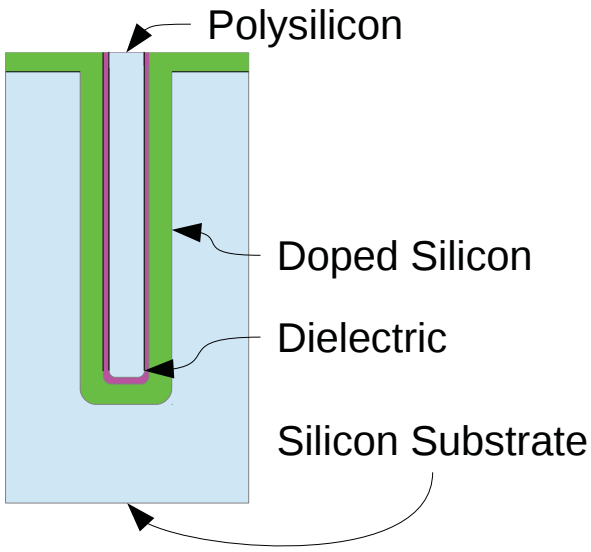


Figure 2.2: Trench capacitor and capacitance equation.

to store sufficient charge.

Formation of Capacitors While the storage mechanism between Intel and IBM is the same, the substrate in which it is created is different. For IBM, this substrate is the bulk silicon itself [12], while Intel forms its trench capacitors in the interlayer dielectrics between the metal interconnects above the silicon [10]. While the close integration between silicon and the capacitor in IBM's process can achieve very good access times, Intel's decision to create the trench in the interlayer dielectrics likely provides for better integration with existing process techniques at the expense of access time. Further, by not interfering with the silicon process, Intel can likely achieve greater density, taking away less silicon from logic, and integrating at a later stage in the fabrication process.

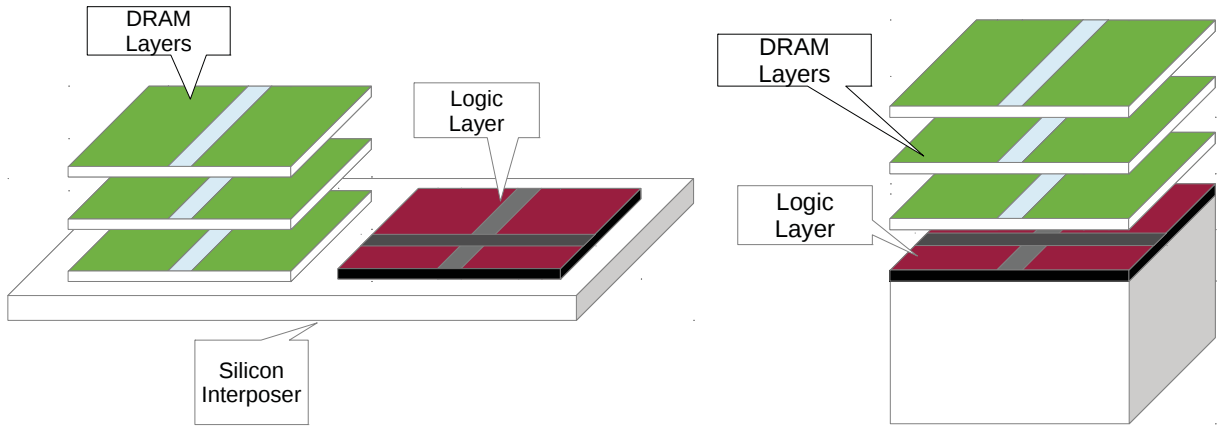


Figure 2.3: (a) 2.5D and (b) 3D die stacking of DRAM on top of high-speed logic (Recreated from [33]).

2.2.2 Stacked DRAM

Contrary to eDRAM which is embedded in the high-speed logic die, stacked DRAM is an application of 3D die stacking allowing the integration of disparate technologies [9, 32, 54]. In this way, both the logic die and the DRAM die can be fabricated in their optimal processes, meaning certain concessions made in eDRAM such as high refresh rates can be avoided. While 3D integration has been proposed as an opportunity for providing high-speed communication between multiple logic dies by extending communication into the third dimension, thereby reducing wire length [34, 40], initial commercial activity appears focused on stacking dense DRAM on top of a high-speed logic and SRAM die. Unfortunately, the capacities afforded by stacked DRAM are unlikely to be sufficient as a system's only data store for all but certain embedded applications, motivating its use as a high-capacity cache [9].

Physically, interfacing multiple dies via stacking can occur either horizontally/2.5D [16]

or vertically/3D [19, 53] as shown in Figures 2.3a and 2.3b. Further, the connections between the dies are likely to consist of silicon interposers [9] or through-silicon vias (TSV) [45, 37], respectively, providing relatively high bandwidth, low latency communication between the two.

2.3 Tag Tracking for DRAM Caches

While eDRAM has been around for some time and board-level caches predated them, the problem of tag tracking for large caches has only recently received significant research interest with the promise of stacked DRAM. Perhaps this is due to the confluence of a large number of blocks to be tracked relative to on-die capacity (a scenario that existed with board-level caches) with the fact that what would previously be considered highly complex on-chip operations (parallel tag checks, memory access predictors, etc.) are now mature, well-understood mechanisms that architects feel comfortable manipulating. Or maybe it is merely because there was so much more low hanging fruit in the days of board-level caches that supporting large, high latency caches was relatively unimportant. Regardless, this section serves to present recent promising proposals in the realm of tracking tags for large DRAM-based caches, motivated by the promise of stacked DRAM that does not have historic precedent whether for a board-level cache or eDRAM cache. These proposals span a range of designs from a scaling up of recent on-chip designs with high associativity and small block sizes, to a direct-mapped cache *a la* historical board-level caches, and finally to a sectored approach.

2.3.1 Loh-Hill Cache

The Loh-Hill Cache (L-H Cache) can be thought of as an approach to push the tag tracking mechanism of recent last level caches (LLC) to a large DRAM-based cache [33]. By maintaining support for "traditional" block sizes (on the order of 32B to 128B) and scaling up associativity with capacity (32-way associativity for up to a 1GB cache), the L-H Cache provides the sort of cache parameters one would expect with a simple scaling of cache capacity. Of course concessions must be made to accommodate this step increase in cache size, considering common LLC capacities are currently on the order of 8MB to 16MB and the L-H Cache considers 64MB to be the smallest size of interest for stacked DRAM caches. Indeed, capacities in the few gigabytes have been announced for the similar Hybrid Memory Cube (HMC) technology [35]. The primary of these concessions is the storage of the tags in the DRAM cache array itself. Considering a 256MB DRAM cache with 64B blocks and 6B of tag for each block leads to 24MB of tag storage. Compared to typical LLC sizes, this is clearly infeasible to store on the logic die, leading to the L-H Cache decision to store them in the DRAM. In order to prevent the high latency of accessing the DRAM array for tags from making a DRAM cache access even slower than a main memory access, the L-H Cache makes two important design decisions, involving 1) exploiting open-row accesses and 2) decoupling hit determination from block locating.

As mentioned earlier, the L-H Cache maintains a high degree of associativity. It does this by considering all of the blocks in a DRAM cache row as a set, leading to 32-way associativity given their 2KB rows and 64B blocks. This leads to the requirement to transfer

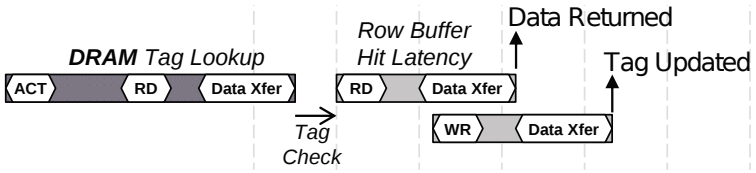


Figure 2.4: Breakdown of latencies for a cache hit in the Loh-Hill Cache (Taken from [33]).

three 64B blocks of tags for every access[1]. By intelligently colocating these tag blocks in the same set as the data, the L-H Cache is able to guarantee an open-row hit for all blocks present in the cache, leading to operation as outlined in Figure 2.4, notably avoiding an 'ACT' for the data access portion. In so doing, the precharge and activate latencies can be avoided, eliminating approximately 16% of the cycles from the data access[2]. A secondary effect of storing the tags in the DRAM row is that the associativity is effectively reduced from 32-way to 29-way to account for the three blocks of tags.

Unfortunately, this open-row optimization only helps DRAM cache hits and does not help misses, leading to significantly longer off-chip access times (an increase on the order of 50 cycles while tags are checked). Therefore, the second key design decision in a L-H Cache is to decouple the hit determination and the data location operations. In a typical cache, the tag check not only reveals whether or not the access hits or misses but it also indicates where the data is located in the cache as the way associated with the tag is the same as the way associated with the data. In the L-H Cache a separate structure called a "MissMap" is used to more quickly determine whether or not a cache access hits or misses without the need to access the tags. Serialized with the tag check, it is only when this map

[1]6B tags multiplied by 32 tags per set results in 192B of data or three 64B blocks
[2]Reduction in access cycles from 114 cycles to 96, as evaluated

indicates that the access will hit that the cache will even access the tags and try to find the data. This way, tag checks are eliminated from misses, replaced by a much lower latency access to the MissMap.

The MissMap is implemented as a map, similar to the tags of a sectored cache, where the page numbers are the key and a bitvector indicating the presence or absence of a block is the value. Unlike a sectored cache however, the resulting location of the block in the row is maintained by the tags not by its position in the page. In this way, the L-H Cache is able to avoid internal fragmentation and conflict misses of a sectored cache since data from conflicting memory locations can co-exist in the same sector of the data array.

As proposed, the MissMap is implemented by "carving off" a portion of the existing on-chip L3 cache. Given a MissMap entry size of 12.5B (36 bits of tag plus 64 bits for the bitvector), approximately 167,000 MissMap entries consume 2MB [33]. Therefore, a careful balance must be struck between reduced L3 capacity and the ability of the MissMap to track data in the DRAM cache. By enforcing the requirement that all blocks in the cache must be tracked by the MissMap, any time a new entry needs to be allocated that would require eviction of an existing entry, the blocks associated with the victim must be evicted. This situation is encountered whenever more unique rows are present in the cache than there are entries in the MissMap. Therefore, while too large of a MissMap will increase L3 miss rates, too few could prevent the DRAM cache from achieving full utilization. Through evaluation, the MissMap is given 2MB of the 8MB L3 cache to balance L3 miss rates with DRAM cache coverage over the evaluated workloads, though the authors concede that this

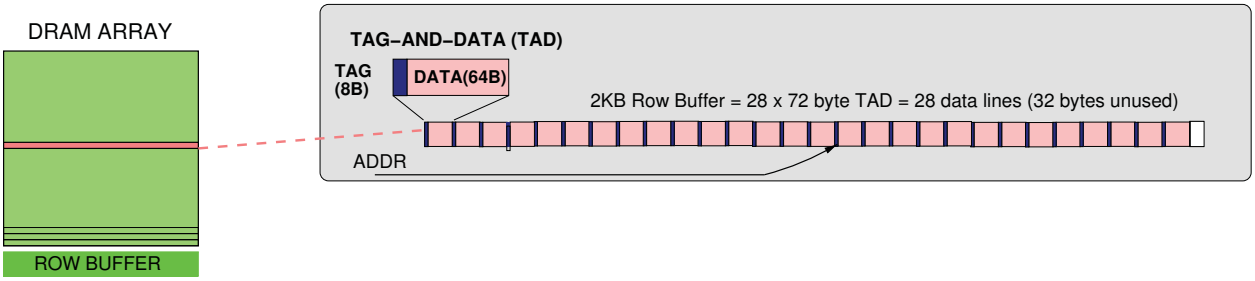


Figure 2.5: DRAM cache row in an Alloy Cache showing the colocation of tags and data (Taken from [41]).

value is highly application dependent.

2.3.2 Alloy Cache

The Alloy Cache is a stacked DRAM tag tracking proposal that returns to the roots of board-level caches. Proposed as an optimization for hit latency, the Alloy Cache is implemented as a direct-mapped cache similar to board-level caches, replacing the long latency tag check of multiple blocks incurred by the L-H Cache with a simple check of a single tag, colocated with the data as shown in Figure 2.5 [41]. In this way, the acquisition of three blocks of tags necessary for a tag check in the L-H Cache is replaced with a single additional burst of data on the bus to communicate the tag associated with the block. Considering 64B blocks, a 16B-wide data bus, and 8B of tag data, the Alloy Cache requires five bursts on the data bus versus the traditional four bursts.

Along with improving the latency of hits, a secondary objective of making the Alloy Cache direct-mapped is to maximize bandwidth. By consuming the cache's data bus for three blocks of tag data only to receive a single block of application data, the L-H

Cache is rather inefficient, achieving only 1.8x of the potential 8x bandwidth improvement of a DRAM cache [41]. In contrast, the Alloy Cache is able to achieve 6.4x bandwidth improvement by keeping the data bus busy transferring useful (i.e., non-tag) data more often.

However, this organization does not address the added latency of the traditional *Serial Access Model* (SAM) when accessing the DRAM cache for requests that eventually miss, increasing the off-chip latency when the cache must be verified as a hit or miss before issuing an off-chip request. This is not a significant problem for applications that have a high hit rate in the DRAM cache, but can have a significant effect on those that frequently miss. Therefore, the Alloy Cache introduces a *Parallel Access Model* (PAM) that issues main memory requests in parallel with the cache access when it believes the access will miss. The determination of the likelihood of a hit or miss is performed by small predictors that can be queried first on an access when in PAM mode. If the predictor indicates that the access will likely miss, an off-chip request is immediately issued to occur in parallel with the access to the cache. Unlike the MissMap, these predictors do not have perfect knowledge of the data in the cache, therefore the access must be made regardless of the predictor outcome in order to avoid receiving stale data from memory. This predictor verification is off of the critical path of misses however, so as long as the predictor is accurate, most misses can complete without increasing the service time beyond the minimal single-cycle access of the predictor itself.

In conclusion, relative to the L-H Cache, the Alloy Cache eschews associativity in order

to improve hit latency by only accessing a single tag. This approach is advocated by a claim that the capacities afforded these DRAM caches makes associativity relatively unimportant. While the benefit of improved hit latency is not likely to be debated, given the *2:1 cache rule of thumb* that states that a 2-way cache is similar to a direct-mapped cache of double capacity [22], the direct-mapped Alloy Cache is equivalent to a much smaller 29-way L-H Cache. Even considering diminishing returns on increased associativity, it is questionable whether this extreme reduction in associativity is worth the hit latency improvements if hit rate is significantly affected. In fact, the replacement of the relatively long-latency access of a MissMap (24 cycles as evaluated in [33]) with simple single-cycle predictor in the Alloy Cache may be sufficient to make the highly associative L-H Cache a better design point. While the predictor is not sufficient for determining if an access is a hit or miss, and it does nothing to improve effective bandwidth into the DRAM cache (since every request has to access the cache regardless of hit/miss), it does serve to remove the penalty incurred for misses to a DRAM cache, similar to the MissMap but much faster, by opportunistically issuing off-chip requests before accessing the cache. Perhaps a balanced approach of moderate associativity on the order of one cache block's worth along with a memory access predictor provides a good balance of hit rate and hit latency between the L-H and Alloy Caches.

2.3.3 Footprint Cache

Finally, the Footprint Cache (FPC) is a sectored-cache approach to tracking data in a DRAM cache [27]. Proposed as a tracking mechanism for some of the earliest cache implementations, sectored caches can greatly reduce the storage overhead of cache tags by only associating a single tag for a large region of memory (a sector) and using a bitvector to indicate which smaller blocks associated with the tag are actually present in the cache [1]. Unlike the MissMap however, which similarly identifies the data present in the cache by using a tag to a large region, since data from multiple regions of memory which map to the same sector can not co-exist in a sectored cache, there can be a significant increase in evictions. These evictions occur when a reference to data that maps to the same sector but with a different tag from the existing data stored in the sector causes the eviction of the existing data, potentially creating a significant amount of writeback traffic if there are many dirty blocks. Further, sectored caches can exhibit significant internal fragmentation if few blocks associated with a sector are actually referenced.

The FPC proposal avoids a significant amount of the fragmentation and conflict issue by applying the approach to a specific set of *scale out workloads* which exhibit a high degree of *page density* which means they reference a significant number of blocks within a sector between allocation and eviction. Further, the FPC includes a predictor for *singletons*, or pages which only reference a single block before eviction, preventing allocation (and subsequent victimization of existing data) for the sector, simply forwarding the data when received. This of course requires a non-inclusive cache policy since otherwise lack of a

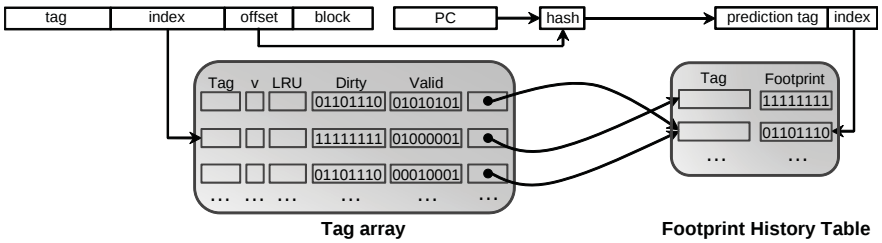


Figure 2.6: The Footprint History Table utilized by the Footprint Cache (Taken from [27]).

particular block of data is not allowed.

The cost of this prediction is relatively small as it builds on another major feature of the FPC, the footprint predictor. Proposed originally as a prediction mechanism for a sectored L1 cache, the footprint predictor opportunistically fetches the blocks that will be referenced in a sector (the sector's "footprint") on allocation in the hopes that they will be subsequently referenced and already present (i.e., will be usefully prefetched) [31]. Since a sectored cache causes eviction of all existing blocks in the sector on allocation, such a prediction merely has an effect on cache fill bandwidth since it will not cause any additional data evictions. Similar to the original proposal, the FPC incorporates a footprint history table (FHT) as shown in Figure 2.6, which is indexed by the allocating reference's offset in the sector along with the program counter (PC) value that referenced it, which itself is retained in the tags of the sector during its lifetime in the cache. The values associated with this index are then the corresponding bitvector of referenced blocks present when the sector was last evicted. The intuition is that the blocks that are referenced within a sector will be stable from one allocation to the next, assuming the allocation occurs from the same instruction and with the same initial offset. As will be shown in Chapters 3 and 6, the applicability of these footprint predictions do not necessarily hold for general workloads, but *scale-out workloads*

exhibit a strong correlation between data accessed and the accessing instruction [48]. Since the bandwidth imposed by the mechanism is highly dependent on the accuracy of the predictor, Chapter 6 will quantify the effect of increased bandwidth, coupled with poor prediction for many general workloads.

2.4 Summary

This chapter first discussed the similarity between large DRAM-based caches and board-level caches of the past. Next, it reviewed the embedded DRAM and stacked DRAM technologies which make these dense, high speed caches possible. Finally, it concluded with a survey of existing proposals for tracking the tags required for managing these high-capacity caches, representing widely disparate views on the importance of such fundamental cache properties such as block size, associativity, and latency to access tags.

3 Tag Tables

As discussed in Chapter 2, the capacities possible with DRAM-based caches lead to a tag tracking problem. Either excessive storage is required for the tags if traditional block sizes are utilized (32B to 128B) or fragmentation issues limit application performance when large block sizes are utilized to mitigate tag storage. This chapter will describe the fundamental features of Tag Tables that facilitate tracking of traditionally-sized cache blocks with storage requirements competitive with large-granularity designs. In this way a Tag Table design is capable of realizing the performance potential of a cache with small blocks while keeping tag storage low enough to be maintained on the latency-optimized logic die itself.

3.1 Motivation

Standard tag array designs have relatively few parameters available to tailor their structure to system operating characteristics (e.g., block size, associativity) and none are easily adaptable at run time. Instead, this inflexibility necessitates extensive offline characterization of expected workloads during the design stage in order to create an acceptable tag tracking

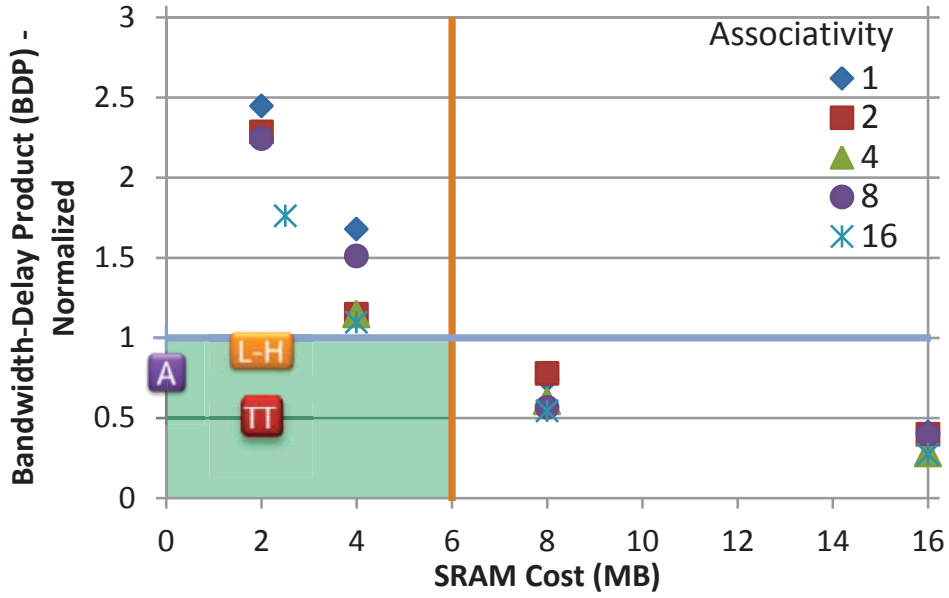


Figure 3.1: Performance of various cache configurations for a 256 MB L4 cache. Shaded area indicates desirable region of improved bandwidth-delay with practical SRAM cost. Results shown for the PARSEC *canneal* benchmark normalized to a baseline with no L4.

structure for the widest possible set of operational characteristics. While many proposals currently exist in the realm of caching to overcome these static constraints, they do so merely by providing a semblance of adaptivity through the addition of various structures and do not address the underlying inflexibility of the substrate [42, 43, 44]. In order to avoid these compromises, this section first investigates a more flexible substrate for tracking tags that itself has a distinguished track record of tracking metadata for large memories - the forward page table. The section continues by evaluating the opportunities and limitations of an oft-considered, decades-old mechanism for limiting tag overhead - sectored caches - to not only highlight the need to shift the baseline tracking mechanism to something more flexible but to identify potential for compressing the storage required for tags.

Figure 3.1 presents the "Bandwidth-Delay product" (BDP) achieved versus the tag stor-

age 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 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 this 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 3.5.4 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 prediction or SRAM-based hit/miss determination structures as the other three state-of-the-art techniques rely on and can in fact be extended by any one of a number of prediction schemes (Chapter 6 will investigate the opportunity

spatial footprint prediction might have for Tag Tables).

3.1.1 Page Tables

Motivated by Figure 3.1, this section begins 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 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, opportunities for adapting the FPT mechanism are investigated which provides the flexibility to adapt to application behavior on demand.

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 3.2. Upon access, a tag check occurs as a virtual address translation does in a forward page table through indexing

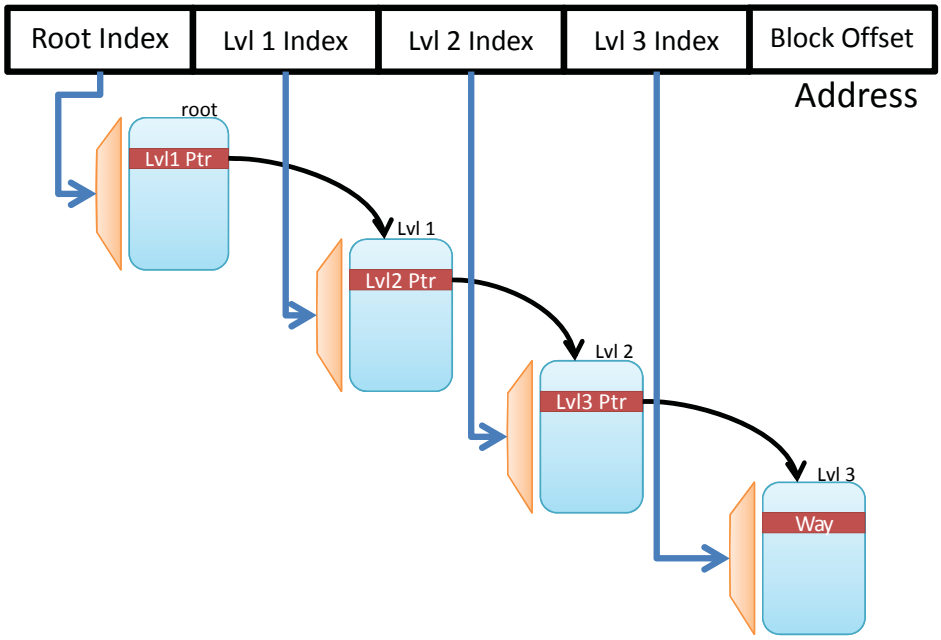


Figure 3.2: High level operation of a forward-page-table-based implementation of cache tags. The set selection and tag check occurs implicitly with the walk, culminating in a hit in the leaf if it is found (which indicates the way associated with the tag if set associative) and a miss if any entry on the path is missing.

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 at 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. The identification of a victim is unchanged from a traditional cache (e.g., selection of the least recently used member of the set). In and of itself, this is not a fundamental

improvement as the system would still need to be provisioned for a worst-case, fully occupied cache at design-time. Further, the combination of multiple levels and way storage in the leaf is not inherently less expensive than a traditional tag structure, but it provides a starting point that is free from inflexible storage requirements and opens up a path to exploit system operating characteristics in order to realize reduced storage requirements.

3.1.2 Sectored Caches

As a first step toward realizing compressed tag storage, lessons from sectored caches - a tag tracking mechanism for some of the earliest cache implementations - are investigated. 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 - or "chunks" - 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* [27]. 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

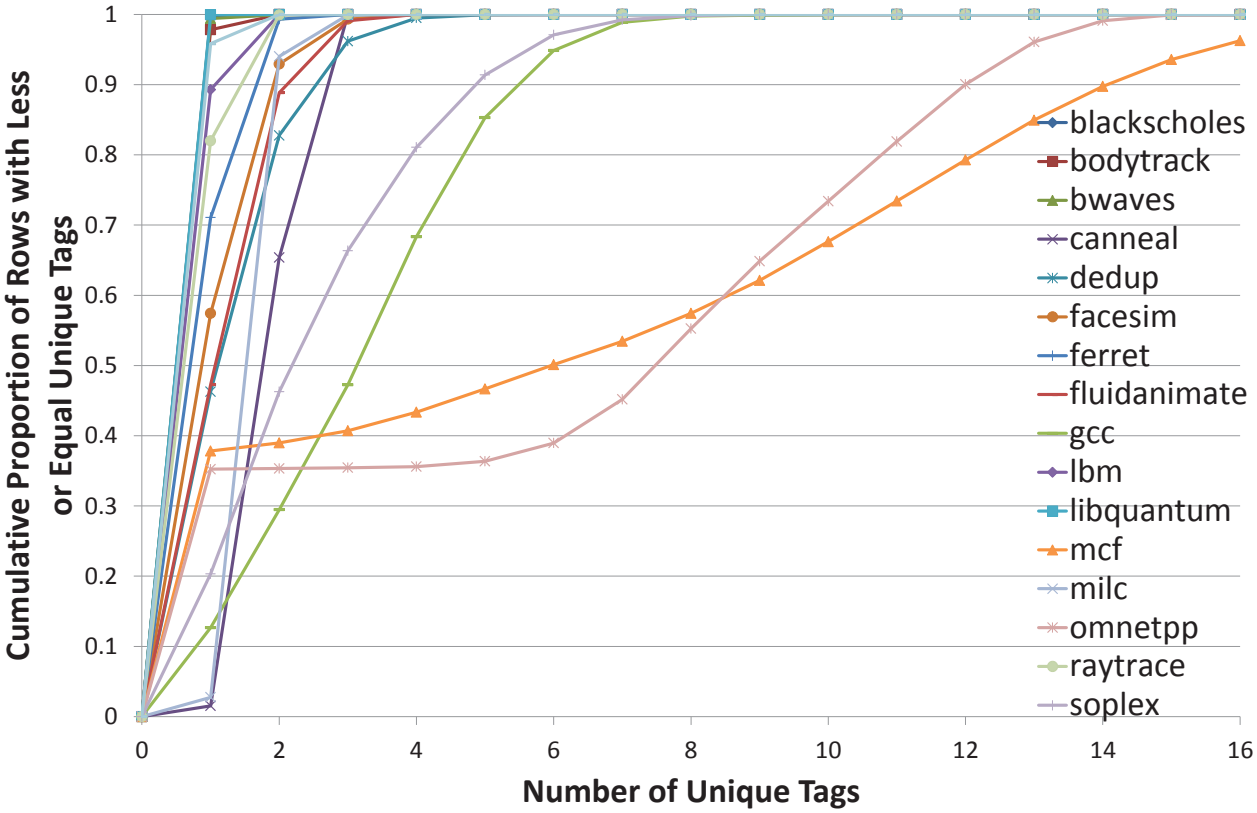


Figure 3.3: Cumulative proportion of DRAM cache rows with less than or equal to a given number of unique tags.

necessary to realize their benefit.

In order to investigate the feasibility of sectored cache designs for large DRAM caches for systems running modern applications, the number of unique tags in a given DRAM cache row is measured as a proxy for the locality available for a sectored cache's large tags. The number of unique tags is determined through simulation of multi-threaded PARSEC and multi-programmed SPEC workloads on a direct-mapped Alloy Cache [41], capturing a snapshot of the unique tags present in a row at the end of simulation (details of the simulation parameters are given in Section 3.5). Presented in Figure 3.3, 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 the identification of a more robust mechanism for reducing the storage requirement of cache tags.

3.2 Compressed Entries

Motivated by the results of Figure 3.3 which indicate a range of spatial locality within a cache both over rows within the cache and from application to application, this section

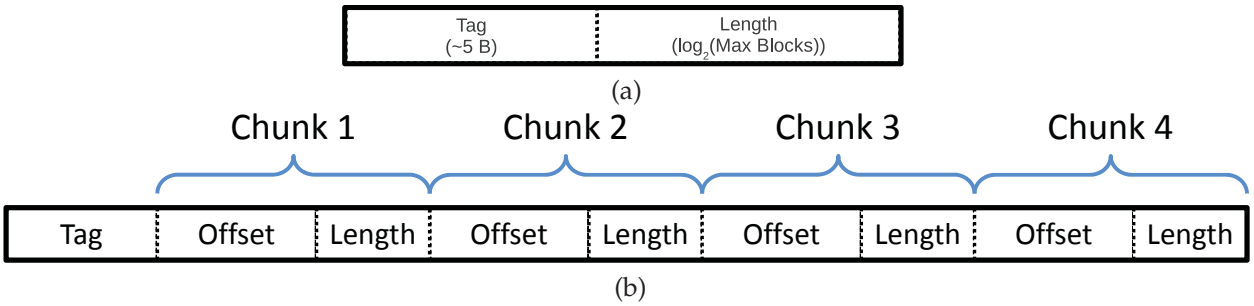


Figure 3.4: (a) Basic & (b) Expanded Entry Formats.

presents 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, the use of a "base-plus-offset" encoding for tracking regions of memory in the cache is employed. In Tag Tables, rather than impose a static sector size and track the blocks within the sector using a bitvector, this encoding instead tracks "chunks" of data, where a chunk is a region of the cache where the data present is contiguous in their addresses. In this way, the storage of the tag associated with the bottom block of the region and the length of the chunk is sufficient to indicate exactly which blocks are present and where they are located (i.e., contiguously with each other from the base up to the base plus the length). Such an encoding can be seen in Figure 3.4a, 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 $\log_2(\textit{Max. no. blocks})$ 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, a hybrid approach utilizing some number of chunk representations greater than one as shown in Figure 3.4b can be used, allowing the representation of holes implicitly in the gaps between chunks. In this format, in addition to the tag and length previously discussed, an "offset" specification must also be used 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 3.1.1, by replacing the simple way-identifying entries proposed at the leaf of the page table with these compressed entries, this structure frees the tag tracking mechanism from a one-to-one mapping of blocks present in the cache to leaf entries in the page table structure. This then allows amortization of the inherently larger entries over a greater range of blocks. Section 3.5 will quantify this amortization by showing that the average number of blocks tracked per entry is actually quite high for those applications in the benchmark set.

3.3 Operation

This section serves to combine the structures motivated and introduced by the previous sections into one coherent structure and provides examples to more concretely describe each component. The basic 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

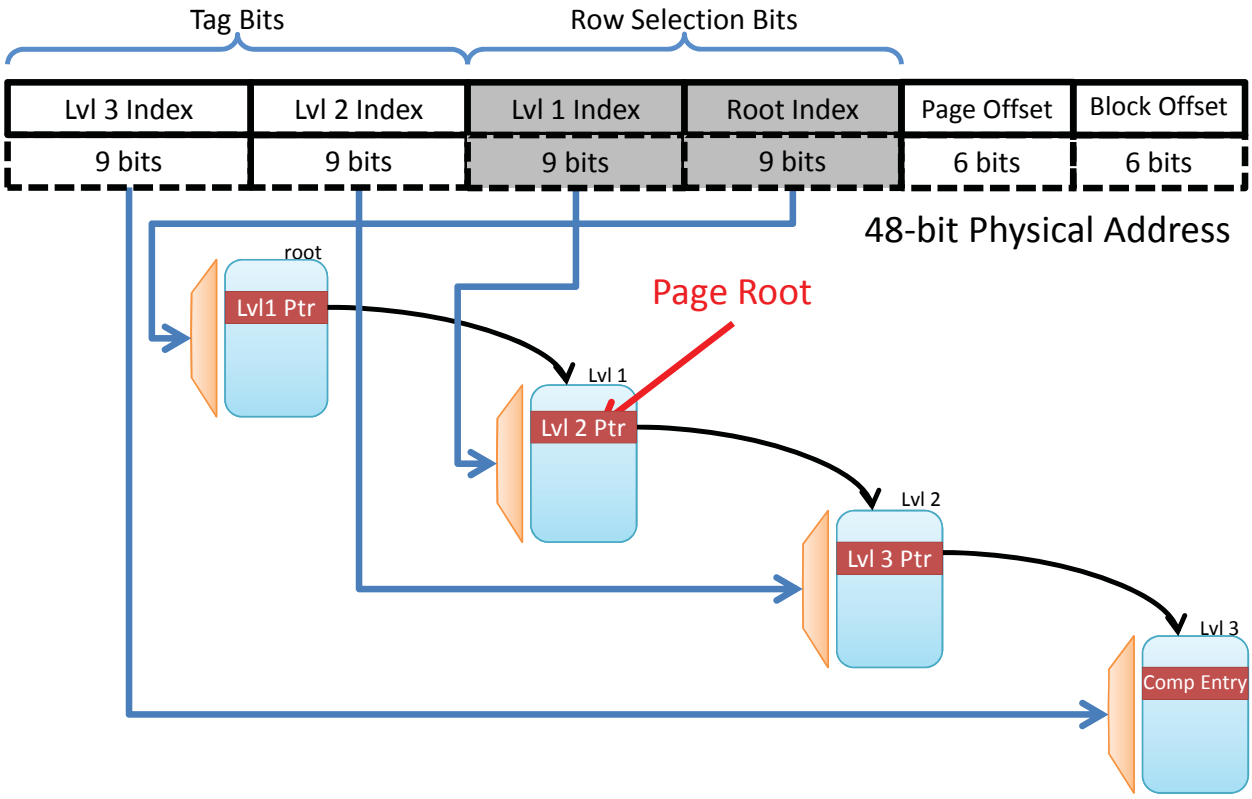


Figure 3.5: Modified high level operation, specific to a 1GB DRAM cache with 4KB rows and 64B blocks, accessed by 48-bit addresses, highlighting the design decisions presented in Section 3.3.1.

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.

3.3.1 High Level

Figure 3.5 presents modifications to the baseline operation presented in Section 3.1.1.

Walk Bit Selection Unlike a traditional page table, a Tag Table reverses the order of the bits used to index various levels in order to utilize the high entropy bits for set selection. 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. A Tag Table however, does not benefit from such support and indeed, can gain significant advantage by reversing the order of the bits selected as will be discuss in the next paragraph and in Section 3.4.2.

Figure 3.5 shows this mapping with low order bits selected to index the upper levels of the table. This is analogous to using low order bits to select a set in a traditional cache with many of the same memory level parallelism opportunities, such as separating spatially local accesses. However, unlike the high level structure presented in Figure 3.2, the "Page offset" bits - just above the "Block offset" - are not used in the walk. This means that spatially local data, within and aligned on a main memory row boundary, are present in the same row of the DRAM cache as well, maximizing the potential for open row hits which is an important feature for row-buffer-based designs. The compressed entries facilitate this decision by including the page offset in the definition of a chunk.

Page Roots One important opportunity of the Tag Table's tree structure and reverse walk, is the ability to infer data location based on the path traversed through the tree. As Figure 3.5 shows, by evenly dividing the number of bits required to uniquely identify a row - $\log_2(\textit{Number of Rows})$ - across some number of upper levels of the table, all of the entries associated with a given row of the cache can be completely isolated to a specific subtree of the Tag Table. The roots of these subtrees are called "page roots" and correspond to the pointers in the first level of the table (Lvl 1) in Figure 3.5. They 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 will be discussed in the next section, this analogy extends to placement in the DRAM cache in that blocks are allowed to be placed flexibly anywhere within their appropriate DRAM cache row.

3.3.2 Insertion

As with any cache, insertion of data is a fundamental 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 multiple 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 3.3.1, 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, Section 3.4.1 will present an optimization that 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 3.5, 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 investigation into more sophisticated approaches is left 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 to 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.

The remainder of this section provides discussion on additional issues related to inser-

tion decisions and identifies choices made in the proposed implementation to maximize the presence of contiguous chunks and limit the worst-case size of the structure.

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) [33]. 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 3.1 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 the 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 3.1).

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

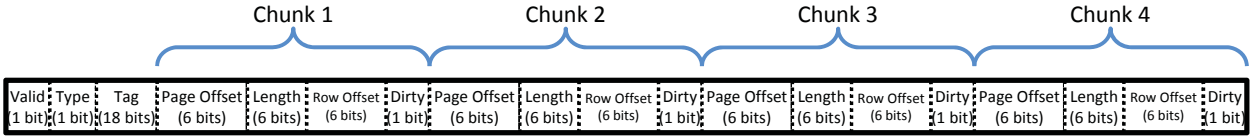


Figure 3.6: Detailed diagram of a compressed entry capable of tracking 4 contiguous chunks of data associated with a tag. Bits correspond to 1GB cache with 4KB rows and 64B blocks associated with a 4-level Tag Table that utilizes 9 bits to index each level.

system beyond the inherent hit rate benefits of associativity which will be discussed in more detail in Section 3.5.4.

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, each offset and length field must be 6 bits wide. This leads to the updated entry format presented in Figure 3.6 whose new fields will be described fully in the summary of this section (Section 3.3.4), 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 from the query would be

0x9.

Limiting Chunks Finally, in order to accommodate the limited chunk specification imposed by the compressed entries, Tag Tables implement a simple eviction mechanism to maintain no more than four chunks per tag. This operation occurs when an access results in adding data in a distinct chunk within an entry that is already tracking four chunks and functions by finding and evicting the shortest chunk (i.e., the one with the least *length* field). The unique compressed entries allow for an alternative mechanism however, by prefetching the blocks existing in a "gap" between existing chunks, allowing two chunks to merge into one instead. Investigation of that mechanism is reserved for Chapter 6, however.

3.3.3 Locating Metadata

The final major design point to discuss relates to the physical location of the metadata that comprises the Tag Table, where metadata refers to the data that is required in order to logically create the Tag Table: the compressed entries and the pointers to intermediate levels. In a traditional tag array, this metadata is the tags themselves. Section 3.5 will show that the compressed entries and forward page table format is capable of tracking the data in a very large cache with conventional block sizes (64B) with a storage overhead on the order of a fraction of a typical L3 cache capacity for the application behaviors we study (on the order of 1 - 2MB for a 256MB cache), therefore it is practical to consider on-chip implementations of storage structures for this metadata. While it would be possible to dedicate a structure by carving out a portion of the L3 cache as is proposed for the *MissMap*

in the Loh-Hill cache [33], Tag Tables instead locate metadata dynamically in the L3, tagging it as such to differentiate it from actual application data. This means that upon creation of a new entry or pointer in the Tag Table, the data associated with it (pointer value, tag, offsets, lengths, etc.) are stored via a write to the L3 and flagged as metadata.

This metadata is allowed to compete with real application data without restriction up to a *high watermark*. The *high watermark* is enforced on a per-set basis and limits 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.

3.3.4 Summary

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 3.5. 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. Hit determination can be performed once a valid entry matching the access's tag is found through the use of a few adders and comparators. Specifically, once an entry is extracted from a metadata block in the L3, each chunk is evaluated in turn. First, using a 6-bit adder, the page offset of the top block of the chunk is computed. This is followed by two comparators that check if the access's offset is within the range of the chunk by testing individually if the offset is greater than or equal to the bottom block and less than or equal to the top block. Finally, the output of the comparators are fed to a 2-input AND gate that indicates a hit on the chunk when both comparators' operations are true. Also, similar to tag checks in traditional arrays, area, power, and speed can be traded off by comparing multiple chunks in parallel.

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 3.1.1 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 dozens 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. This is shown in Section 3.5 where these latencies are taken 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.

3.4 Opportunities and Optimizations

While previous sections have developed a functional tag tracking structure with compression facilitated by base-plus-offset encoded entries and on-demand block tracking through a forward page table, significant opportunities exist for improved performance. In particular, the multiple accesses required for a full page table traversal can be particularly detrimental to both power and performance. Specifically, since Tag Table metadata is tracked in the L3, each level traversed corresponds to an L3 access and its associated latency and energy.

3.4.1 Reducing Levels Traversed

The primary mechanism employed to reduce the average number of levels traversed in the Tag Table is the allowance of flexible placement of compressed entries. This means that the Tag Table is not required to place compressed entries in the last level of the Tag Table and instead is allowed to place them at any other intermediate node on the correct path if

it does not result in ambiguity. 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 compressed entry at the current level for the new data 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 compressed entry 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 Lvl 1, Lvl 2, and Lvl 3 indices as shown in Figure 3.5. This flexible placement further necessitates the "Type" bit in the compressed entries because location in the table no longer implies the type of entry encountered (i.e., whether the entry is a pointer to a subsequent level or a compressed entry).

A useful second-order effect of this flexible placement is observed when inserting new data (as discussed in Section 3.3.2). In the case where an insertion's walk terminates above a page root due to this optimization, a full traversal of a tree is unnecessary for building

the bitvector of blocks present in the row, needing only to set the bits associated with the blocks tracked in the chunks of the terminating entry.

In addition to this upper-level placement of entries, investigation into the use of translation caching to "skip" intermediate levels of the table that are frequently traversed [5] will be presented in Chapter 5.

3.4.2 Colocating Metadata

In addition to reducing the average number of L3 accesses for metadata, the Tag Table 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 the 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 are 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 the structure from the long latency accesses previously assumed for DRAM cache metadata, specifically the 24 cycles assumed for the MissMap [33]. 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 [50] for a 32 nm SRAM process, rounded up to the nearest whole cycle. In total, these latencies are consistent with 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 consumed within the L3 bank.

3.5 Evaluation

Evaluation of the performance of the Tag Table structure is performed in this section 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 3.1. Performance of two prior state-of-the-art tags-in-dram approaches - the Alloy Cache and the Loh-Hill Cache - are also performed to place Tag Tables in perspective. From these analyses, the main contributors to Tag Table's improved performance over the prior proposals are illustrated.

3.5.1 Simulation Infrastructure

In order to simulate sufficiently large regions of applications to exercise such large DRAM caches, a trace-based simulator is utilized 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). Traces are generated using the Pin-based Sniper

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	256MB
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 3.1: System Configuration.

simulator [13] 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. [15]. 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 cache. Misses and dirty writebacks from the L3 are then fed to either a DRAMSim2 [46] 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, parallel access prediction (PAM) is implemented utilizing a perfect 0-cycle memory access predictor, allowing DRAM cache misses to be issued to main memory in parallel with the DRAM cache tag access. Further, the Alloy Cache configurations are not penalized for their additional burst of traffic for tag data over the DRAM cache data bus.

3.5.2 Workloads and Methodology

Cache configurations are evaluated on the applications of both the PARSEC benchmark suite utilizing the native input sets [8] 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. The first 10 million L3 accesses in the regions of interest (ROI) are simulated for all benchmarks. For SPEC, the ROI is the primary 1 billion instruction simpoint [47] (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.

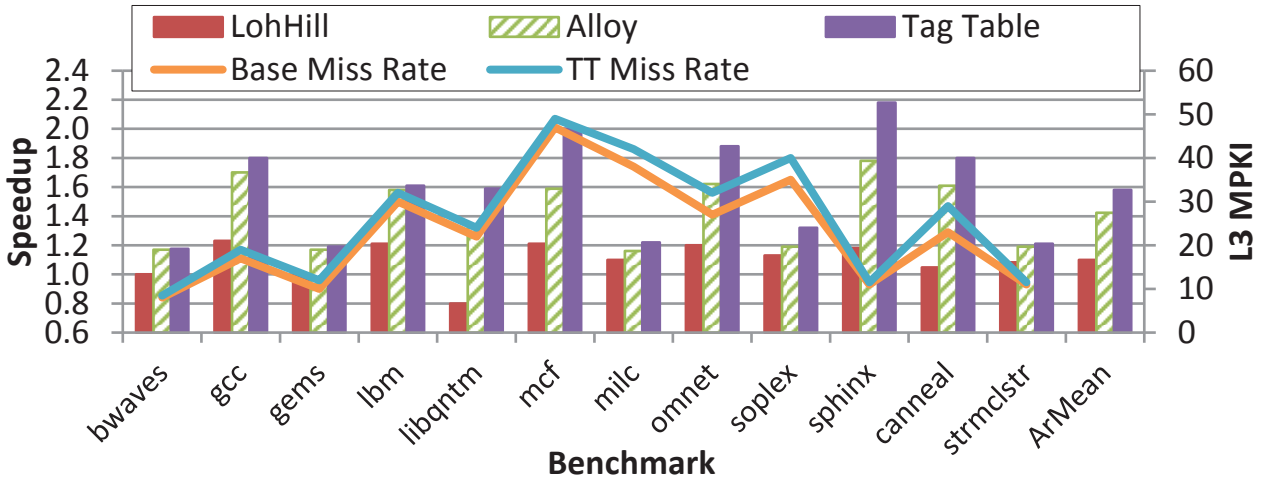


Figure 3.7: Results for 256MB DRAM Cache.

Warm cache state is ensured by restoring L3 and DRAM cache state through a memory timestamp record mechanism [4]. 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.

3.5.3 Overall Results

Figure 3.7 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 minor. 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 3.9 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 3.2 provides further high-level details, summarizing many key metrics for evaluating Tag Tables. From left to right, these metrics are the average number of blocks tracked per entry, the percentage of the L3 occupied by Tag Table metadata, the average depth of the Tag Table tree, and the average number of L3 accesses for metadata required by each DRAM cache access (i.e., the number of levels traversed per access). 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). Further, given this L3 occupancy relative to the number of tags tracked in the DRAM cache leads to an average cost of 3.18 bits per tag lending credence to the claim that Tag Table's compressed entries can significantly reduce tag storage overhead.

Benchmark	*Blocks /Entry*	*L3 Occupancy*	*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 3.2: Impact of Design Decisions on a 256 MB DRAM Cache.

3.5.4 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 3.8 presents a breakdown of first-order effects leading to the improvement in the *mcf* benchmark[1]. 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 improved DRAM Cache hit rate, accounting for about three-quarters of the improvement. A non-trivial additional factor however, is the improved off-chip DRAM service time allowed by reduced off-chip traffic, accounting for the remainder of the difference. Finally, the figure also takes into account the small effect of reduced L3 capacity of a Tag-Table-

[1]The *mcf* benchmark is chosen 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

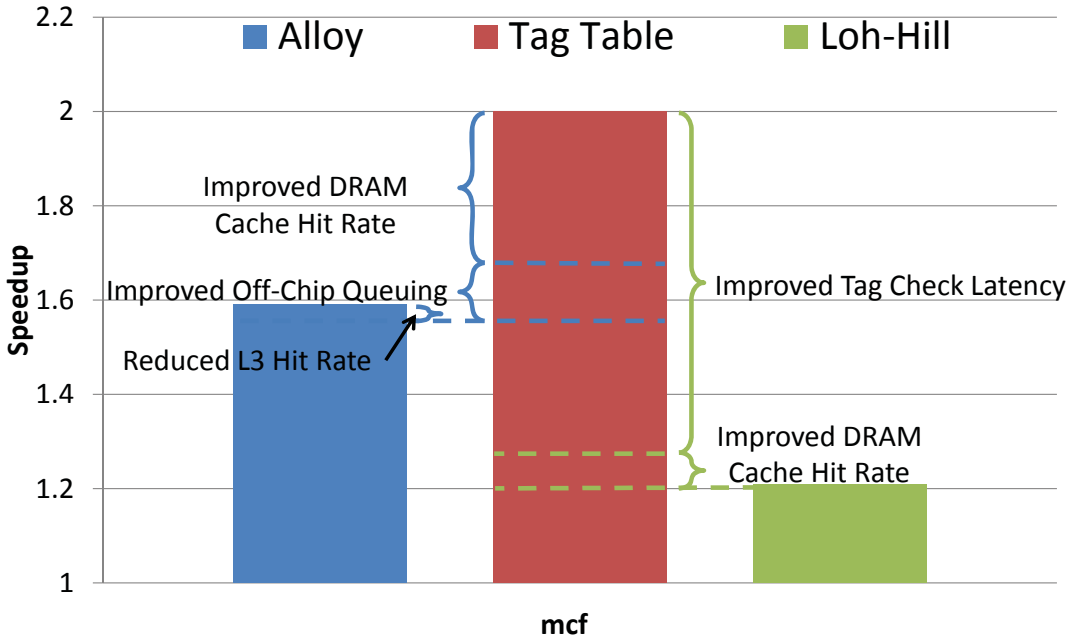


Figure 3.8: Illustrative breakdown of key characteristics resulting in Tag Table performance relative to Alloy and Loh-Hill caches.

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 *high watermark* setting 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. These additional blocks also create 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 3.9 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

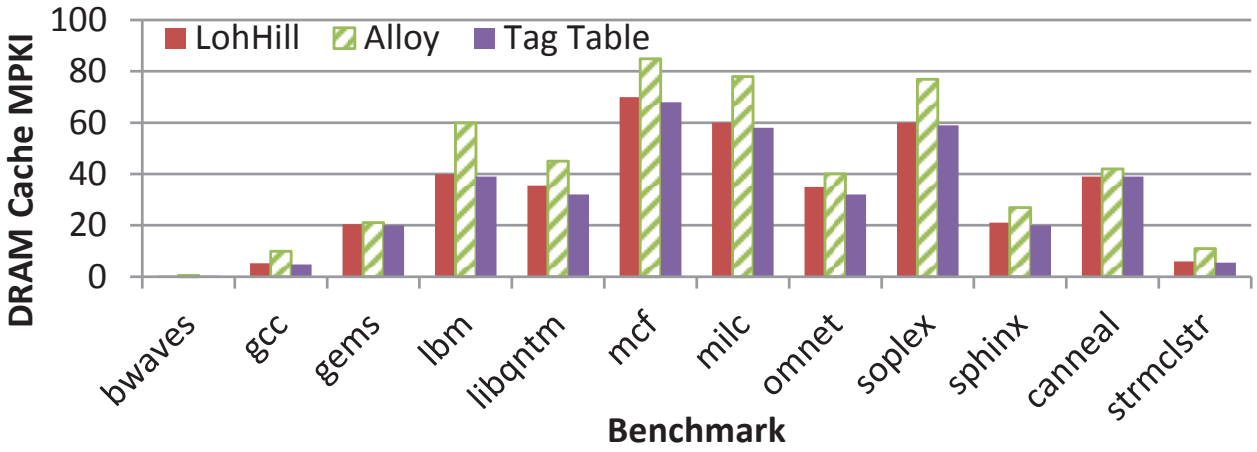


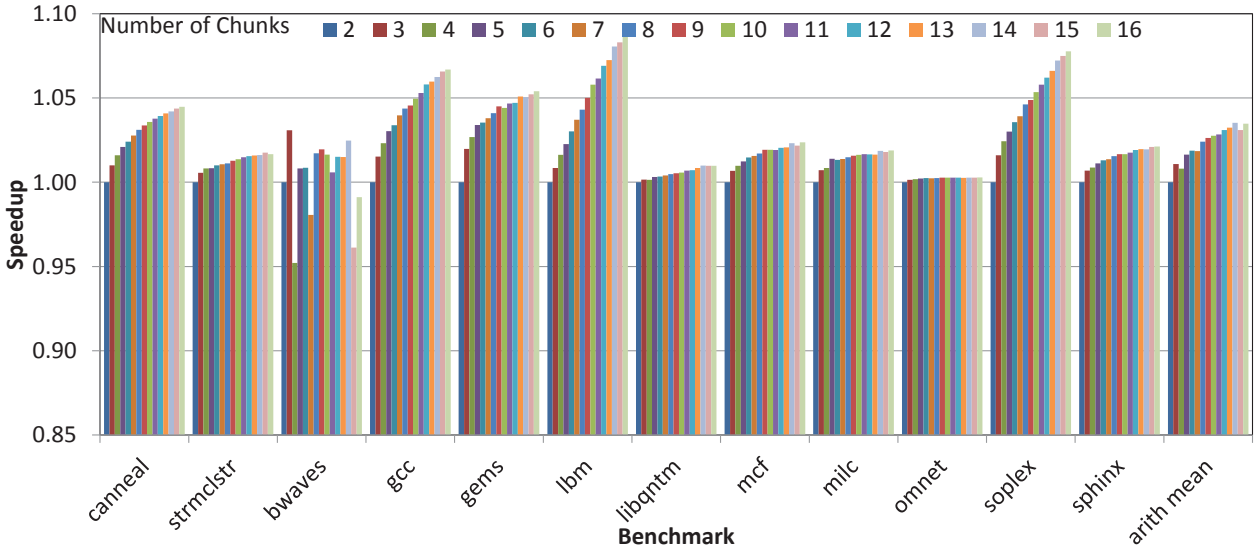
Figure 3.9: Miss Rates of Various Tag Tracking Mechanisms.

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.

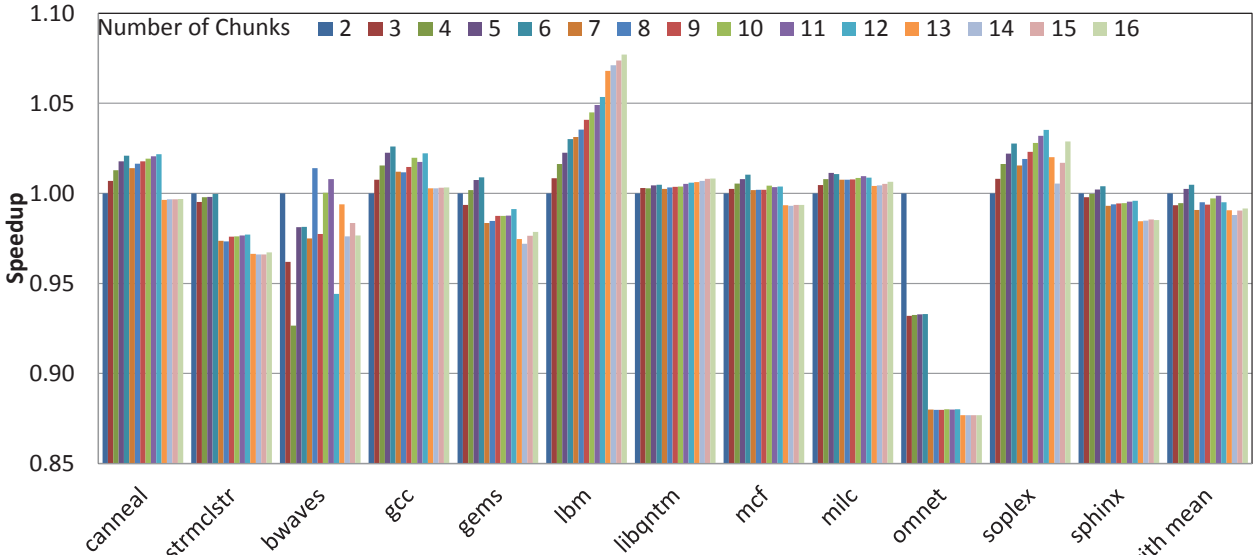
3.5.5 Number of Chunks

Figures 3.10a and 3.10b show the effect of changing the number of chunks each entry is capable of tracking, normalized to the 2-chunk value. While the first figure (3.10a) represents ideal improvement when the change in entry size is not taken into account (i.e., entry size is kept at its 4-chunk size of 96 bits - or conservatively, 4 entries per L3 block), the second figure (3.10b) takes into account the fewer metadata blocks that can be stored per L3 block[2]. Looking only at the first figure, it can be seen that increasing the ability of entries to encode more "gaps" between chunks, ignoring the pollution effect, provides the expected application speedup (note: *bwaves'* wildly varying behavior is likely noise due to

[2]The size of entries is rounded to the nearest power-of-2 bytes when increasing the number of chunks



(a)



(b)

Figure 3.10: Sensitivity of Tag Tables to the number of chunks each entry can maintain, (b) with and (a) without taking changes in entry size into account.

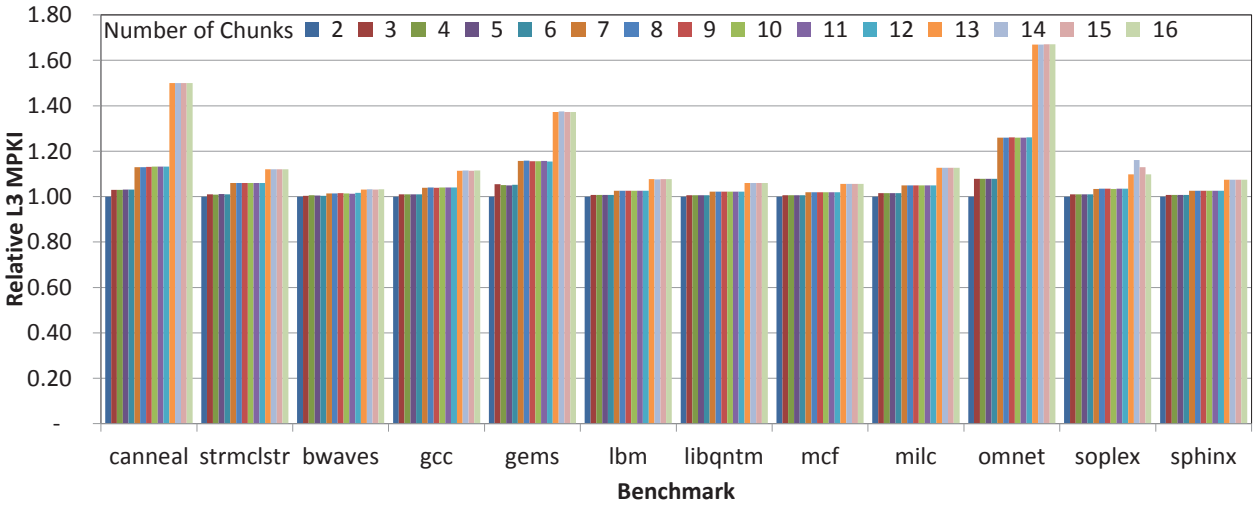


Figure 3.11: Effect on L3 miss rate (per 1K instructions) when increasing the entry size.

its very low DRAM cache miss rate as shown in Figure 3.9). When considering the more realistic second figure however, interaction between L3 metadata pollution and the more flexible chunk tracking is apparent. Reflected in Figure 3.11, the increase in L3 miss rate reflects a strong correlation with application slowdown for those applications that exhibit slowdown in Figure 3.10b, such as *omnetpp*.

Altogether, these graphs reveal a complex interaction between the ability of Tag Table entries to encode more gaps and approach the sectored cache ideal of full enumeration, and the increased L3 pollution these larger entries cause. Chapter 4 will re-visit this analysis when Set Dueling is incorporated to dynamically adapt L3 metadata pollution policies, but for a basic Tag Table, the decision on the correct number of chunks to support per entry is highly application-dependent.

3.6 Summary

In summary, this chapter outlined the detailed operation of a baseline Tag Table tracking mechanism. It showed that Tag Tables are a robust and dynamically adaptable solution to the tag tracking problem for large capacity caches with traditional block sizes. The specific features of Tag Tables allow 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%.

4 Set Dueling

As revealed in Chapter 3, the decision to maintain Tag Table metadata in the L3 cache comes at the cost of reduced effective L3 capacity for application data. This cost is apparent in Figure 3.7 in the increased L3 MPKI for nearly all workloads. However, effective L3 capacity for application data is not the only consideration as highlighted by Figure 3.10b, which indicates complex interactions between L3 metadata pollution and the ability of Tag Table entries to track more ranges. Therefore, metadata occupancy of the L3 involves a careful balance between L3 and DRAM cache capacities. Further, whether metadata is considered merely L3 pollution or whether it is a more valuable quantity is highly dependent on system operating characteristics which can change from application to application and even from phase to phase within applications.

While the static *high watermark* setting for metadata in the L3, discussed in Section 3.3.3, and the standard L3 replacement policy can help dynamically balance these competing factors, this data gives indication that opportunity remains for a more intelligent mechanism to adapt runtime policies regarding Tag Table metadata in the L3. Therefore, in addition to the *high watermark* setting which protects application data from metadata pollution, this

chapter will investigate a *low watermark* setting which protects metadata from application data. Further, it will explore the ability of Set Dueling to accurately adjust these settings to dynamically adapt the balance of metadata and application data in the L3.

Previous work in Set Dueling has shown that it can be very effective at making binary decisions, selecting between competing policies in the cache [43, 44], therefore it should be capable of choosing between two static *high* and *low watermark* setting pairs (i.e., a metadata-centric policy and a application-centric policy). However, it would be advantageous for a Tag Table system to do more than simply chose between two levels of metadata pollution, therefore this chapter will not only evaluate the ability of Set Dueling to select between two watermark setting configurations, but propose a novel adaptation, called Set Scouting, to allow it to modify policies over a range of values.

4.1 Background

This section presents prior work on Set Dueling to include its initial proposal along with follow-on work that extended it to its current state. This provides background knowledge on prior implementations and use cases for understanding how it can be adapted to balance Tag Table metadata in an L3 cache.

4.1.1 Sampling Based Adaptive Replacement

A Set Dueling approach was first proposed and applied as a mechanism for dynamically adapting caching policies in a system advocating a memory-level parallelism (MLP) aware

replacement algorithm [44]. In this system, Set Dueling is used to select between a default least recently used (LRU) cache replacement policy and *linear* (LIN), that incorporates memory-level parallelism metrics to determine the victim when inserting new data. In this approach, it is advocated that isolated misses in the cache - or misses that cannot be overlapped with others - are relatively more important to avoid than misses that occur with multiple others. In order to quantify the level of isolation any particular miss encounters, the authors introduce an *mlp-cost* metric that quantifies the number of additional misses that the cache was servicing while the miss-under-investigation was active in the cache's miss status handling register (MSHR) waiting for its data and the time required to receive that data. The LIN policy then determines a victim through a linear function of *recency* (the relative location of the block in the LRU chain) and *mlp-cost*.

Specifically, *mlp-cost* is calculated in this system while the request is present in the cache's MSHR waiting for data. Every cycle, the system increments an *mlp-cost* field in every active entry in the MSHR, inversely proportional to the number of active entries. For instance, if there is only one active entry in the MSHR, the *mlp-cost* field of that entry will be incremented 4x as much as it would be if it was active with three others. Upon receipt of the data and insertion into the cache (causing the MSHR entry to become inactive), the tag of the block is appended with an additional 3-bit quantized value corresponding to the *mlp-cost* incurred while waiting for the fill. In this way, misses that are satisfied with many others and/or are satisfied quickly will have a lower value for their *mlp-cost*. Then, when it comes time to choose a victim they are more likely to be evicted to the point that

they may not have to travel to the end of the LRU chain before the mechanism determines they are less valuable to keep. Isolated misses, on the other hand - those misses that were not satisfied along with others - will have higher values for *mlp-cost* and be less likely to be replaced.

As with many replacement proposals, while LIN works well for a portion of the studied applications, certain others are negatively affected. Those applications which do not have stable MLP characteristics - where the likelihood of a given block incurring a miss with others varies over time - are those that prefer the standard LRU policy.

Therefore, the authors propose "Sampling Based Adaptive Replacement" (SBAR) as a lightweight mechanism to dynamically adjust the cache policy between LRU and LIN. In this system, some subset of the cache's sets are chosen as *leader* sets and augmented with an additional tag directory and a saturating counter called the "tournament selector" (TSEL). The additional tag directory acts as a sort of shadow directory and implements LRU in order to track the number of hits an LRU-based set would incur while the data of the set is actually controlled by the main directory which implements LIN. The remaining sets in the cache are *follower* sets and implement a replacement policy as determined by the MSB of the TSEL.

The TSEL is updated on every access to a *leader* set. When one directory (either the primary LIN directory or shadow LRU directory) results in a hit and the other a miss, the tournament selector is incremented (in the case of a LIN hit) or decremented (in the case of an LRU hit) by the value of *mlp-cost* when the miss completes. In this way, the TSEL is

updated relative to the *costs* of the misses of the two policies and the MSB determines the replacement policy followed by the *follower* sets. If the TSEL MSB is '1' they implement LIN, otherwise they implement LRU.

4.1.2 Dynamic Insertion Policy

A second paper, advocating adaptive insertion policies, extends Set Dueling to its current state-of-the-art [43]. Set Dueling is again utilized to select between two competing cache replacement policies but this time, as opposed to affecting victim selection, the proposed policy modifies insertion policies, advocating placing newly inserted blocks into positions other than most-recently used (MRU) in the LRU chain, a position that avoids thrashing in streaming workloads or workloads otherwise too large to fit in the cache. With this "Bimodal Insertion Policy" (BIP) in effect, incoming blocks are usually placed in the LRU position (with a very small, random probability otherwise) and only promoted to MRU if there is a subsequent re-reference before eviction.

Not surprisingly and as with the previous proposal, BIP is not a universally good match for all applications. Indeed BIP is primarily advantageous for a relatively small set of applications - those that exhibit either streaming behavior (i.e., no re-use) or have working sets that cannot fit in the cache. Therefore, the authors utilize Set Dueling to dynamically select the proper global policy for the cache with a "Dynamic Insertion Policy" (DIP), that selects between BIP insertion and MRU insertion dependent on the characteristics exhibited by the running application.

The major adaptation to Set Dueling in this work relative to SBAR is to do away with the shadow directory and instead identify two competing collections of cache sets (called the *dedicated sets*), each statically configured for one of the two policies. As long as the number of *dedicated sets* configured for the lesser-performing policy is not too great, this rigid configuration is not likely to adversely affect performance. Of course care must be taken to make sure that *enough* sets are assigned to each competing policy so as to make an accurate decision on the global policy. The paper performs an analytical evaluation to determine the correct number of sets utilizing the *central limit theorem*, concluding that 32 to 64 sets are sufficient to provide reasonable confidence (>95%) that the best global policy is selected. This result is notably independent of the size of the cache. Instead, the main determining factor is a ratio between the average difference between sets implementing the two policies and standard deviation exhibited by the *dedicated sets*.

In this system, instead of a single set tracking both policies, the cache is broken up into multiple *constituencies*, with each *constituency* having one *dedicated set* assigned to each competing policy and the remaining sets following the "Policy Selector" (PSEL). The PSEL is equivalent to the TSEL in SBAR with the exception that it is merely incremented/decremented by one to track the raw difference in hits between the two dedicated - and competing - sets in the *constituency* and does not incorporate any MLP cost.

In order to assign sets to *constituencies* and identify the dueling sets within the *constituencies*, DIP utilizes a *complement-select policy*. With this policy, the most significant $\log_2(K)$ bits (where K is half the number of *dedicated sets*) serve as the *constituency ID* while

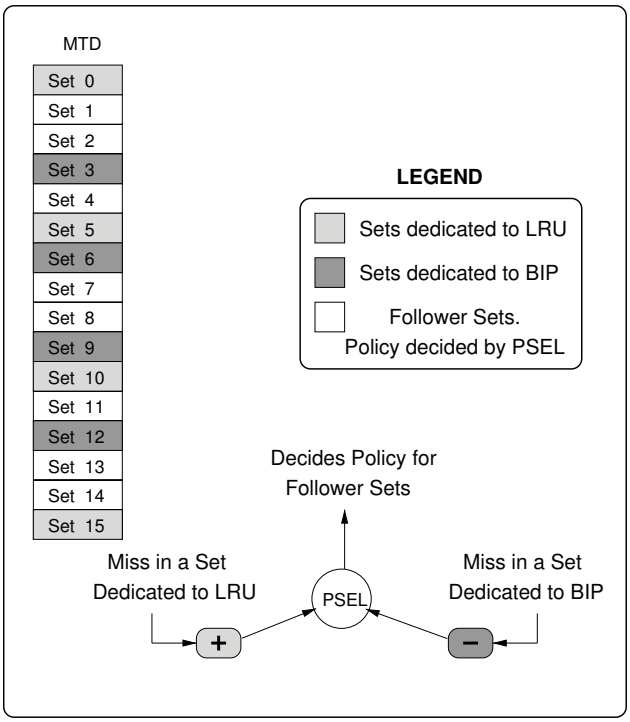


Figure 4.1: Example operation and set assignment in DIP (Taken from [43]).

the least significant $\log_2(N/K)$ bits identify the *constituency offset* (where N is the number of sets in the cache). Sets whose *constituency offset* bits equal their *constituency ID* bits are then assigned to implement LRU, while those for which the *complement* of their *constituency offset* bits are equal to their *constituency ID* bits implement BIP. By way of example, the paper considers a cache with 1,024 sets with 64 *dedicated sets* (32 for BIP, 32 for LRU). Since there are 10 set selection bits ($\log_2(1,024)$), the upper 5 bits ($\log_2(32)$) determine the *constituency ID* while the lowest 5 bits determine the *constituency offset*. This leads to the situation where Set 0 and every 33rd set are LRU and Set 31 and every 31st set are BIP. Figure 4.1 provides a simple example for a 16-set cache.

4.2 Application to Tag Tables

Applying Set Dueling to Tag Tables utilizes distinct *high* and *low watermark* settings as the competing policies, allowing metadata occupancy in the L3 to provide feedback as to the best decision. While the *high watermark* setting of the baseline Tag Table proposal is shown to be effective at limiting the occupancy of metadata in the L3, certain workloads, particularly when the entry size is increased by tracking more chunks as shown in Figure 3.10b, can benefit from consuming more of the L3 in order to track more data in the DRAM cache.

In order to understand those situations where increased L3 metadata occupancy can be advantageous, it is first important to review application working sets and their affect on various levels in the cache hierarchy. As Jaleel investigates in depth for the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites, applications can exhibit several degrees of working set size [26]. This can be seen when simulations sweep the cache size parameter and the cache miss rate is observed to decrease rapidly in response to some larger cache size, creating a "knee" in the graph. For instance *wupwise*, in Figure 4.2 exhibits one knee around 4MB indicating the primary working set is approximately 4MB in size, which could correspond to a frequently executed inner loop. A second knee however can be seen around 32MB, indicating a secondary working set that could correspond to an outer loop. Therefore, one situation where increased L3 pollution can be advantageous is when it allows a secondary working set to fit in the DRAM cache as long as a primary working set is not prevented from fitting in the L3 in response. For example, if the primary working set size of *wupwise* of 4MB plus the metadata pollution is still less than the L3 cache capacity,

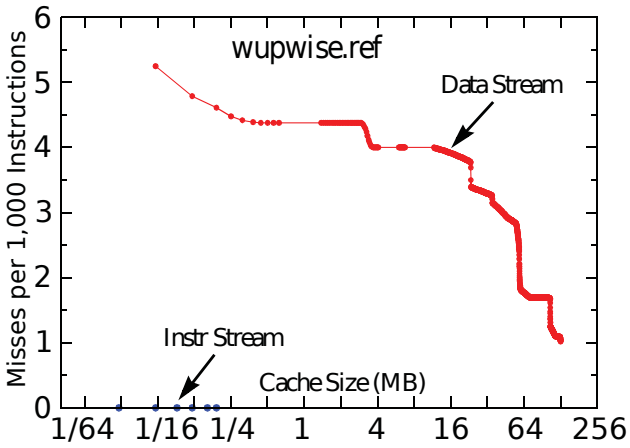


Figure 4.2: Visible knees in cache miss rate as capacity increases indicates various working set sizes (Taken from [26]).

application performance will not be negatively impacted. Beyond that, if the increase in pollution results in the DRAM cache tracking more data such that it now covers 32MB, application performance will be improved since the secondary working set will be available in the relatively fast DRAM cache (with respect to main memory) and requests will have to go off-chip less frequently.

Therefore, a new mechanism is introduced to the Tag Table, the *low watermark*, which protects Tag Table metadata in the L3 cache by preventing application data from evicting metadata below a certain value per set. While there is no requirement for the set to allocate a minimum storage for metadata in the case where the Tag Table does not require the *low watermark*, it does prevent insertion of new application data from evicting a metadata block - even if it is in the LRU position - if such an act would cause the metadata occupancy to drop below the *low watermark*. Instead, the non-metadata block in the set closest to the LRU position would be chosen for eviction.

Policy	*Low Watermark*	*High Watermark*
Application-Centric	0	1/4 ways
Metadata-Centric	1/4 ways	1/2 ways

Table 4.1: Tag Table L3 metadata dueling policies.

4.2.1 Set Dueling

Similar to LIN and BIP, for reasons mentioned above related to working set sizes, the enforcement of a *low watermark* will not be universally advantageous, nor is a set value for the *high watermark*, for that matter. Therefore, Set Dueling is proposed to achieve dynamic adjustment of these values based on cache performance. Since the baseline Set Dueling approach has been shown to work well at deciding between two opposing policies, the first approach is to apply it to select between two *low* and *high watermark* pairs - a metadata-centric policy and an application-centric policy. These two policies are outlined in Table 4.1 and quantify the fraction of the ways in a set that can be allocated to metadata. For instance, given the 16-way set associative L3 cache assumed for our baseline, the metadata-centric policy would enforce a *low watermark* setting of 4 and *high watermark* setting of 8. This means that no *more* than 8 block locations in any given set may be occupied by metadata and also that application data may not evict metadata to *fewer* than 4 blocks.

Beyond the specific policies of interest, Tag Tables present one major deviation from previous proposals utilizing Set Dueling. In previous proposals, policies were being selected that resulted in the best operation of the cache implementing the dueling. In other words, a given cache was comparing the performance of the dueling sets to determine which one was resulting in the fewest or least costly misses and chose that policy globally

to optimize the overall cache hit rate. In Tag Tables, Set Dueling is instead used to select the best policy for the *system*, which could actually degrade the L3 hit rate. In fact, increasing metadata in the L3 will never improve the performance of the L3, it can only harm. Set Dueling can therefore help in the situations where the harm to the L3 of increased metadata occupancy is outweighed by its benefit to the system (i.e., the improvement in the DRAM cache hit rate).

How to quantify the "benefit to the system" itself is a challenge. While it is a simple matter to count hits between dueling sets in prior applications, it is not clear what is best for Tag Tables. Although a mechanism could be envisioned which incorporates DRAM cache hit rate feedback at the L3, as an initial implementation, a simpler self-sacrificing mechanism was implemented instead that requires no external information. In this self-sacrificing design, the global policy is opportunistically selected as the high (metadata-centric) policy if it does not significantly harm L3 hit rates. In other words, even if the sets implementing the metadata-centric policy performed worse, it would still be chosen as the global policy if it was not *too* much worse, therefore the L3 would choose to sacrifice its own hit rate in the hopes that it would aid the DRAM cache.

Beyond the concessions necessary for lack of system-level feedback, Set Dueling in Tag Tables is consistent with the system utilized by DIP as describe in Section 4.1.2. Specifically, some number of *dedicated sets* are assigned to implement the competing policies, half to the application-centric policy and the other half to the metadata-centric policy. Given the large number of sets in the DRAM cache configurations (from 16K for a 64MB DRAM cache up

to 256K for a 1GB DRAM cache), the following evaluation conservatively selects the upper bound from the analytical evaluation in [43] - 64 sets - to be configured as *dedicated sets* - 32 to the application-centric policy and 32 to the metadata-centric policy.

4.2.2 Set Scouting

Unlike application in BIP and DIP however, applying two distinct policies to the Tag Table for Set Dueling is somewhat arbitrary since there are 153 possible combinations of high and *low watermark* values[1]. Therefore this section extends traditional Set Dueling by proposing and evaluating a set "scouting" policy.

Set Scouting may be best described by way of analogy. Consider the situation of a caravan, flanked on either side by scouts. The job of the scouts is to find the best path for the caravan, so they ride along trails on their assigned side. Over time, the scouts can compare the path they are traversing relative to the caravan to determine if they are on a better path. If so, the caravan can transition to the new path, the scout that found the new path moves farther off to maintain about the same distance from the caravan, and the scout on the opposite path moves somewhat closer in the same direction, not wanting to be too far from the main group. In this way, all three categories of traveler (each of the scouts and the main caravan) can give feedback to the others on the ease of their path.

We can implement a similar approach in a cache by adapting Set Dueling such that the *follower sets* (the caravan) do not have the same policy as either of the *dedicated sets* (the

[1]Considering either setting can take a value in the range of 0 to 16 and that the *high watermark* can be no less than the *low watermark* for any configuration

scouts) and by also tracking the effectiveness of the follower sets' policy by assigning one follower set from each *constituency* to be the *follower representative*. In order to compare the *follower sets* effectiveness versus the scouts, an additional PSEL is introduced such that the *follower representatives* can compare their performance relative to the low scouts and the high scouts separately. Then in operation, misses to the *follower representative* decrement the PSELs and misses to the scouts increment their respective PSEL, therefore once a PSEL MSB becomes '1', a policy transition is triggered. This transition differs from traditional Set Dueling in that it does not merely change the policy of the follower sets, but also changes the policy of the scouts to ensure that scouts are always exploring different settings from the followers. During this transition, the follower sets adopt the policy of the scout that triggered the transition (unchanged from traditional Set Dueling), the scout that triggered the transition moves to a more extreme policy (by halving their distance from their most extreme setting), and the remaining scout modifies their policy in the same direction as the others. Specific modifications to each category's policy will be described by way of example in the following paragraph. In addition, following a transition, the PSELs must be reset in such a way as to prevent thrashing (i.e., rapidly bouncing between policies due to noise immediately following a transition). Therefore, the PSEL reset point is chosen to be 1/4 the maximum value, thereby giving a buffer before either scout can trigger another transition, making it more likely to indicate a valuable transition, not just noise immediately following a prior transition. Further, the policy of the scout sets are selected such that they do *not* match the policy of the followers. This is contrary to traditional Set Dueling where the

global setting matches one of the two competing policies.

For example, consider initialization of policies such that the scout sets implement the competing policies in Table 4.1 (consistent with binary Set Dueling) while the follower sets are initialized to a midway point. While the exact specification of the midway point is arbitrary (and could even be more extreme than one or both of the scout policies), the evaluation in Section 4.3 will assume an initial *low watermark* setting of '0' and *high watermark* setting of 1/2 the set's ways for the follower sets. Then during operation, if one of the scouts causes their PSEL MSB to become set a transition is triggered. Specifically, if the low scout sets its PSEL MSB first following initialization, the follower sets will transition to its policy (i.e., the *application-centric* policy the low sets were initialized with). The low scout, which triggered the transition, will halve its distance to its most extreme setting. In the case of the low scout, the extreme setting is '0', therefore the *low watermark* remains unchanged (it is already '0') and the *high watermark* becomes '2' (halfway between the initial '4' and '0'). The high scout will move in the same direction as the followers and the low scout (i.e., they will *reduce* the value of both of their watermarks) by modifying them to be halfway between their current setting and the follower sets' new setting. Therefore the high scout settings will be modified to be '2' for the *low watermark* setting (halfway between its current value of '4' and the follower sets' '0') and their *high watermark* setting will become '6' (halfway between '8' and '4'). Finally, both PSELs in a *constituency* are reset to 1/4 of their maximum value (e.g., 512 for an 11-bit PSEL, consistent with the PSEL size evaluated in [43] for 64 dueling sets).

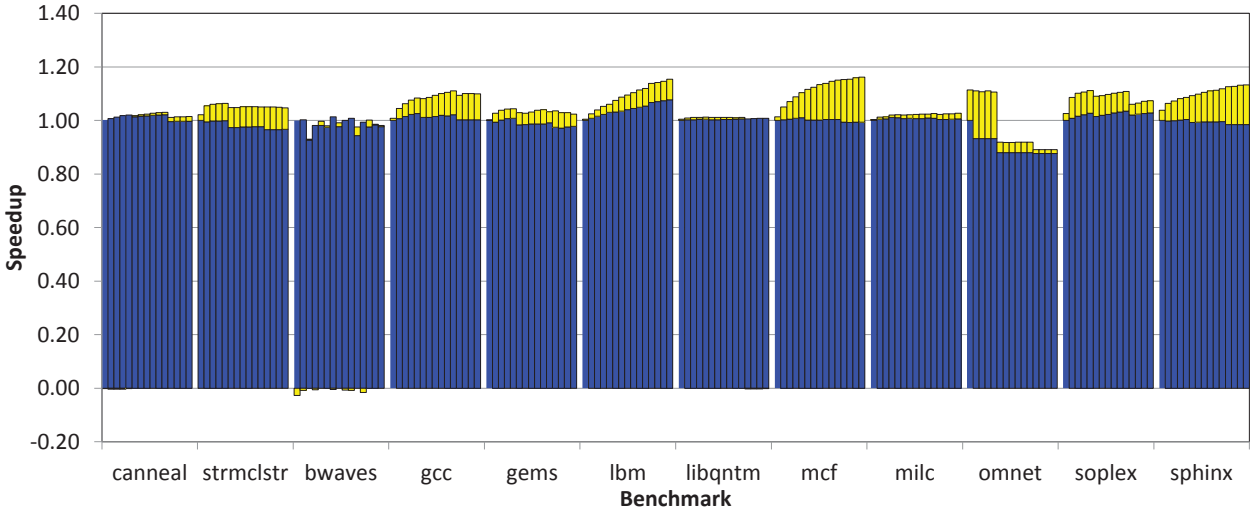


Figure 4.3: Results of binary Set Dueling with varying number of chunks per entry. Benefit of Set Dueling are the top portions of the bars.

4.3 Evaluation

4.3.1 Dueling Evaluation

Initial evaluation assumes traditional, binary Set Dueling, the results of which are shown in Figure 4.3 for a range of entries that can track from 2 to 16 distinct chunks as evaluated in Section 3.5.5. This stacked bar chart reflects the speedups achieved for various chunk numbers when increased entry size was taken into account along with the additional speedup on top due to Set Dueling. As can be seen, Set Dueling can lead to non-trivial improvement over the baseline system. For applications such as *mcf* and *sphinx* in particular, Set Dueling achieves the expected outcome of having a greater effect on performance as the entry size is increased. Due to the increased L3 capacity pressure of these larger entries, it is relatively more important to include a control mechanism in the amount of pollution they are allowed to create.

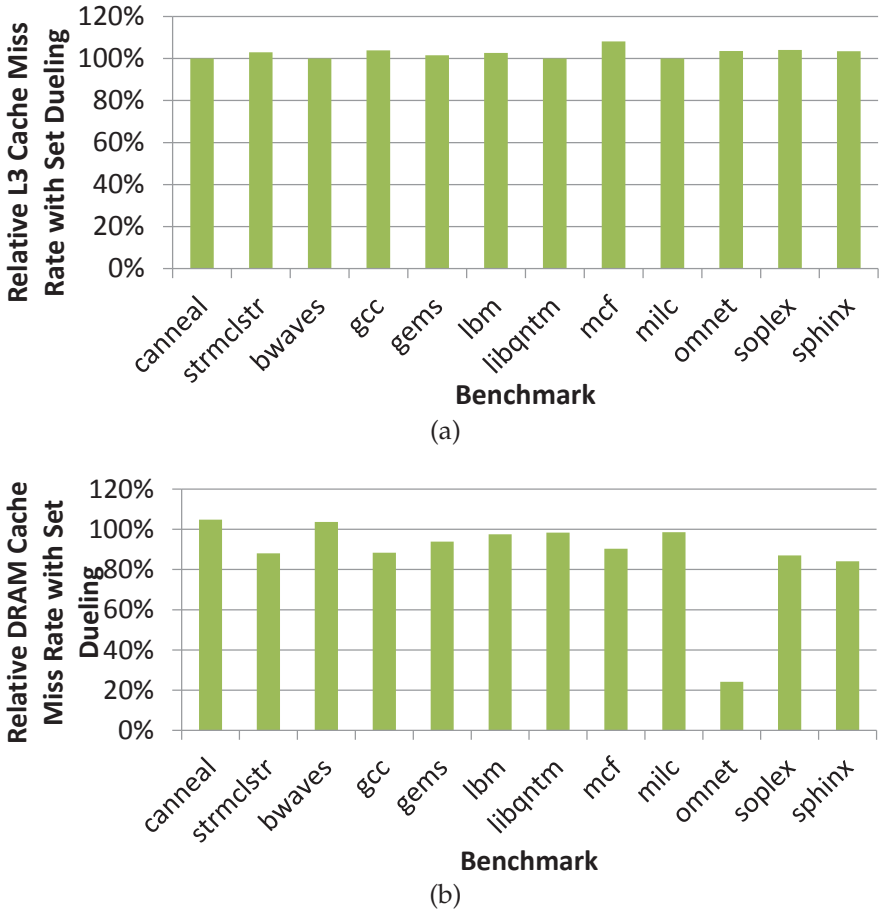


Figure 4.4: Miss rate effects of Set Dueling on the (a) L3 and (b) DRAM Caches.

The workload *omnet* is an interesting outlier in these results, not only in the fact that it exhibits the highest improvements with Set Dueling when there are few chunks - specifically from 2 to 6 - but also in the fact that they achieve relatively more improvement when *fewer* chunks are supported, not more. This is due to the significant improvement in DRAM cache hit rate at these low chunk number configurations as shown in Figure 4.4b which is not present when there are more chunks due to the fact that the entry size increases once the seventh chunk is added. This increase in entry size leads to the workload switching to a metadata protection policy for a lesser proportion of the runtime.

Beyond giving indication to the performance of *omnet*, Figures 4.4a and 4.4b highlight the mechanism by which Set Dueling affects Tag Table performance, namely that it allows L3 capacity to be traded for DRAM cache capacity. Given the implementation of Set Dueling in Tag Tables, specifically that the default policy is to not protect metadata in the L3, the only way that it can affect system performance is by sacrificing L3 hit rate for DRAM cache hit rate (i.e., there is not a scenario where it can improve *L3* hit rate). Therefore, the lack of improvement in any L3 hit rates is expected and the negligible degradation occurs only with applications that exhibit an improvement in DRAM cache hit rate, indicating the L3 sacrificed some of its own performance for the DRAM cache. Also due to this self-sacrifice and lack of actual feedback between the L3 and the DRAM cache, the figures further show the situations such as *canneal* and *bwaves* where there is actually a slight degradation in L3 hit rate without a subsequent improvement in the DRAM cache hit rate. This corresponds to almost imperceptible degradation in overall performance for those benchmarks in Figure 4.3 in the 4-chunk configurations (approximately 0.5% slowdown for both).

Overall, these figures indicate that Set Dueling can have a non-negligible effect on the performance of Tag Tables by adding additional control at the L3 for adapting metadata occupancy.

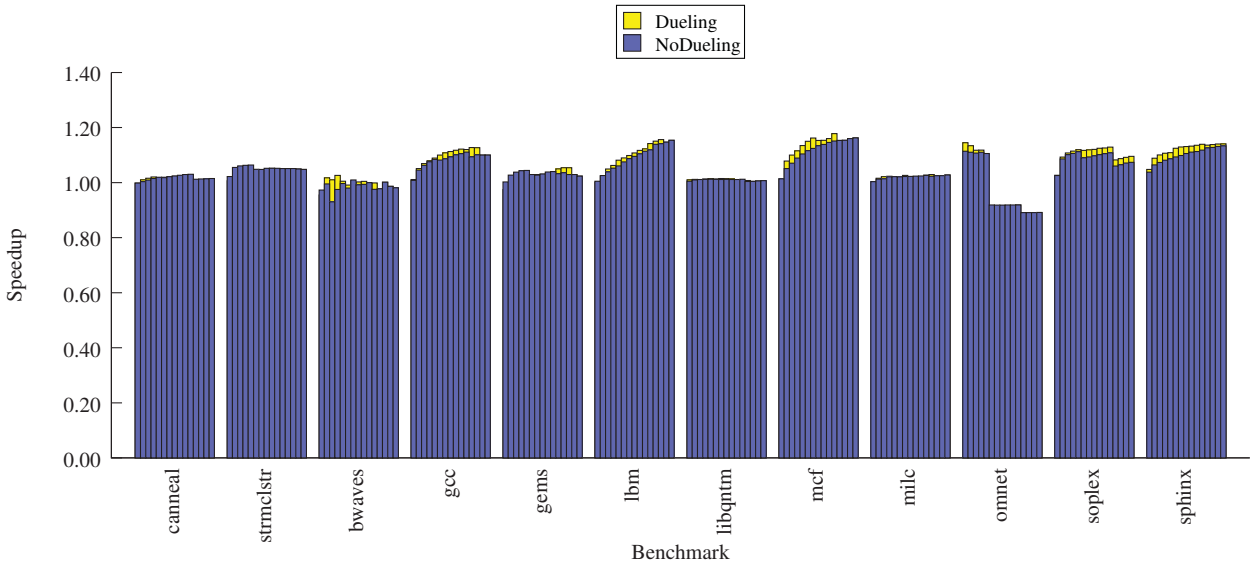


Figure 4.5: Results of Set Scouting with varying number of chunks per entry beyond that achieved with traditional, binary Set Dueling.

4.3.2 Scouting Evaluation

Figure 4.5 presents the additional benefit possible by implementing Set Scouting over Set Dueling. Beginning with a baseline of Set Dueling - the bottom portion of the stacked bars - the top portion of the stacked bars represent the additional performance achieved with Set Scouting. As expected, there is additional benefit when the L3 is allowed to select from a range of values for the low and high watermark settings, though it is relatively small ranging up to near 5% in the most extreme case. Further, the L3 and DRAM cache hit rates are relatively unchanged except for small improvements consistent with those workloads that achieved relatively more performance with Set Scouting relative to Set Dueling.

4.4 Summary

This chapter investigated the opportunity of adapting Set Dueling concepts to improve Tag Table performance. Due to the decision to store Tag Table metadata in the L3 cache, a careful balance needs to be struck between L3 capacity for application data and capacity for Tag Table metadata. The standard LRU replacement policy of the cache accomplishes this goal to a point by evicting metadata that has not been accessed for a long time but the long residency time of blocks in a DRAM cache means that useful data can be retained for a significant amount of time without being accessed and LRU can cause useful metadata to be excessively evicted. Therefore, this chapter first presented a configurable low watermark mechanism to protect metadata from eviction in the L3 despite not being referenced often. However, since applications can vary significantly in terms of how relatively important L3 capacity is versus DRAM cache capacity, this chapter further proposed an adaptation of Set Dueling to dynamically adjust both watermark settings in response to runtime characteristics and found that it can achieve significant improvement up to 17% with an average of 5% over all benchmarks with generally more opportunity when entry sizes are larger due to tracking more distinct chunks. Finally, given the range of watermark values that can be selected, a novel extension to Set Dueling, Set Scouting was proposed that allows modification of policies over a range which can extend the benefit of Set Dueling by a few additional percentage points for certain applications.

5 Translation Caching

This chapter discusses techniques for limiting the number of levels traversed in a forward page table structure. These techniques not only include relatively small translation lookaside buffers (TLBs), but also the caching of pointers to intermediate levels of the table in order to accelerate walks after a TLB miss. Given its page table-based structure, these techniques are naturally applicable to Tag Tables, thus this chapter evaluates their opportunities for accelerating the tag check process of a Tag Table.

5.1 Key Concepts

Before discussing existing techniques for accelerating page walks and how they may be applied to Tag Tables, it is important to review a few fundamental concepts related to address translation. The following sections serve to present a brief overview on the operation of TLBs and translation caches along with a walkthrough of a page table walk in order to define terminology for subsequent sections and ensure understanding of foundational concepts.

5.1.1 Translation Lookaside Buffers

A TLB is a map structure accessed first in the virtual-to-physical address translation process where the key is a virtual page number and the associated value is a physical page number. A TLB caches the recently translated mappings in a relatively small table, allowing it to quickly provide the virtual address translation associated with a memory operation without incurring the very long latencies associated with a page table access. This is critically important for allowing L1 cache accesses to complete quickly since a full walk of the page table requires multiple memory accesses. This is due to the fact that the metadata that composes a page table exists in memory itself, therefore the page walk requires accessing memory for every intermediate level of the table.

In current latency-optimized designs, L1 caches are implemented as virtually indexed, physically tagged (VIPT), necessitating a virtual address translation, and thus TLB access, prior to checking tags but allowing set activation to occur in parallel with the translation. This means the proper set - or index - can be identified and activated in the cache using bits of the virtual address without having to wait for the translation to complete. However, the translation must still finish in no more time than it takes to activate the set, providing the proper bits for the tag check, to avoid becoming a bottleneck. This, in turn, provides a constraint on the acceptable latency for a TLB lookup which subsequently places a limit on the size of a TLB.

This size and latency interplay involves an important tradeoff. In addition to increasing the latency to perform a check, increasing the size of the TLB also increases its "reach" [49].

This "reach" refers to the effective range of memory that the translations can provide mappings to, so while a smaller TLB may be desirable for cache access purposes, limiting the reach may result in more TLB misses, necessitating a long-latency page walk. In the case of a 256-entry TLB for 4KB pages, the reach corresponds to 1MB. Therefore, any given TLB in a system which may be composed of multiple cores - each having their own TLB - can only map a subset of current last-level caches (LLC) which are typically multiple megabytes.

Such latency/reach tradeoffs have led to various compromises in TLB design that mirror the sorts of compromises seen in general caches including the use of multiple levels of TLB, set-associative versus fully-associative design points, as well as supporting multiple granularities of translations (i.e., super pages or huge pages) which are roughly analogous to cache block sizing tradeoffs.

Fundamentally, the effectiveness of a TLB at preventing address translations from stalling processor operation lies in the rate at which it hits, thus providing the appropriate tag bits in time for the L1 tag check. In addition to the reach previously mentioned, the locality of reference present in applications plays a major role in this hit rate. Specifically, if an application accesses several data locations in each page mapped by the TLB - in other words, it has high access density - it will be much more effective than if the application sparsely accesses the mapped pages. While some noteworthy exceptions exist (such as large databases), many applications exhibit high access density with respect to TLB entries, leading to high temporal locality. Further, following the first access to a block in a page,

many subsequent blocks in the same page are likely to be referenced (spatial locality), allowing a single entry in the TLB, allocated by one access, to satisfy the translations for many following requests.

5.1.2 Translation Caching

Beyond opportunities for fast translations provided by TLBs, recent designs from both Intel and AMD have incorporated structures to accelerate the page walks incurred on a TLB miss [6, 25]. These structures both allow the page walk hardware to begin the walk at some intermediate level in the page table by "skipping" the root and potentially other intermediate levels. Due to the high temporal locality in upper levels of the page table, these caches can be especially effective at skipping the initial levels, while their effectiveness is somewhat reduced for the lower levels.

Figure 5.1 provides a diagram of the high-level operation of a translation cache. In this example, a translation that misses in the TLB, which would traditionally begin performing a page walk by indexing into the root level of the page table, now first accesses the translation cache. If recent accesses have traversed shared high level paths, the translation cache will have pointers to intermediate nodes allowing the walk to start at them rather than the root. In this example, the request is able to skip the root and second level of the page table and directly access the third level because previous walks have already traversed them, reducing the page walk to two traversals.

While Intel's Paging-Structure Caches [25] and AMD's Page Walk Caches [6] express

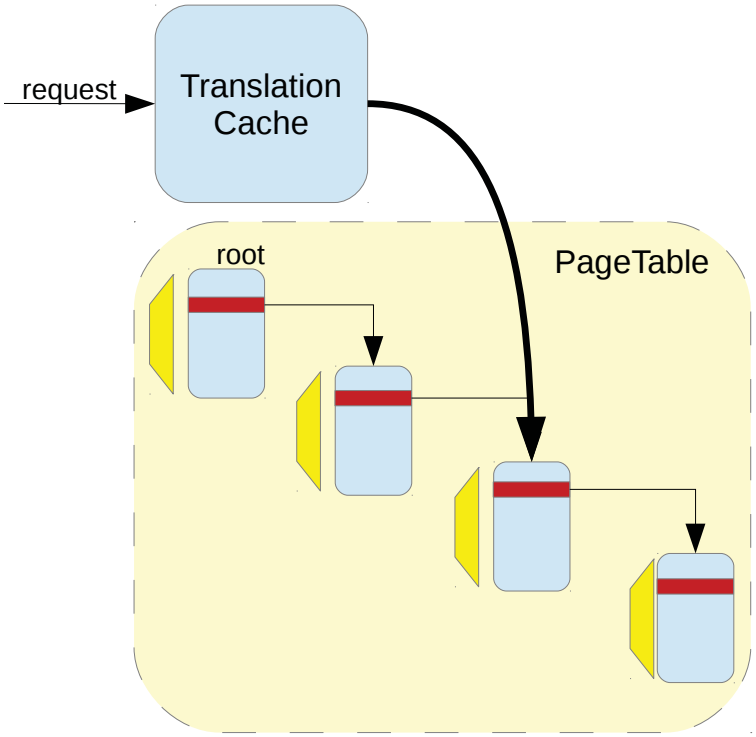


Figure 5.1: High-level overview of translation cache operation.

distinct points in the design space, Barr, et. al. generalize the concept, referring to the structures as "translation caches," and enumerate additional points [5].

5.1.3 Page Walk Primer

Before continuing with examples of specific translation cache designs, a brief background on the steps involved in translation is in order. Much of the operation will be familiar from the description of a Tag Table walk, but particular details are important for understanding specific characteristics of various translation cache designs.

In current x86-64 processors, a 48-bit virtual address consists of the page offset - which for 4KB pages consists of the bottom 12 bits - and the virtual page number (VPN) - the

remaining 36 upper bits. The goal of translation then consists of determining the physical page number (PPN) associated with the VPN. Following translation, this PPN is concatenated with the page offset to identify the precise location of the data in the memory.

As mentioned previously, since the metadata that comprises a page table itself is stored in memory, this translation actually requires performance of multiple memory accesses. Translation of this VPN to a PPN consists of traversing - or "walking" (hence the term "page walk") - four levels of page table, each level indexed by 9 bits of the VPN, in decreasing order (e.g., the first level is indexed by the most significant 9 bits of the VPN). Each of the steps in this walk then correspond to the acquisition of a block of data from the memory hierarchy which contain some portion of the metadata that comprises a level in the page table.

Using the translation of virtual address 0x5c8315cc2016 shown in Figure 5.2 as an example, the first memory access is triggered by the need to access the portion of the top-level table ("L4" in the figure) which will give the pointer to the appropriate L3 table. The memory location for this access is determined by concatenating the PPN of the top-level (or root) table, which is stored in the CR3 register, with the top 9 bits of the VPN ("0b9" in the figure). The data returned from this access (the PPN "042" shown in entry "0b9" of the L4 table in the figure) in turn, provides the base address of the next level table in the tree that is on the path to the final translation (the "L3" table shown). Translation then continues by appending the next 9 most significant bits of the VPN to this PPN until all bits of the VPN are exhausted or a "NUL" entry is located. Similar to a Tag Table, "NUL"

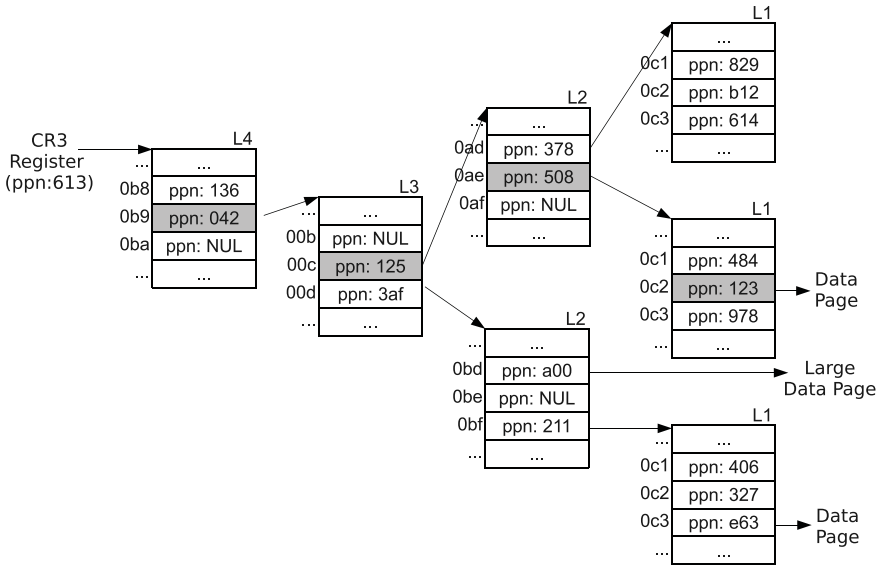


Figure 5.2: Page walk example for 0x5c8315cc2016 (from [5]).

indicates no translation exists for the VPN. If instead, the walk terminates in an L1 entry with valid data, the data corresponds to the desired PPN and the walk completes.

As with generic application data, data acquired from a page table walk can be cached in certain designs. For instance, one commercial example allows page table metadata to be cached in the L2 [7], accelerating subsequent walks involving this path.

5.2 Background

5.2.1 AMD's Page Walk Cache

The acceleration achieved by obtaining page table data from the L2 cache is the motivating factor behind AMD's Page Walk Cache. Rather than have page table metadata compete with application data in the L2, the Page Walk Cache serves as a private cache just for this

Base Location	Index	Next Page
125	0ae	508
042	00c	125
613	0b9	042
...	...	...

Figure 5.3: Sample entries in a Unified Page Table Cache (UPTC), the structure exemplified by AMD's Page Walk Cache (from [5]).

metadata. Categorized as a "Unified Page Table Cache" (UPTC) by Barr, et. al., this structure, shown in Figure 5.3, stores the 40-bit PPN along with the 9-bit index corresponding to a recent access on the path of a page walk. As the figure shows, again for an access to 0x5c8315cc2016, the access to the root, consisting of PPN 613 (from the CR3 register) concatenated with the index 0b9 led to the PPN 042 for the following level of the table. This, in turn, combined with index 00c, led to PPN 125 for the third level of the table and so on until the translation was complete. With this information, subsequent accesses which walk any portion of this path will be able to quickly locate metadata on the path by merely accessing this small, fast, dedicated cache without accessing any portion of the system memory hierarchy.

5.2.2 Intel's Paging-Structure Caches

Unlike the Page Walk Cache, Intel takes a "path" approach to translation caching with their Paging-Structure Cache, accessing only with bits of the virtual address. Further, instead of creating a single structure for entries associated with all levels of the page table (a "unified" approach), the Paging-Structure Cache implements a "split" approach by utilizing separate

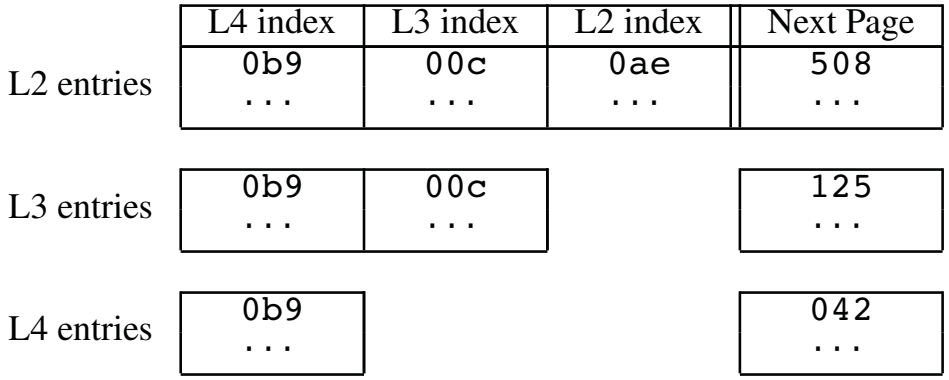


Figure 5.4: Sample entries in a Split Translation Cache (STC), the structure exemplified by Intel's Paging-Structure Cache (from [5]).

structures for each level. Shown in Figure 5.4, Barr, et. al. define this as a "Split Translation Cache" (STC). In this scheme, each structure is indexed in parallel using the appropriate bits of the virtual address (e.g., the top three 9-bit chunks access the "L2 entries" cache). When complete, a priority select mechanism chooses the valid "Next Page" result furthest down the tree by emphasizing hits to the "L2 entries" cache and working up, effectively performing a "longest prefix match" operation by selecting the longest path.

5.2.3 Translation Path Cache

The longest prefix match operation utilized by the STC is the same operation utilized by the Translation Path Cache (TPC) defined by Barr, et. al. as a translation cache design not exemplified by any commercial offering. The TPC combines the unified storage structure of the UPTC along with the virtual address indexing of the STC to create a fast, space efficient structure by storing full translation paths as keys and multiple columns representing the related PPN's into the page table indicated by the associated prefix, as shown in Figure 5.5.

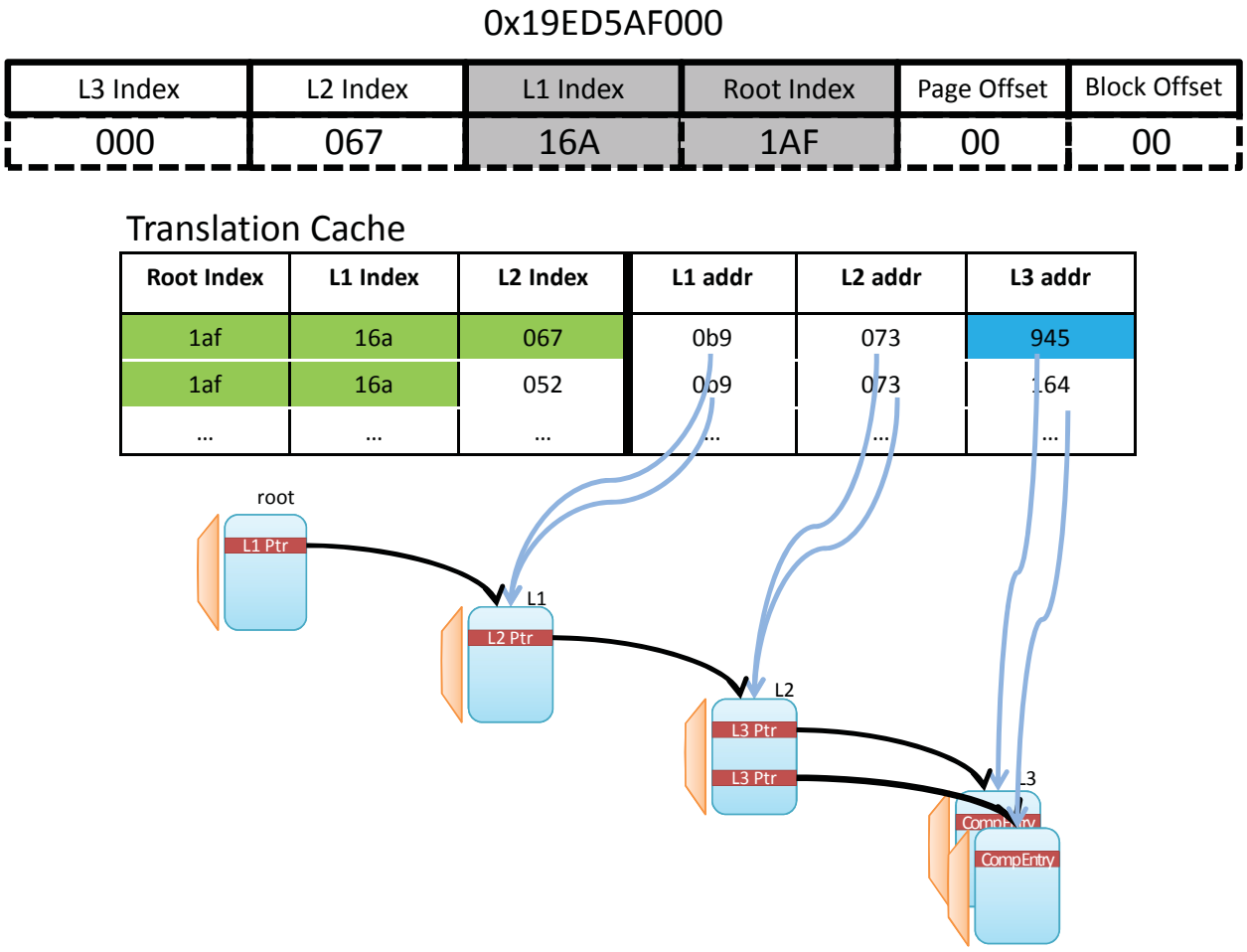


Figure 5.5: Example operation of the translation path cache. Longest prefix matching returns a pointer to a leaf entry (highlighted result in 'L3 addr' column) for the access, bypassing the three intermediate levels.

As an example of operation, Figure 5.5 illustrates an access for address 0x19ED5AF000, with its constituent bits displayed in the appropriate fields. Although potentially multiple entries can provide pointers to intermediate page table levels - as shown by both entries in the figure - the TPC selects that entry and the column corresponding to the matched length of the prefix as the output of the translation cache, allowing the walk to start at the farthest level possible.

5.3 Application to Tag Tables

Application of the preceding concepts related to TLBs and translation caches represent several distinct opportunities for Tag Tables. First of all, the operation of a TLB can be easily adapted to Tag Tables by similarly implementing a table structure that is queried at the beginning of an access and returns the location of data in the DRAM cache if it or its neighboring blocks have recently been accessed. This page offset lookaside buffer (POLB) can be realized in a Tag Table by mapping an address to a compressed entry. This physical-address-to-entry mapping is therefore analogous to a traditional TLB's virtual-address-to-physical-address mapping. While it could be envisioned that an access to the TLB be at a block granularity as opposed to an entry (i.e., an index consists of the entire address and the value is a single block offset), storing a large entry is similar to the large region a page encompasses for a TLB, which is crucial for high performance operation. Further, in order to make accesses fast the lookaside buffer should be set associative, as current TLBs.

Second, regarding adaptation of translation caching, any of the preceding designs - or any of those in Barr, et. al. that were not mentioned - could conceivably be selected for accelerating Tag Table walks. Due to the space efficiency of the Translation Path Cache however, and the fact that Barr, et. al. reveal relatively little practical performance difference between the various designs, the evaluation of this chapter will explore the application of the Translation Path Cache to Tag Tables. Adaptation of such a structure is straightforward to Tag Tables, with appropriate bits of a request's address being used to index the cache as

shown in Figure 5.5.

5.4 Evaluation

This section evaluates the implementation of a page offset lookaside buffer (POLB) and a translation cache on the performance of a Tag Table system. They are implemented in isolation, either evaluating the performance of a POLB *or* the performance of a translation cache, never together. Further, it is assumed that these structures must be accessed prior to initiating a walk, therefore their latency can not be overlapped with any other operation in the Tag Table. For the POLB, regardless of size, the assumed latency is two cycles while the translation caches are assumed to take 4 cycles. While a 2-cycle POLB or 4-cycle translation cache may be optimistic, results below will show this point is moot since even the assumed latencies are incapable of providing significant advantage.

5.4.1 POLBs

Figure 5.6 presents the impact of a very large, highly associative (8K entry, 16-way) POLB on the performance of a Tag Table system. The results are normalized to a baseline Tag Table system without any translation optimizations, showing there is no benefit to be had. Indeed, there is minor slowdown experienced by many applications. The reason for this is a very low POLB hit rate due to low locality at such a distant cache level. Further, the difference in latency between a POLB hit (2 cycles) and an average Tag Table walk (~14 cycles) is relatively small when compared to the difference in latency between a TLB and

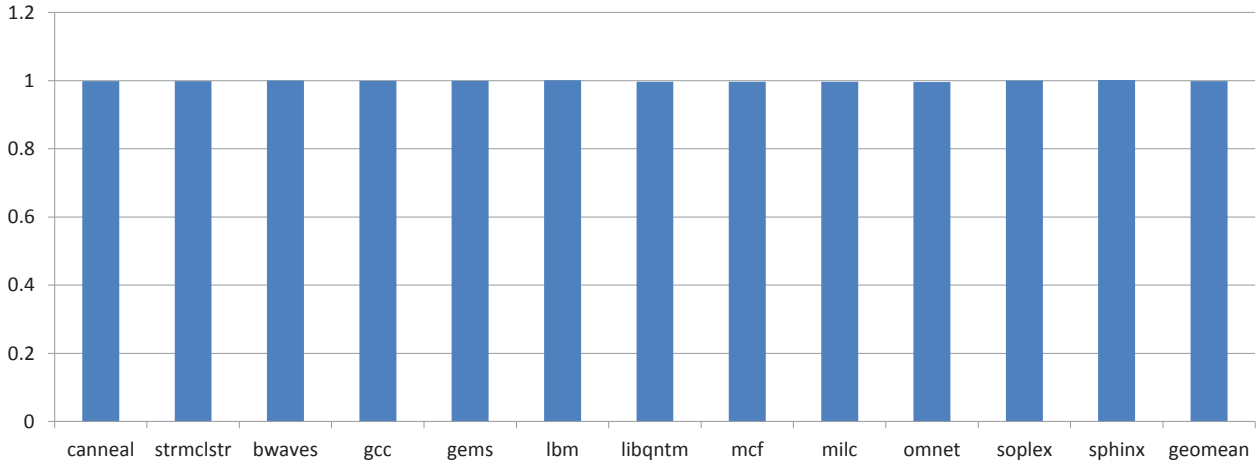


Figure 5.6: Speedup achieved with an 8K, 16-way page offset lookaside buffer.

page table.

5.4.2 Translation Caching

Figure 5.7 presents the rather uncompelling speedups achieved by a system implementing the translation path cache as previously described to allow Tag Table walks to initiate at intermediate levels of the table when considering a large 2K-entry structure. Similar to the POLB, the relatively low locality leads to low hit rates. Further, unlike the POLB which can be effective even when replacing a walk that only requires a single level by trading in an 8-cycle lookup with a 2-cycle lookup, even when the translation cache hits, it is only significantly valuable when it replaces a rather long walk with a significantly shorter one. Given the relatively shallow nature of Tag Tables however, this opportunity is rather rare. Instead, even when effective, the translation cache can frequently only improve a 2-level, 16-cycle access by reducing it to a 12-cycle access (access time of the translation cache plus accessing the leaf), resulting in only 25% improvement versus the 4x improvement

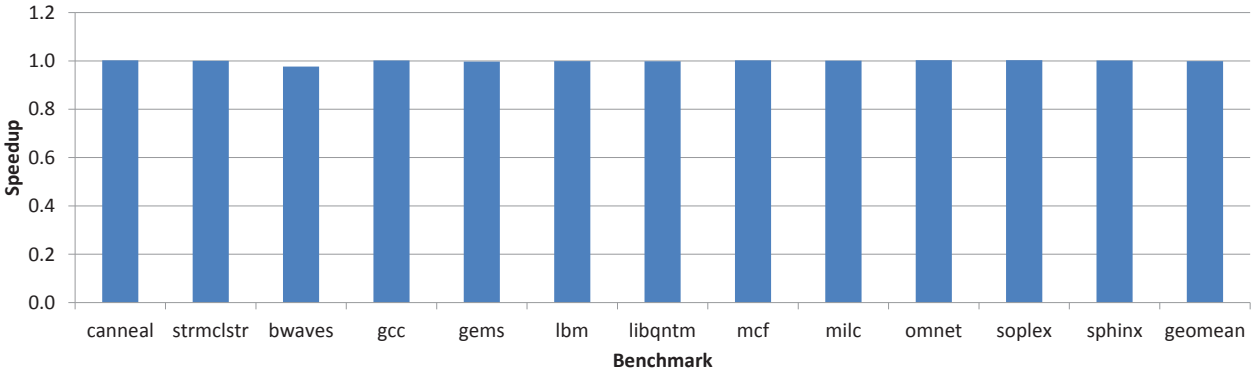


Figure 5.7: Speedup achieved with a 2K-entry translation cache.

capable by the POLB. Further, though these results do not include a POLB, a system with a POLB would provide even less opportunity for translation caching since most of the temporal locality that a translation cache could exploit would also be exploitable by a POLB. Therefore, when it comes to incorporating accelerating walks in a Tag Table system, it likely only makes sense to pick one of the two techniques.

5.5 Summary

This chapter has shown that the significantly different operation and design decisions of Tag Tables make them much less amenable to existing techniques for accelerating page walks for a traditional page table. Largely due to the low locality of reference at such a distant cache level, neither approach is frequently capable of accelerating accesses. This low hit rate is even more problematic in Tag Tables relative to a traditional page table due to the fact that there is relatively little opportunity to begin with. Due to the shallow tables facilitated by the upper level entries described in Chapter 3 and the location of metadata in

a collocated L3 bank, the difference between a POLB or translation cache hit is relatively insignificant and overshadowed by the high miss rate.

6 Prefetching

Prefetching techniques have long existed to help reduce cache misses by identifying likely blocks to fetch ahead of demands for the data. This sort of "proactive" data acquisition has been shown to be helpful in all levels of the cache hierarchy, even utilizing relatively simple heuristics like next-line or simple strides.

Tag Table's unique data structure introduces new limitations and new opportunities for prefetching. Due to their base-plus-offset encoding and non-static storage requirement, the additional consideration of tag storage cost needs to be accounted for when prefetching into a cache administered by a Tag Table. However, this encoding can also provide a new prefetching heuristic. Complementing existing "proactive" prefetching techniques, Tag Tables can also incorporate "reactive" prefetching. Instead of utilizing shortest chunk eviction as implemented by the baseline proposal in Section 3.3.2, when confronted with the situation of needing to track more distinct chunks than an entry can provide, Tag Tables can instead prefetch the blocks that represent the gap between existing chunks. In this way, the two separate chunks can be combined into one, freeing a chunk specification (a specific example to illustrate this will be presented in Section 6.2).

6.1 Background

Hardware-based, data prefetching techniques (as opposed to software-based and/or instruction prefetching approaches) can be broadly classified into three categories: stride and/or stream, address-correlating, and spatial-correlating.

6.1.1 Stride and Stream Prefetchers

The most commonly implemented form of hardware-based prefetching in commercial processors is the stride-based prefetcher [18]. Introduced by Baer and Chen, the original stride-based prefetcher utilized a "reference prediction table" (RPT) as shown in Figure 6.1 to track the recent strides and addresses associated with various program counter (PC) values along with a simple state machine to indicate when a particular entry was *untrained*, *training*, or *trained* [3]. Indexed by the PC of the associated load/store instruction, the RPT indicates the last address accessed by the PC, the stride associated with it, and the training state (as determined by the associated state machine). On first reference (i.e., no entry associated with the PC), an entry is assigned in the *training* state, recording the current address. On subsequent access, the stored value of "last address" is compared with the current address and an address-delta is computed and stored in the RPT in the "Stride" field. If an entry is in the *training* state and the stored delta matches the difference between a current access and the "Last Address," the entry transitions to the *trained* state. Upon entering this state, the prefetcher then begins to issue speculative fetches up to the *depth* of the prefetch, where the *depth* refers to the number of consecutive addresses to fetch at the

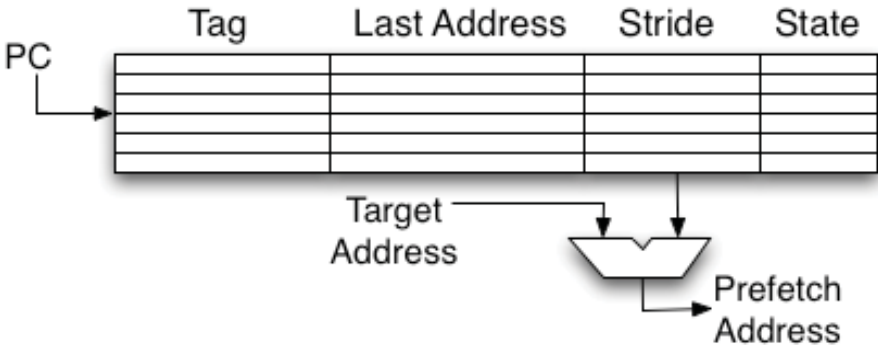


Figure 6.1: The reference prediction table used to identify strides in [3] (Taken from [21]).

given stride. For instance, on an access for address *0x1000* with a trained stride of 0x4 and a depth of 4, the addresses *0x1004*, *0x1008*, *0x100c*, and *0x1010* will additionally be fetched.

6.1.2 Address-Correlating Prefetchers

The Markov prefetcher is a widely cited address-correlating prefetcher that utilizes a Markov model to track the various states a given "trigger" may transition to and their relative likelihood. A "trigger" may be an address, a PC value, or a delta between requests. Figure 6.2 provides an example of a Markov graph using addresses as triggers, where - for instance - a request for address 'A' has a 50% chance of being followed by a request for address 'C' and a 50% chance of being followed by a request for address 'B'. With a Markov prefetcher, the first-order *depth* concept of the stride-based prefetcher is replaced with a *width*. Rather than issue a number of requests down a chain of addresses, the Markov prefetcher issues requests based on the probability that the current trigger will transition to different states in the graph (though it may also traverse the graph and issue a depth of requests). For instance, if a miss address corresponds to trigger 'A', the Markov prefetcher

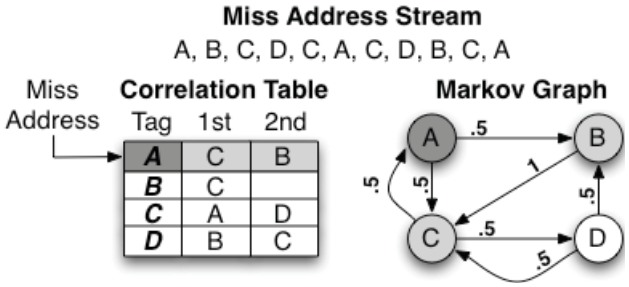


Figure 6.2: Markov Graph and Correlation Table (Taken from [38]).

may issue requests for addresses corresponding to triggers 'B' and 'C' since the graph indicates there's a 50% likelihood each that those addresses will be seen in the near future.

6.1.3 Spatially-Correlated Prefetchers

Spatially-correlated prefetchers take advantage of repetitive patterns of data access with respect to relative offsets of memory occurring near to each other in time [18]. These repeated patterns of access frequently arise due to abstractions in high-level programming paradigms (e.g., accessing various fields in objects or database records).

One early spatially-correlated prefetcher is the "Spatial Footprint Predictor" (SFP), which leverages correlations between the instructions that issue memory operations (determined by their PC value) and the data address's offset from the base of a large region [31]. Applied in the context of a sectored cache (where the large region is a sector), the SFP attempts to prefetch all data into a sector that will be referenced before it is evicted. This is accomplished when, upon first reference to a sector (i.e., a request for a block whose tag does not match the tag of the existing data in the sector, called a *nominating reference* in [31]), the SFP attempts to prefetch all of the blocks that were usefully referenced during the sector's last lifetime in

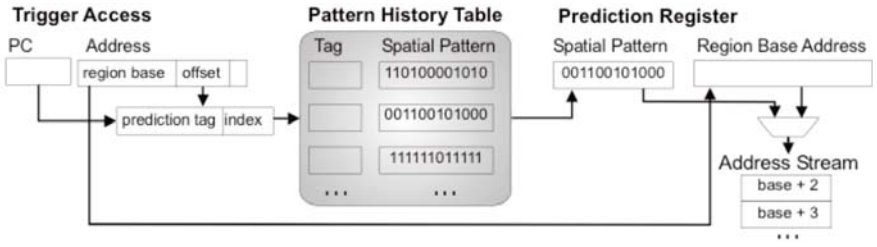


Figure 6.3: The Spatial Footprint Predictor mechanism (Taken from [18]).

the cache, as tracked in a pattern history table. Shown in Figure 6.3, this pattern history table is an associative structure that can be indexed either by the first reference's PC (i.e., the PC value of the block that allocates the sector or the *nominating instruction* per [31]), the first reference's address, or a combination. Evaluation in the original paper - which is corroborated by evaluation of the Footprint Cache [27] - finds that a simple combination of the upper bits of the PC along with the lower bits of the data address results in the best accuracy. On insertion, these bits are stored along with the sector's tag. Upon eviction of the sector, the table is updated by indexing it with this value retained in the tag and a bitvector indicating the blocks that received a demand reference during the current lifetime of the sector is stored.

Footprint Cache

The Footprint Cache is a recent application of SFP to a DRAM-based cache [27]. As discussed in Chapter 2, the low tag storage overhead of sectored caches makes them attractive for DRAM caches. Unfortunately, the evaluation in Chapter 3 reflecting the number of unique tags present in the sets of a cache, as shown in Figure 3.3 indicates this organization is not a good fit for many workloads, with many referencing relatively few blocks per sector (i.e.,

the workloads exhibit low "page density" [27]). However, if considering systems with a limited domain, such an approach may be beneficial. The Footprint Cache does just that by evaluating a sectored cache approach utilizing SFP on "scale-out workloads" [27]. Unlike more general workloads, "scale-out workloads" exhibit high page density, therefore the Footprint Cache is able to achieve impressive performance utilizing a sectored approach augmented with SFP. With the exception of evaluating page-sized sectors (1 KB to 4 KB), the prefetcher utilized in the Footprint Cache is a direct adaptation of the SFP and highlights its effectiveness for large DRAM caches, therefore Section 6.2 will evaluate Tag Tables extended with "proactive" prefetching utilizing SFP and will evaluate SFP on more general workloads.

6.2 Application to Tag Tables

In this section, the specific application of both reactive and proactive prefetching is presented as applied to Tag Tables. Given their distinctive structure relative to any prior work on prefetching, application of existing prefetching methods for proactive prefetching requires specific adaptation to Tag Tables. Further, the reactive prefetching presented is a unique opportunity made possible by this distinctive structure.

6.2.1 Reactive Prefetching

In order to describe the scenario that triggers a reactive prefetch in Tag Tables, consider the DRAM cache row presented in Figure 6.4 when the cache attempts to insert the "New

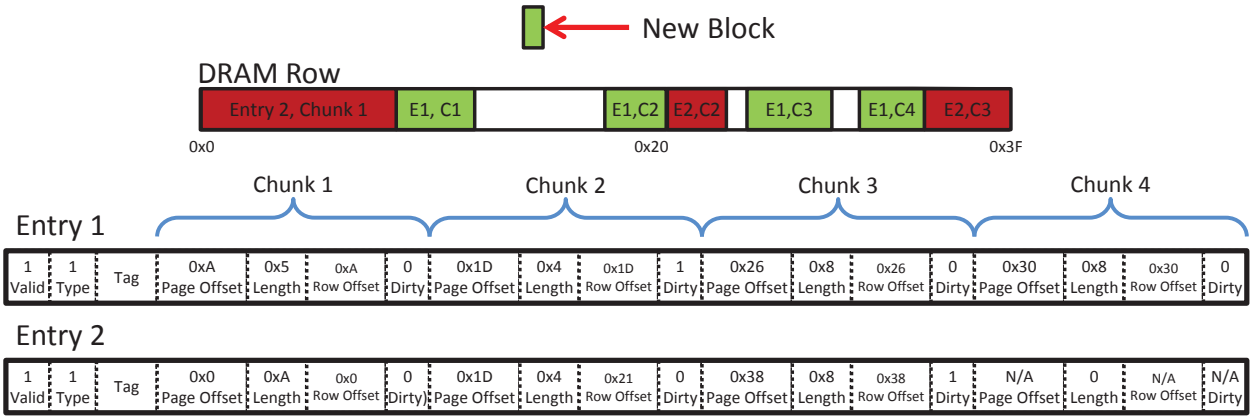


Figure 6.4: Example compressed entries associated with a DRAM cache row.

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 reactive 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 with a distance of two blocks (0x2E and 0x2F). Therefore, the blocks with offsets 0x2E and 0x2F associated with the entry can be fetched from main memory and inserted allowing 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 can occur following the bitvector population and victim determination of insertion as described in Section 3.3.2 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 and following an already long latency miss to acquire the fill data. Further, it only involves serializing two addition operations and a 6-bit comparison and thus is assumed to have negligible effect on performance.

6.2.2 Proactive Prefetching

Following successful adaptation of spatial footprint prediction (SFP) to a sectored DRAM cache in Footprint Caches [27], this section describes similar adaptation of SFP to Tag Tables. Since DRAM cache rows are the Tag Table's analogy to the Footprint Cache's sectors, a simple adaptation of SFP to Tag Tables would involve storing the state of entries upon their eviction in the Footprint History Table (FHT) and prefetching all blocks associated with a full entry on the first subsequent request for a block that maps to it. Unlike the Footprint Cache (and sectored caches in general) however, in a Tag Table multiple entries may be tracking data in the same DRAM cache row, therefore prefetching all blocks in an entry can lead to excessive evictions. While this flexibility in tracking multiple blocks that would conflict with each other in a sectored cache is generally a strength of Tag Tables, frequently improving its hit rate significantly by not incurring false conflicts, it limits the potential for

Tag Table SFP prefetching. Therefore, in a Tag Table adaptation of SFP, Tag Tables instead only prefetch on a chunk granularity. In other words, on a Tag Table access that requires allocation of an entry, the FHT is indexed by the PC and offset of the request and only the predicted blocks in any returned entry that are associated with the demand miss's chunk are prefetched.

For example, consider a miss for address 0x19ed5af140, triggered by the instruction at the program counter (PC) value of 0x4000 as shown in Figure 6.5 with the associated FHT state, assuming no blocks currently exist in the row with the same tag (0x19ed5af1). Therefore, a new entry must be allocated associated with the tag. After determining the miss and issuing the fill request for 0x19ed5af140, the Tag Table accesses the FHT with 0x19ed5af14000 (the tag of the request concatenated with the PC), retrieving the associated entry (5:3/20:5/32:7/3f:1) indicating the presence of offset 0x5 and the 3 following blocks, offset 0x20 and the following 5 blocks, offset 0x32 with the following 7, and offset 0x3f by itself; representing the status of Chunks 1, 2, 3, and 4 respectively upon the entry's last eviction. While there are many blocks associated with this entry, in order to limit evictions as mentioned previously, the Tag Table only issues prefetches for the blocks in the chunk associated with offset 0x5 (the offset associated with 0x19ed5af140). In this case, that is Chunk 1. Therefore blocks at offset 0x6 and 0x7 from the tag are also issued to memory as prefetches (0x19ed5af180 and 0x19ed5af1c0). Note, the FHT only needs to store the page offsets and lengths from the initial entry since the *dirty* and *row offset* fields are specific to the location and state of the data in the cache and the *valid*, *type*, and *tag* bits are either

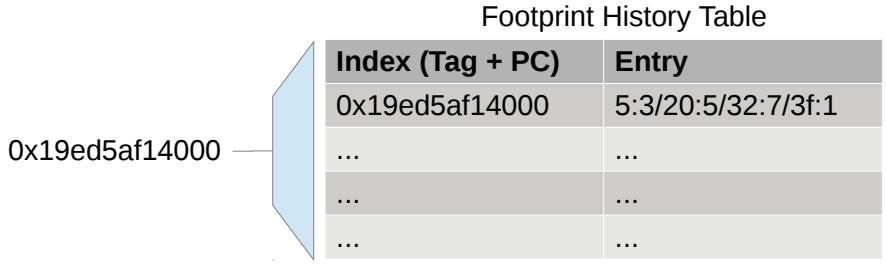


Figure 6.5: Example state of Footprint History Table when request for address 0x19ed5af140 is received. The second column retains relevant information from an entry at the time of its previous eviction, namely the page offsets and lengths of the chunks.

implicit with their presence in the table (i.e., not valid and compressed entry) or are part of the table's index. Therefore, the storage required for an FHT in a Tag Table implementation is less than the FHT utilized in Footprint Caches, which was already negligible at 144KB for a 16K entry table [27]. This is due to the fact that entries only consume 48 bits (6 bits for the offset and 6 bits for the length for each chunk) while the bitvector needed for Footprint Caches is 64 bits long for the 64 possible subblocks in a sector.

6.3 Evaluation

6.3.1 Reactive Prefetching

This section presents the opportunity available when the baseline Tag Table policy of shortest chunk eviction is replaced with prefetching of the least gap between chunks, also known as "reactive prefetching" per Section 6.2.1. Presented as speedup relative to a baseline Tag Table system that utilizes shortest chunk eviction, Figure 6.6 indicates this opportunity is frequently insignificant, but can range into the 5% - 7% region for workloads

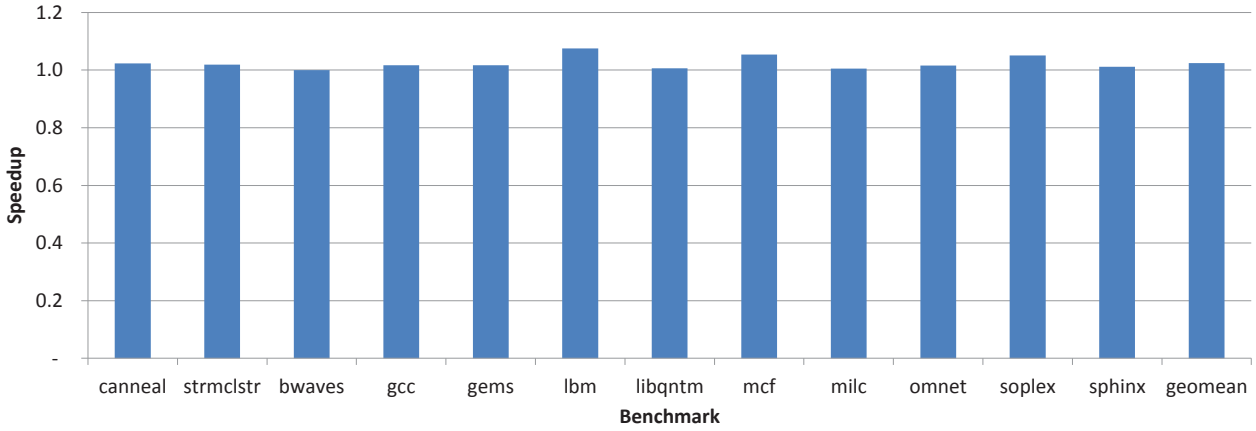


Figure 6.6: Speedup observed when changing Tag Table policy from eviction of shortest chunk to prefetching the least gap to maintain the correct number of chunks per entry.

such as *lbm*, *mcf*, and *soplex*. Overall, while it may not present a substantial opportunity, depending on the complexity of implementing the mechanism as described in Section 6.2.1, it may be a worthwhile opportunity.

6.3.2 Opportunity of Proactive Prefetching

Before evaluating the specific ability of spatial footprint prediction (SFP) to improve the performance of applications executing on systems utilizing DRAM caches, this section presents a limit study on the available opportunity for prefetching in sectored and Tag Table caches with a general workload set. This opportunity can be quantified by evaluating a system that models perfect prediction, meaning all subsequently referenced blocks are fetched to a prefetching region (a sector for sectored caches and a chunk for Tag Tables) following an allocating (i.e., triggering) request. This means that all references to a prefetching region following the allocating request hit. This is accomplished in the model by returning a hit for any reference to a prefetching region that is already allocated

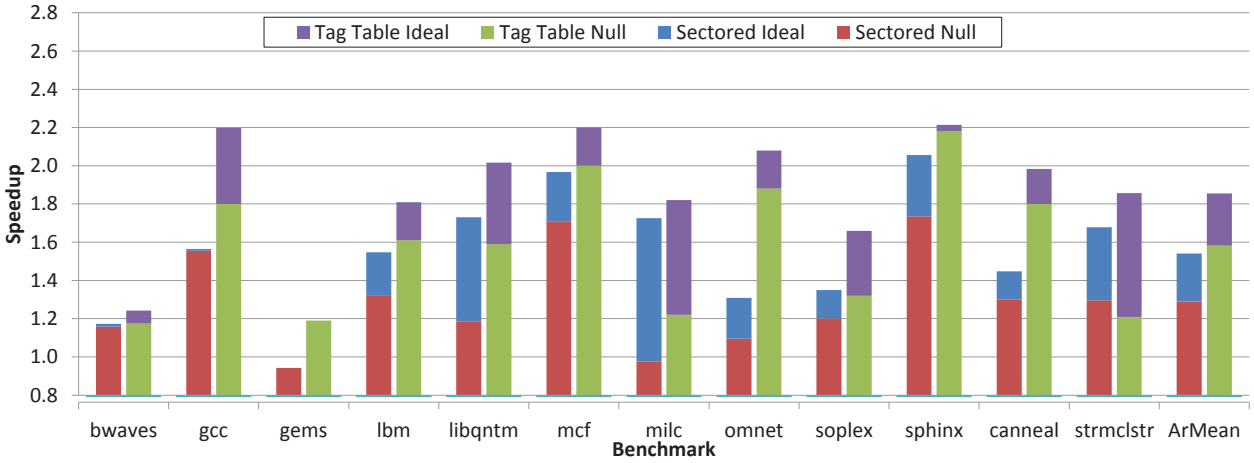


Figure 6.7: Speedup possible versus baseline system without DRAM cache with perfect prefetching in a Footprint Cache system with ("FP Ideal") and without ("FP Null") prediction and a Tag Table system with ("TT Ideal") and without ("TT Null") prediction.

(e.g., if the tag of the allocated sector matches the request's sector tag, the request is considered a hit). Further, it means that the appropriate bandwidth pressure off-chip is modeled by prefetching an appropriate number of blocks. In the model, this second goal is accomplished through an initial profiling simulation with a baseline sectored or Tag Table configuration that does not implement prefetching but instead generates a histogram of the number of present blocks in a sector or contiguous blocks in a chunk on eviction. Subsequent simulation implementing perfect prediction then generates a random number of prefetches to issue along with allocating fetches consistent with this histogram.

Figure 6.7 presents the results of this study for Sectored and Tag Table systems with and without prefetching, utilizing "Ideal" or "Null" prediction, respectively. The results reflect the speedup available over a baseline system as compared against in Section 3.5 which does not have a DRAM cache and instead interfaces its L3 cache with main memory. Notably, these results indicate there is substantial opportunity for prefetching even when

considering increased bandwidth pressure. Therefore, the following sections can indicate how much of this opportunity is realized by the particular SFP scheme.

6.3.3 Spatial Footprint Prediction with General Workloads

This section serves to evaluate the ability of spatial footprint prediction to prefetch the data that will be referenced in a sector when running general workloads in a system with a DRAM cache before evaluating the approach applied specifically to Tag Tables in the subsequent section. This is a similar approach to that taken by the Footprint Cache but applied to more general workloads.

Figure 6.8 presents the speedup achieved by a system utilizing SFP over a standard sectored cache with 4KB sectors and 64B lines leading to 64 blocks per sector. While exhibiting modest speedups in some benchmarks with *lbm*, *libqntm*, and *milc* notably in the 10% range, the average speedup of just 4% and the *slowdown* of *omnet* is somewhat disappointing.

Figure 6.9 helps explain the root of these disappointing results by presenting the accuracy of the predictor. Since a block in the DRAM cache can be categorized in two meaningful dimensions with respect to prefetching: 1) whether or not they were prefetched and 2) whether or not they were demanded, there are four combinations of these dimensions that represent the possible states of the blocks on eviction - when the graph's statistics are updated. Three of these states are useful in evaluating the predictor accuracy: 1) that a block is demanded and prefetched, 2) that it is demanded but not prefetched, and 3) that

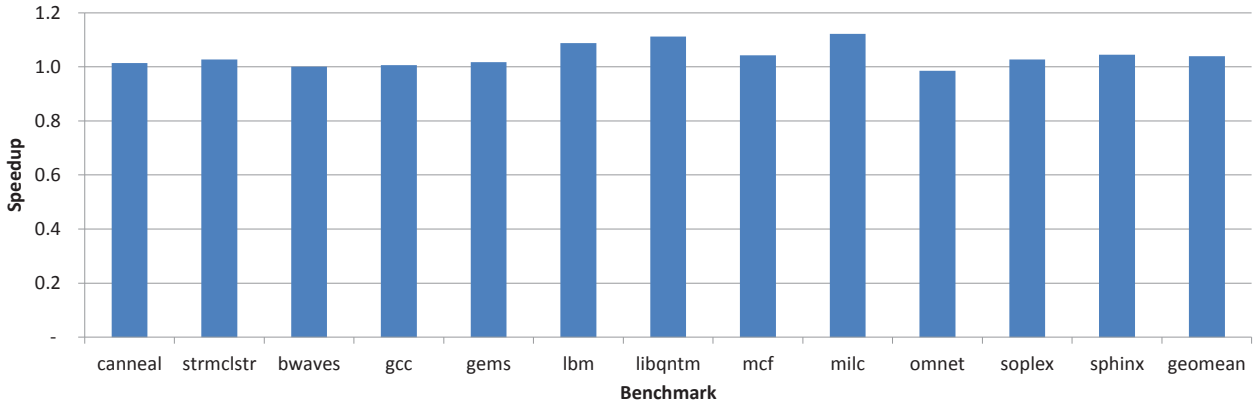


Figure 6.8: Speedup observed by use of Footprint Predictor on a sectored cache.

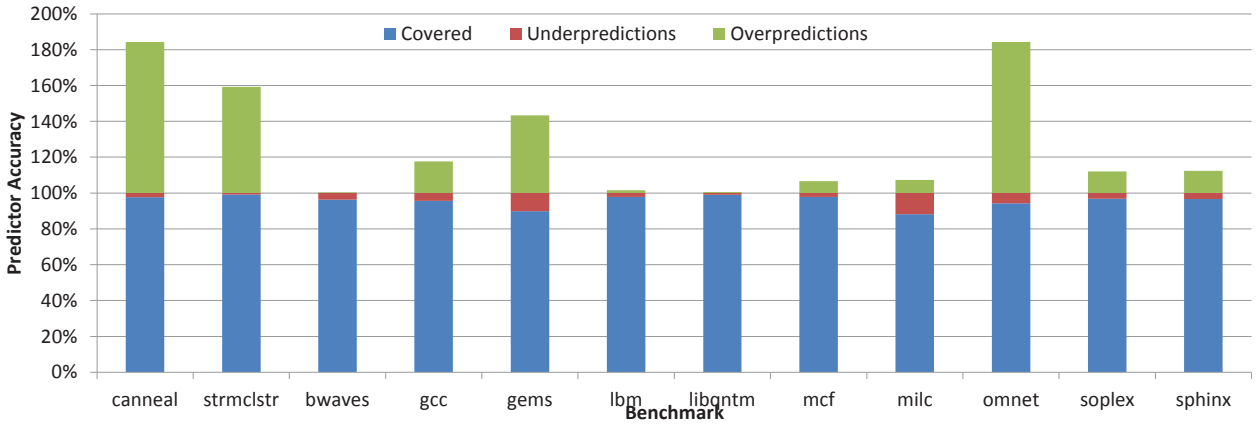


Figure 6.9: Accuracy of the Footprint Predictor over evaluated workloads, showing number of correctly predicted ("Covered"), Overpredicted, and Underpredicted blocks following sector allocation.

it is prefetched but not demanded[1]. These states lead to the three stacked categories of the graph, "Covered," "Underpredicted," and "Overpredicted," respectively. "Covered" references represent the ideal where prediction properly indicates a block will be referenced and issues a prefetch such that the data is present when the demand reference is seen (i.e., it is timely). "Underpredictions" represents both demands that are not predicted and therefore not prefetched and demands that are predicted but do not receive fill data in time

[1]For the sake of evaluating prediction accuracy, it is not interesting to consider those blocks that are neither demanded nor prefetched

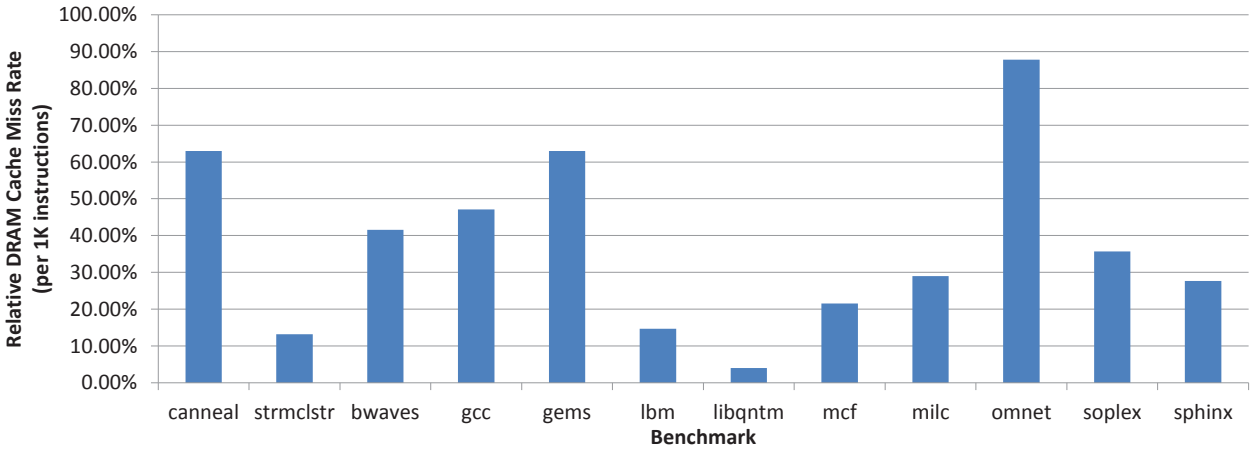


Figure 6.10: Relative DRAM cache misses per 1K instructions with Footprint Prediction.

(i.e., late prefetches). Finally, "Overpredictions" are those blocks that are prefetched but never referenced and merely serve to increase the bandwidth demand on main memory.

Altogether, while some benchmarks achieve high coverage with low overpredictions - not surprisingly headlined by *lbm* and *libqntm* which achieved two of the best speedups - many of the remaining benchmarks have significant overprediction rates. This overprediction leads to significant increase in DRAM bandwidth demand, in turn increasing main memory latency and subsequently slowing the system down relative to one without prediction.

While overall speedup results are disappointing and many workloads exhibit significant overfetching, Figure 6.10 indicates that prefetching is at least accomplishing the primary first-order goal of reducing DRAM cache miss rates. As shown and expected from the high "Covered" rate in Figure 6.9, DRAM cache miss rates can be substantially reduced with prefetching, even considering these reductions do not account for late prefetches (i.e., prefetches that do not return data in time are considered misses). Therefore, it is clear that

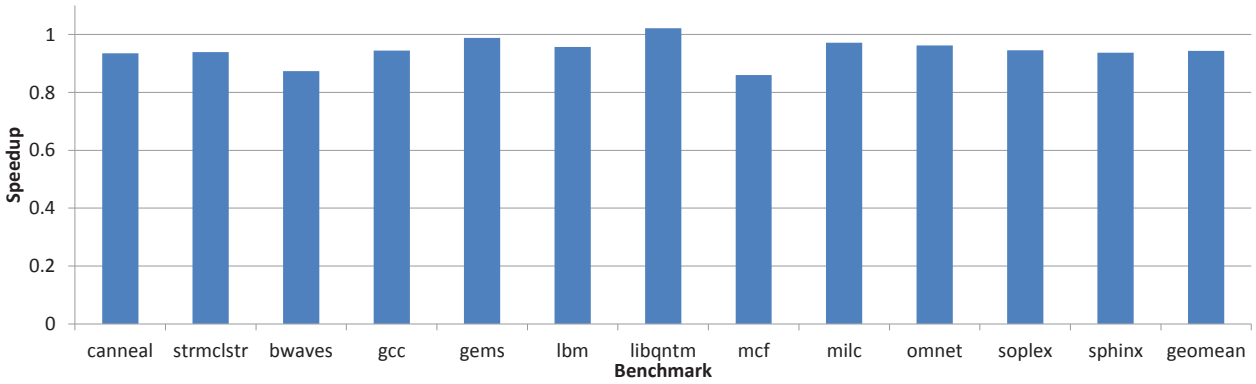


Figure 6.11: Speedup observed by use of Footprint Predictor on a Tag Table.

the only significant impediment to better performance with SFP-based prefetching is the significant overfetching rate. Given this conclusion, an adaptation that mitigates this effect could help substantially. While not investigated in this work, it can be envisioned that a throttling mechanism could be helpful. For instance if a feedback mechanism were present that let the system know when significant overfetching was occurring, it could adjust to reduce the amount of blocks prefetched. Whether the reduction in "Coverage" would be worth this concession is unclear, but would be an interesting study that is saved for future work.

6.3.4 Tag Table Prediction

Continuing by applying SFP to Tag Tables results in similarly disappointing results. As mentioned earlier, the application of proactive prefetching to a Tag Table encounters much different operating characteristics than application to sectored caches since, unlike a Tag Table, prefetching within a sector cannot displace useful data since the allocation of the sector itself already cleared the sector. In contrast, Tag Tables frequently must select a victim

in order to make room for prefetched data, leading to a tradeoff between the usefulness of the victim versus the prefetched block. Consistent with the prediction accuracy in the previous section, this results in additional penalty for Tag Tables, leading to the ability of SFP to improve performance of only benchmark - *libqntm* - as shown in Figure 6.11. With the exception of DRAM cache miss rates, the bandwidth and prediction accuracy results of the Tag Table with SFP is very similar to that achieved by the sectored cache. The DRAM cache miss rates on the other hand, are largely unchanged from the baseline system, indicating a canceling effect between useful prefetches and evictions created by useless prefetches.

6.4 Summary

This chapter proposed and evaluated prefetching techniques in a Tag Table system. Along with adaptation of spatial footprint prediction (SFP) - a prefetching technique consistent with the goals of prior work that attempts to provide data to the cache prior to first reference - it further presented a reactive prefetching mechanism. This reactive mechanism is presented as an alternative to eviction of shortest chunks for maintaining correctness in a Tag Table system with respect to limiting the number of chunks tracked by an entry that utilizes prefetching of the shortest gap between chunks. It is this unique ability of Tag Tables - to *reduce* tag storage overhead by tracking more blocks - that ends up providing the greatest benefit. While limited to only a few percent improvement, it is a simple mechanism for correctness that is capable of improving performance beyond that achieved by a more

destructive approach for correctness.

Beyond reactive prefetching, analysis of proactive prefetching in the form of SFP reveals significantly different characteristics with general workloads than those results achieved with "scale-out workloads." While the initial evaluation of perfect prediction indicated significant opportunity for prefetching, SFP was unable to effectively capture this opportunity, indeed resulting in slowdown for all but a few workloads in the sectored configuration which exhibit very minor speedup. Not surprisingly, the lack of correlation between instructions and data in general workloads leads to relatively poor prediction and subsequently low performance. Further, SFP would appear to have little practical value to a Tag Table system targeted at general workloads (though evaluation of the Footprint Cache indicates that it may be worthwhile if the system were targeted to "scale-out workloads" [27]). Instead, further investigation into proactive prefetching for Tag Tables may be more fruitful if more general stride or address-correlating prefetchers were investigated which may be better able to capture the opportunity.

7 Conclusion

Large caches relative to the capacity available on a high-speed logic die are re-emerging as a challenge for computer architects. This has initiated a period of rediscovery of prior techniques proposed in an age of large board-level caches and spawned the discovery of new approaches. Whether implemented as trench capacitors on the logic die itself as in embedded DRAM (eDRAM) or existing as a separate DRAM chip, tightly integrated to the logic die through 3D die stacking technology, the availability of high-capacity, DRAM-based caches appear to be a design consideration for the foreseeable future. Fundamental to the designs to support these large caches is the mechanism by which data is tracked in the cache since a simple scaling up of existing techniques encounters substantial issues. These issues center on the storage and latency of accessing the tags for locating data in the cache. These tags are either too big to store on-die, requiring a vast reduction in storage overhead or they may be stored in the DRAM array themselves but then the issue of access latency becomes a major concern.

Prior techniques to leverage this big, fast data store by efficiently tracking the tags has spawned a range of proposals. From early proposals to simply scale the block size with

the cache size in order to maintain the tags on the logic die [17, 28, 28] to more advanced techniques leveraging sectored cache concepts and prefetching *a la* the Footprint Cache [27], significant work has been done to keep the tags on-die. On the other end of the spectrum, approaches such as the Loh-Hill Cache [33] and Alloy Cache [41] argue that block sizes must remain small in order to efficiently utilize this greater cache capacity and avoid false conflicts and fragmentation, thus the tags need to be stored in the DRAM. However, even these tags-in-DRAM approaches have taken divergent paths on fundamental design choices. While the Loh-Hill Cache advocates maintenance of associativity, the Alloy Cache advocates for a direct-mapped, hit-latency-optimized approach similar to the design decisions of past board-level caches, arguing that the capacity afforded these caches make associativity less important.

7.1 Summary

In order to tackle the fundamental issue of tag tracking for large DRAM-based caches, this thesis has presented Tag Tables, a dynamic mechanism for tracking tags in a cache. Though specifically designed for large-capacity, DRAM-based caches, it is a generally applicable tag tracking structure that incurs little storage overhead and can adapt to runtime characteristics of the system. Therefore, it is conceivably applicable to even smaller caches if adaptation or storage overhead are important concerns.

Tag Tables further advocate the importance of quickly accessing tags by storing them on the high-speed logic die, but also argue that small block sizes and associativity are

important, putting them at odds with prior techniques. Tag Tables are able to address these seemingly contradictory goals with compressed tag storage leveraging the intuition of long residency of blocks in the cache leading to many contiguous blocks with many repeating tag bits. By encoding this repetition into a base-plus-offset encoding, Tag Tables are able to achieve significant storage savings on the order of a sectored cache. Further, the dynamic nature of a forward page table is utilized to allow on-demand allocation of tag storage metadata, allowing the Tag Tables to adapt to runtime characteristics of the system and trade off DRAM cache tracking capacity for other on-die commodities such as L3 capacity.

Through this thesis, it has been shown that Tag Tables are capable of exceeding the performance of existing state-of-the-art proposals for tracking data in a large capacity cache. Further, through additional optimizations, this advantage can be extended. Namely, Set Dueling was first shown to be effective in Chapter 4 at selectively protecting Tag Table metadata in the L3 cache to allow for increased DRAM cache capacity when the L3 cache is not undergoing significant capacity pressure itself. This protection of metadata resulted in modest improvement for certain workloads that was amplified in systems utilizing bigger Tag Table entries in order to track more distinct chunks. Further, Set Dueling was adapted and a novel Set Scouting technique was proposed that increased the benefit achievable by Set Dueling by a small additional amount for certain workloads.

Next, the application of translation lookaside buffers (TLBs) in the form of page offset lookaside buffers (POLBs) were shown in Chapter 5 to provide significant benefit when sized on the order of at least a few hundred entries. By significantly reducing the tag check

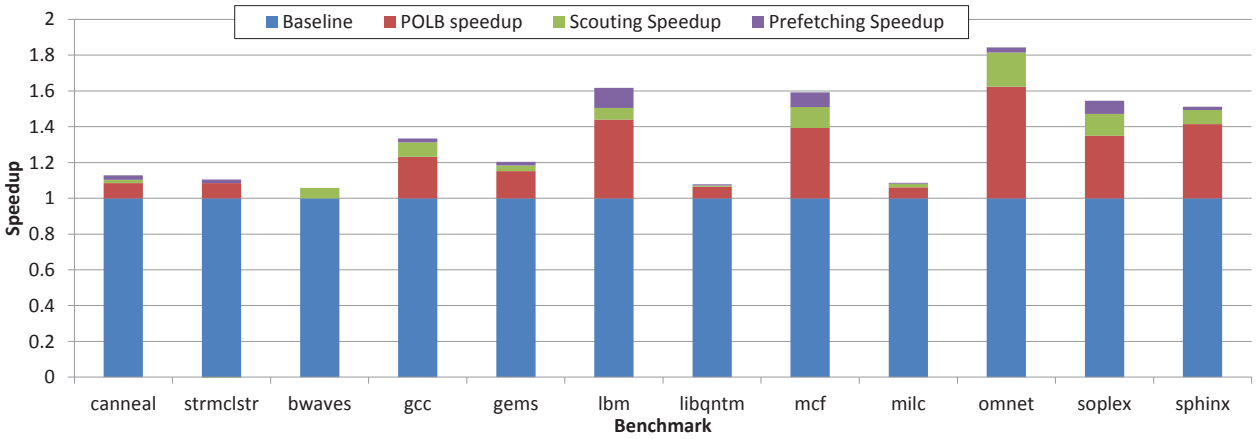


Figure 7.1: Combined effect of all useful Tag Table optimizations relative to baseline system as proposed in Chapter 3.

time of Tag Tables from a minimum of an L3 cache data access to two cycles, these POLBs were capable of improving performance by more than 20% on average with 1K entries. It should be noted that, unlike other advantageous optimizations presented which rely on intrinsic features of the Tag Table structure, the POLB is easily adaptable to other tag tracking techniques. Therefore, the most significant optimization uncovered for Tag Tables could be similarly applied to prior proposals in the realm of tag tracking.

Finally, prefetching in the form of reactive prefetching to maintain the maximum number of chunks trackable by an entry was shown to provide small, but non-negligible performance improvement in Chapter 6. Unlike proactive prefetching which is the type most commonly associated with prefetching, reactive prefetching exploits a unique feature of Tag Tables - that tracking *more* blocks can result in less storage overhead - by bringing in additional blocks to allow existing disjoint chunks to join and be trackable by a single entry.

Figure 7.1 presents the performance of a Tag Table system with all of the aforementioned

valuable optimizations included relative to the baseline system presented in Chapter 3, outlined in Table 3.1. The optimized system evaluated consists of the baseline system augmented with a 1K-entry POLB, Set Scouting, and reactive prefetching to maintain a maximum number of chunks in entries.

7.2 Future Work

While this thesis presents extensive evaluation into the structure and opportunities of Tag Tables, additional work could be done to extend or adapt it. Further, throughout the course of this work, several alternatives and additional opportunities were considered related to various Tag Table structures and operations. This section serves to enumerate several of these observed opportunities for further study regarding Tag Tables.

7.2.1 Chunk Specification

As proposed, a chunk in Tag Tables is encoded with an offset and a length. If instead, two offsets were encoded - one representing the bottom of the chunk and the other the top - certain operations may be less complex. While not taken into account in evaluations, certain proposed microarchitectural features of Tag Table operations have indicated the addition of adders to determine the top block of a given chunk. Specifically reactive prefetching of the shortest gap between chunks as presented in Chapter 6, relies on 6-bit adders to determine the top of the chunks in the entry for comparison with the base of the following chunk. If instead the top of the chunk were stored, these adders would not be necessary.

7.2.2 Combining Tag Tables with the System Page Table

Considering the reach of the translation lookaside buffers (TLBs) in current systems relative to the capacity of DRAM caches as discussed in Section 5.1.1, it would seem that many DRAM cache accesses may follow from TLB misses. While it is entirely possible that temporal locality prevents this from being the case (i.e., enough references within a small set of pages are responsible for a majority of the DRAM cache accesses over a window of time), if it in fact does happen that many accesses follow a TLB miss, incorporation of Tag Tables into the existing system's page table may be advantageous. Rather than allocating separate resources to implement a Tag Table, it may be possible to just extend the translation stored in the leaves of the page table to indicate which blocks are located in the DRAM cache and the way they are located at. If indeed a page walk would have occurred in the common case anyway, not only are no additional on-chip resources devoted to tag tracking for the DRAM cache, but no additional latency is incurred as the location will be known before the DRAM cache is even accessed. Further, this may not be limited to just a DRAM cache. Given the relative reach of existing TLBs and rapidly increasing LLC sizes, it may be that eventually many LLC accesses will occur after a page walk has been performed. Granted, this is much less likely than with a DRAM cache and indeed TLBs will likely grow in response to LLCs, but it may be worth investigating.

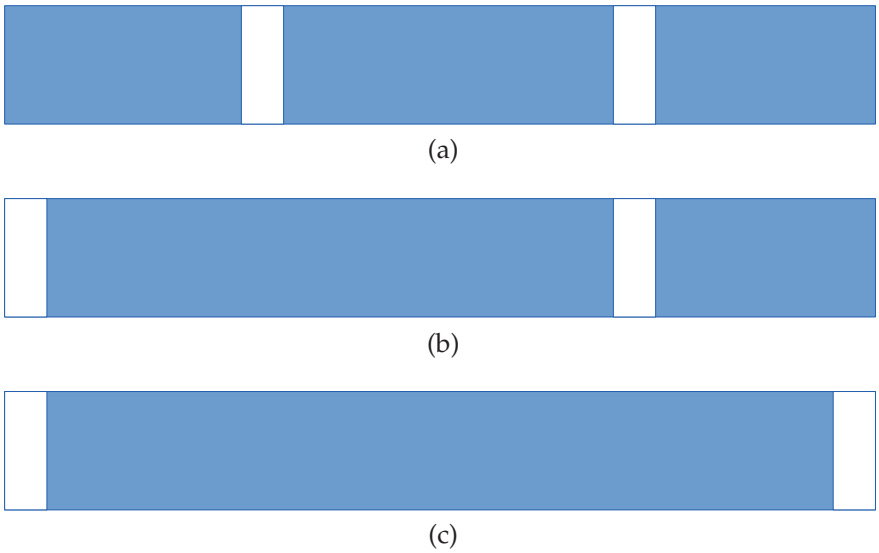


Figure 7.2: (a) Good, (b) bad, and (c) worse scenarios for exception tracking.

7.2.3 Encoding Exclusions in Entries

With the data presented here, corroborating the intuition that large caches will frequently store large contiguous chunks of data, dependent on application characteristics, it may be fruitful to explore the storage of *exceptions* in Tag Table entries. Consider, for instance, a row of the cache with all contiguous data except for one block. Instead of using two entries - one for all of the contiguous data which exists in two chunks and one for the lone block - a single entry could be allocated that indicates that *all* of the data in the particular row corresponded to a given tag, *except* for a chunk that contained that lone block. Implementing such an encoding would require an additional bit per entry indicating whether it was a standard entry or an exception entry.

While seemingly a good fit for this extreme example, it has not yet been investigated due to the belief that in the much more common case, exceptions will require *more* entries

and that even that one extra bit per entry would not likely be useful. This belief stems from the observation that unless the present (i.e., the non-exception) blocks are at the front and back of the row (i.e., they are present at offset 0x0 and offset 0x3f) then it will require tracking more chunks. Take the example in Figure 7.2a. For this row, it will be less costly to encode exceptions because they only correspond to two chunks while the present blocks correspond to three. If, on the other hand, the row was like Figure 7.2b or worse, like Figure 7.2c, it would require equal or even more chunks to encode the state. While not evaluated quantitatively, it would seem that situations from Figure 7.2b and 7.2c are relatively more common than Figure 7.2a, therefore this idea has not yet been investigated further.

7.2.4 Prefetching

While spatial footprint prediction (SFP) was not shown to work well with Tag Tables and general workloads, it may be that other prefetching techniques would work better. Considering that SFP is specifically targeted to sectored caches which have the nice property of not creating evictions if a footprint is prefetched, it was not well-suited to Tag Tables. However, Tag Tables have much more in common with traditional caches in that respect. Therefore, given the fact that assumptions and constraints are not as drastically different between the two, it might be better to adapt prefetching techniques that have been shown to work well with traditional small-block-sized caches. Specifically, incorporation of a stride prefetcher [3] or even a global history buffer [38] could provide the sort of prefetching

targets that would be valuable enough to justify the creation of additional victims.

7.2.5 Splitting Entries

If an insertion would cause an existing entry to track more than the number of chunks supported by a single entry (four in the baseline), it may be possible to instantiate a second entry for the tag and utilize the chunks in this new entry to begin tracking additional data. This opportunity potentially provides a third solution to the chunk limitation beyond simple eviction and reactive prefetching. The dynamic nature of Tag Tables, based on the page table structure, makes this a plausible option, contrary to the rigid storage of data in other tag tracking structures.

Since a single additional bit per entry can be used to indicate that an additional entry exists, the remaining implementation issue becomes how to locate the second entry. Conceivably, a sort of linked list could be created from one entry to another, but provisioning for the cost of storing a pointer to a separate memory location in all entries would be prohibitively expensive. Perhaps instead of full pointers, just indices into a table could be used. Then - assuming that the additional entry would be allocated in the same level as the current entry exists - the stored offset could then simply trigger an additional access into the table at the current level. This additional access would specifically be a second data access into the L3 for a metadata block. Although, perhaps an optimization could be to attempt to allocate this new entry at an index that would exist in the parent's metadata block, eliminating this second L3 access.

In the case of a subsequent reference walking to their proper leaf entry but encountering one of these "squatter" entries, a simple tag check can indicate that the entry does not belong where it is stored. It can then be moved to a new empty entry, requiring an update of the original entry, whose overflow caused the instantiation of the current problematic entry in the first place. This locating of the parent entry should be simply accomplished by indexing the table with the tag of the "squatter." This update again requires an additional access of the L3 for a metadata block, one that cannot be conceivably optimized.

7.3 Closing Remarks

Two primary features of the research presented in this thesis make me optimistic as to its future relevancy: 1) it is based on technology that has a relatively high probability of seeing widespread adoption and 2) at its core, Tag Tables are an adaptation of an existing, well-known technique for tracking memory. Both of these features I believe, are common research proposal pitfalls that result in them failing to achieve their desired relevancy.

First, concerning technology Tag Tables are primarily contingent on the adoption of DRAM-based caches. While it is not strictly limited to these caches - and can indeed be valuable to any high-capacity cache - the assumptions influencing Tag Table design decisions were all made with the application to tracking tags for a DRAM-based cache in mind. Fortunately, the presence of eDRAM on existing commercial products and widespread industrial support for stacked DRAM means that the availability of high-capacity, high-bandwidth DRAM to future processors is relatively likely. Further, the fact that *two distinct*

technologies could be independently possible of providing this data store makes it even more likely. Unfortunately for many other research proposals of the recent past - phase change memory comes immediately to mind - this robust technological basis is not so rosy.

Second, the fact that Tag Tables - at their core - adopt a ubiquitous and well-known structure in the page table means that their implementation would be well-understood and relatively non-disruptive, particularly since they can be incrementally realized. Such an incremental adoption could first involve only the very basic features of Tag Table design and evolve over multiple generations to incorporate the more advanced features such as compressed entries, etc. For instance, an initial implementation could merely perform a tag check as a walk of a forward page table, tracking only the "way" of the data in leaf entries and only allowing leaves to exist at levels reached after utilizing all bits of the address. Storage of the metadata could be in main memory, just as a standard page table and even small translation caches - which would be relatively more useful in this scenario - could be included to mitigate the long walks. An evolution of the design could then incorporate compressed entries and storage of the metadata in the on-chip cache. This process could continue - or not, depending on performance of the simple techniques - for many generations until what would initially be highly disruptive would be well-understood and simple to realize.

While I am generally pragmatic and somewhat cynical, I hope that together these two features mean that tracking tags in large caches will remain a relevant problem that Tag Tables help to solve. Whether or not I personally work on solving it is unclear but it would

be exceptionally gratifying if at some point hindsight indicates that this work represented a meaningful contribution to the tracking of cache tags.

Bibliography

[1] Graig Anderson and Jean-Loup Baer. *Design and evaluation of a subblock cache coherence protocol for bus-based multiprocessors*. Tech. rep. UW CSE 94-05-02. University of Washington, 1994.

[2] J. Andrews and N. Baker. "Xbox 360 System Architecture". In: *Micro, IEEE* 26.2 (2006), pp. 25–37. ISSN: 0272-1732. DOI: 10.1109/MM.2006.45.

[3] Jean-Loup Baer and Tien-Fu Chen. "An Effective On-chip Preloading Scheme to Reduce Data Access Penalty". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: ACM, 1991, pp. 176–186. ISBN: 0-89791-459-7. DOI: 10.1145/125826.125932. URL: http://doi.acm.org/10.1145/125826.125932.

[4] K.C. 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. DOI: 10.1109/ISPASS.2005.1430560.

[5] Thomas W. Barr, Alan L. Cox, and Scott Rixner. "Translation caching: skip, don't walk (the page table)". In: *Proceedings of the 37th annual international symposium on Computer architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 48–59.

[6] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. "Accelerating two-dimensional page walks for virtualized systems". In: *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ASPLOS XIII. Seattle, WA, USA: ACM, 2008, pp. 26–35.

[7] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. "Accelerating Two-dimensional Page Walks for Virtualized Systems". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: ACM, 2008, pp. 26–35. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346286. URL: http://doi.acm.org/10.1145/1346281.1346286.

[8] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, 2011.

[9] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, Lei Jiang, G.H. Loh, D. McCauley, P. Morrow, D.W. Nelson, D. Pantuso, P. Reed, J. Rupley, Sadasivan 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.

[10] R. Brain, A. Baran, N. Bisnik, H.-P. Chen, S.-J. Choi, A. Chugh, M. Fradkin, T. Glassman, F. Hamzaoglu, E. Hoggan, R. Jahan, M. Jamil, C.-H. Jan, J. Jopling, H. Kan, R. Kasim, S. Kirby, S. Lahiri, B.-C. Lee, D. Lenski, J. Limb, N. Lindert, M. Musorrafiti, J. Neulinger, L. Rockford, J. Park, K. Singh, C. Staus, J. Steigerwald, B. Turkot, P. Vandervoorn, R. Venkatesan, S. Wu, J.-Y. Yeh, Y. Wang, Z. Zhang, and K. Zhang. "A 22nm high performance embedded DRAM SoC technology featuring tri-gate transistors and MIMCAP COB". In: *VLSI Technology (VLSIT), 2013 Symposium on*. 2013, T16–T17.

[11] Brian K Bray, William L Lynch, and Michael J Flynn. *Page allocation to reduce access time of physical caches*. Tech. rep. 1990.

[12] N. Butt, K. McStay, A. Cestero, H. Ho, W. Kong, S. Fang, R. Krishnan, B. Khan, A. Tessier, W. Davies, S. Lee, Y. Zhang, J. Johnson, S. Rombawa, R. Takalkar, A. Blauberg, K.V. Hawkins, J. Liu, S. Rosenblatt, P. Goyal, S. Gupta, J. Ervin, Z. Li, S. Galis, J. Barth, M. Yin, T. Weaver, J.H. Li, S. Narasimha, P. Parries, W.K. Henson, N. Robson, T. Kirihata, M. Chudzik, E. Maciejewski, P. Agnello, S. Stiffler, and S.S. Iyer. "A 0.039um2 high performance eDRAM cell based on 32nm High-K/Metal SOI technology". In: *Electron Devices Meeting (IEDM), 2010 IEEE International*. 2010, pp. 27.5.1–27.5.4. DOI: 10.1109/IEDM.2010.5703434.

[13] Trevor E Carlson, Wim Heirman, and Lieven 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.

[14] Licheng Chen, Yanan Wang, Zehan Cui, Yongbing Huang, Yungang Bao, and Mingyu Chen. "Scattered superpage: A case for bridging the gap between superpage and page coloring". In: *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. 2013, pp. 177–184. DOI: 10.1109/ICCD.2013.6657040.

[15] Yuan Chou, Brian Fahs, and Santosh Abraham. "Microarchitecture optimizations for exploiting memory-level parallelism". In: *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. 2004, pp. 76–87. DOI: 10.1109/ISCA.2004.1310765.

[16] Yangdong Deng and Wojciech P. Maly. "Interconnect Characteristics of 2.5-D System Integration Scheme". In: *Proceedings of the 2001 International Symposium on Physical Design*. ISPD '01. Sonoma, California, USA: ACM, 2001, pp. 171–175. ISBN: 1-58113-347-2. DOI: 10.1145/369691.369763. URL: http://doi.acm.org/10.1145/369691.369763.

[17] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. ISBN: 978-1-4244-7559-9. DOI: 10.1109/SC.2010.50. URL: http://dx.doi.org/10.1109/SC.2010.50.

[18] Babak Falsafi and Thomas F Wenisch. "A Primer on Hardware Prefetching". In: *Synthesis Lectures on Computer Architecture* 9.1 (2014), pp. 1–67.

[19] Subhash Gupta, Mark Hilbert, Sangki Hong, and Robert Patti. "Techniques for producing 3D ICs with high-density interconnect". In: *Proceedings of the 21st International VLSI Multilevel Interconnection Conference*. Waikoloa Beach, HI, 2004.

[20] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Tom Piazza, and Ted Burton. "Haswell: The Fourth-Generation Intel Core Processor". In: *Micro, IEEE* 34.2 (2014), pp. 6–20. ISSN: 0272-1732. DOI: 10.1109/MM.2014.10.

[21] Mitchell Hayenga. "Power-Eï¬cient Loop Execution Techniques". PhD thesis. University of Wisconsin - Madison, 2013.

[22] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[23] Mark Hill, Susan Eggers, Jim Larus, George Taylor, Glenn Adams, B. K. Bose, Garth Gibson, Paul Hansen, Jon Keller, Shing Kong, Corinna Lee, Daebum Lee, Joan Pendleton, Scott Ritchie, David A. Wood, Ben Zorn, Paul Hilfinger, Dave Hodges, Randy Katz, John Ousterhout, and Dave Patterson. "Design Decisions in SPUR". In: *Computer* 19.11 (1986), pp. 8–22.

[24] Mark D. Hill. "A Case for Direct-Mapped Caches". In: *Computer* 21.12 (1988), pp. 25–40. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/2.16187.

[25] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide, Part I*. Vol. 3A. 2013.

[26] Aamer Jaleel. "Memory characterization of workloads using instrumentation-driven simulation". In: *Web Copy: http://www. glue. umd. edu/ajaleel/workload* (2010).

[27] Djordje Jevdjic, Stavros Volos, and Babak 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.

[28] Xiaowei Jiang, N. Madan, Li Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian. "CHOP: Adaptive filter-based DRAM caching for CMP server platforms". In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. 2010, pp. 1–12. DOI: 10.1109/HPCA.2010.5416642.

[29] R. Kalla, B. Sinharoy, W.J. Starke, and M. Floyd. "Power7: IBM's Next-Generation Server Processor". In: *Micro, IEEE* 30.2 (2010), pp. 7–15. ISSN: 0272-1732. DOI: 10.1109/MM.2010.38.

[30] David Kanter. *Intel's Long Awaited Return to the Memory Business*. 2013. URL: http://www.realworldtech.com/intel-dram.

[31] Sanjeev Kumar and Christopher Wilkerson. "Exploiting Spatial Locality in Data Caches Using Spatial Footprints". In: *Proceedings of the 25th Annual International Symposium on Computer Architecture*. ISCA '98. Barcelona, Spain: IEEE Computer Society, 1998, pp. 357–368. ISBN: 0-8186-8491-7. DOI: 10.1145/279358.279404. URL: http://dx.doi.org/10.1145/279358.279404.

[32] Gabriel H. Loh. "3D-Stacked Memory Architectures for Multi-core Processors". In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 453–464. ISBN: 978-0-7695-3174-8.

[33] Gabriel H. Loh and Mark D. Hill. "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44 '11. Porto Alegre, Brazil: ACM, 2011, pp. 454–464. ISBN: 978-1-4503-1053-6.

[34] Gabriel H. Loh, Yuan Xie, and Bryan Black. "Processor Design in 3D Die-Stacking Technologies". In: *Micro, IEEE* 27.3 (2007), pp. 31–48. ISSN: 0272-1732. DOI: 10.1109/MM.2007.59.

[35] Lucas Mearian. *Micron ships Hybrid Memory Cube that boosts DRAM 15X*. 2013. URL: http://www.computerworld.com/s/article/9242664/Micron_ships_Hybrid_Memory_Cube_that_boosts_DRAM_15X.

[36] Timothy Prickett Morgan. *Nvidia to stack up DRAM on future 'Volta' GPUs*. 2013. URL: http://www.theregister.co.uk/2013/03/19/nvidia_gpu_roadmap_computing_update.

[37] P.R. Morrow, C. M Park, S. Ramanathan, M.J. Kobrinsky, and M. Harmes. "Three-dimensional wafer stacking via Cu-Cu bonding integrated with 65-nm strained-Si/low-k CMOS technology". In: *Electron Device Letters, IEEE* 27.5 (2006), pp. 335–337. ISSN: 0741-3106. DOI: 10.1109/LED.2006.873424.

[38] Kyle J. Nesbit and James E. Smith. "Data Cache Prefetching Using a Global History Buffer". In: *Proceedings of the 10th International Symposium on High Performance Computer Architecture*. HPCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 96–. ISBN: 0-7695-2053-7. DOI: 10.1109/HPCA.2004.10030. URL: http://dx.doi.org/10.1109/HPCA.2004.10030.

[39] 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*. 2004, pp. 18–22. DOI: 10.1109/SBAC-PAD.2004.40.

[40] K. Puttaswamy and G.H. Loh. "Implementing register files for high-performance microprocessors in a die-stacked (3D) technology". In: *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*. 2006, 6 pp.–. DOI: 10.1109/ISVLSI.2006.56.

[41] 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: *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. 2012, pp. 235–246.

[42] M.K. Qureshi, D. Thompson, and Y.N. Patt. "The V-Way cache: demand-based associativity via global replacement". In: *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*. 2005, pp. 544–555. DOI: 10.1109/ISCA.2005.52.

[43] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. "Adaptive insertion policies for high performance caching". In: *Proceedings of the 34th annual international symposium on Computer architecture*. ISCA '07. San Diego, California, USA: ACM, 2007, pp. 381–391.

[44] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. "A Case for MLP-Aware Cache Replacement". In: *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 167–178. ISBN: 0-7695-2608-X. DOI: 10.1109/ISCA.2006.5. URL: http://dx.doi.org/10.1109/ISCA.2006.5.

[45] Rafael Reif, A Fan, Kuan-Neng Chen, and S. Das. "Fabrication technologies for three-dimensional integrated circuits". In: *Quality Electronic Design, 2002. Proceedings. International Symposium on*. 2002, pp. 33–37. DOI: 10.1109/ISQED.2002.996687.

[46] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. "DRAMSim2: A Cycle Accurate Memory System Simulator". In: *Computer Architecture Letters* 10.1 (2011), pp. 16 –19. ISSN: 1556-6056. DOI: 10.1109/L-CA.2011.4.

[47] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. "Automatically characterizing large scale program behavior". In: *ACM SIGARCH Computer Architecture News*. Vol. 30. 5. ACM. 2002, pp. 45–57.

[48] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. "Spatial Memory Streaming". In: *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 252–263. ISBN: 0-7695-2608-X. DOI: 10.1109/ISCA.2006.38. URL: http://dx.doi.org/10.1109/ISCA.2006.38.

[49] Madhusudhan Talluri and Mark D. Hill. "Surpassing the TLB Performance of Superpages with Less Operating System Support". In: *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VI. San Jose, California, USA: ACM, 1994, pp. 171–182. ISBN: 0-89791-660-3. DOI: 10.1145/195473.195531. URL: http://doi.acm.org/10.1145/195473.195531.

[50] S Thoziyoor, N Muralimanohar, JH Ahn, and NP Jouppi. "CACTI 5.3". In: *HP Laboratories, Palo Alto, CA* (2008).

[51] Chris Wilkerson, Alaa R. Alameldeen, Zeshan Chishti, Wei Wu, Dinesh Somasekhar, and Shih-lien Lu. "Reducing Cache Power with Low-cost, Multi-bit Error-correcting Codes". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 83–93. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815973. URL: http://doi.acm.org/10.1145/1815961.1815973.

[52] Emmett Witchel, Josh Cates, and Krste Asanović. "Mondrian Memory Protection". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. San Jose, California: ACM, 2002, pp. 304–316. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605429. URL: http://doi.acm.org/10.1145/605397.605429.

[53] Yuan Xie, Gabriel H. Loh, Bryan Black, and Kerry Bernstein. "Design Space Exploration for 3D Architectures". In: *J. Emerg. Technol. Comput. Syst.* 2.2 (Apr. 2006), pp. 65–103. ISSN: 1550-4832. DOI: 10.1145/1148015.1148016. URL: http://doi.acm.org/10.1145/1148015.1148016.

[54] Li Zhao, R. Iyer, R. Illikkal, and D. Newell. "Exploring DRAM cache architectures for CMP server platforms". In: *Computer Design, 2007. ICCD 2007. 25th International Conference on*. 2007, pp. 55–62.