2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00015

Zero Inclusion Victim: Isolating Core Caches from Inclusive Last-level Cache Evictions

Mainak Chaudhuri
Department of CSE, Indian Institute of Technology Kanpur
mainakc@cse.iitk.ac.in

***Abstract*—The most widely used last-level cache (LLC) architecture in the microprocessors has been the inclusive LLC design. The popularity of the inclusive design stems from the bandwidth optimization and simplification it offers to the implementation of the cache coherence protocols. However, inclusive LLCs have always been associated with the curse of inclusion victims. An inclusion victim is a block that must be forcefully replaced from the inner levels of the cache hierarchy when the copy of the block is replaced from the inclusive LLC. This tight coupling between the LLC victims and the inner-level cache contents leads to three major drawbacks. First, live inclusion victims can lead to severe performance degradation depending on the LLC replacement policies. Second, a process can victimize the blocks of another process in an LLC shared by multiple cores and this can be exploited to leak information through well-known eviction-based timing side-channels. An inclusive LLC makes these channels much less noisy due to the presence of inclusion victims which allow the malicious processes to control the contents of the per-core private caches through LLC evictions. Third, to reduce the impact of the aforementioned two drawbacks, the inner-level caches, particularly the mid-level cache in a three-level inclusive cache hierarchy, must be kept small even if a larger mid-level cache could have been beneficial in the absence of inclusion victims.**

We observe that inclusion victims are not fundamental to the inclusion property, but arise due to the way the contents of an inclusive LLC are managed. Motivated by this observation, we introduce a fundamentally new inclusive LLC design named the Zero Inclusion Victim (ZIV) LLC that *guarantees* freedom from inclusion victims while retaining all advantages of an inclusive LLC. This is the first inclusive LLC design proposal to offer such a guarantee, thereby completely isolating the core caches from LLC evictions. We observe that the root cause of inclusion victims is the constraint that an LLC victim must be chosen from the set pointed to by the set indexing function. The ZIV LLC relaxes this constraint only when necessary by efficiently and minimally enabling a global victim selection scheme in the inclusive LLC to avoid generation of inclusion victims. Detailed simulations conducted with a chip-multiprocessor model using multi-programmed and multi-threaded workloads show that the ZIV LLC gracefully supports large mid-level caches (e.g., half the size of the LLC) and delivers performance close to a non-inclusive LLC for different classes of LLC replacement policies. We also show that the ZIV LLC comfortably outperforms the existing related proposals and its performance lead grows with increasing mid-level cache capacity.

***Index Terms*—Inclusive cache hierarchy, back-invalidation, inclusion victim**

I. Introduction

Today's processors use deep multi-level on-chip cache hierarchies to accelerate data delivery to the functional units. In a chip-multiprocessor (CMP), each processing core has one or two levels of private caches and the last level of the cache hierarchy is usually shared across all cores. The relationship between the contents of the shared last-level cache (LLC) and the private caches is an important property of the cache hierarchy in a CMP. Perhaps the most popular and widely deployed such property is the inclusion property [2]. In the context of a CMP, if the private cache contents are always a subset of the shared LLC contents, the LLC is said to implement the inclusion property and the LLC is referred to as an inclusive LLC. Maintaining the inclusion property requires the LLC to implement the following two actions.

1. A block fetched from the off-chip memory system on an LLC miss must be allocated in the LLC in addition to allocating it in the private caches of the requesting core.
2. When a block is evicted from the LLC, all copies of the block resident in the private caches must be forcefully invalidated. The extra interconnect messages sent by the LLC to the private caches for this purpose are usually referred to as the back-invalidations [17], [20]. The private cache blocks thus invalidated are referred to as the inclusion victims [20]. The inclusion victims are not fundamental to the inclusion property, but arise due to certain choices of replacement victims in an inclusive LLC.

The non-inclusive LLCs do not implement the second action [20], [63], [64]. A subset of these, referred to as exclusive or mostly exclusive LLCs, choose not to implement the first action as well, but may choose to allocate a block in the LLC when the block gets shared by at least two cores or gets evicted from the private caches [3], [8], [9], [21], [22], [28], [50], [58], [66]. In this paper, the non-inclusive LLCs studied for the purpose of comparison with inclusive LLCs implement the first action, but do not implement the second action.

In the following, we discuss, in detail, the advantages and disadvantages of an inclusive LLC. Through this discussion, we motivate the need to design an inclusive LLC that *guarantees* freedom from inclusion victims. This fundamentally new kind of inclusive LLC design is the central focus of this paper.

A. Advantages and Shortcomings of Inclusive LLC

The popularity of the inclusive LLC arises from the bandwidth advantage it offers and the simplification it allows to the cache coherence protocol. Before the advent of the CMPs, the cache-coherent multiprocessors were designed using single-core processors. In such systems, a processor receiving a coherence request (either forwarded by the home directory or through a snoop) would try to find out if the requested block is present in its cache hierarchy. If the LLC is inclusive and the LLC lookup is a miss, there is no need to forward the request to the inner levels of the cache hierarchy. This leads to significant reduction in lookup bandwidth requirement of the inner levels. This advantage of inclusive LLCs is usually referred to as snoop filtering. The inclusive LLC's snoop filtering capability comes for free due to its inclusion property.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00015

With the introduction of the CMPs, the snoop filtering advantage of inclusive shared LLCs can still be enjoyed depending on the implementation of the intra-chip coherence protocol. Additionally, the inclusive LLC offers an important simplification to the intra-chip coherence protocol implementation. We discuss it in the following. Scalable CMPs implement directory-based intra-chip coherence protocols. The coherence directory is necessarily an inclusive structure in terms of the blocks it keeps track of because its responsibility is to maintain the coherence status of all blocks that are resident in the private caches. In such a CMP, an external forwarded request coming from another CMP (through home directory or snoop) first looks up the on-chip coherence directory before getting forwarded to any private cache inside the chip. Thus, the coherence directory now takes up the job of snoop filtering. Although the inclusive LLC can continue to offer snoop filtering in such systems, it is no longer necessary. However, in such a CMP, the inclusive LLC offers a significant reduction in the intra-chip coherence protocol complexity compared to the non-inclusive designs as discussed below.

The complexity advantage of an inclusive cache hierarchy arises from the invariant that a lookup hitting in the on-chip coherence directory is guaranteed to hit in the inclusive LLC as well. This is because an entry is present in the directory if and only if the corresponding block is resident in at least one private cache, which in turn implies that the block must be present in the inclusive LLC. In general, there are four possible outcomes when a request looks up the coherence directory and the LLC depending on hit/miss in the directory and hit/miss in the LLC. When the LLC is inclusive, only three of these are possible. The fourth case which is not possible for an inclusive LLC (i.e., hit in directory and miss in LLC) requires selecting a special sharer core for generating the data response of the request leading to new kinds of protocol races and a new set of transient states in a non-inclusive LLC design. In fact, according to the reverse-engineered directory structure of the Skylake-X processor which has a non-inclusive LLC, this processor has introduced a separate coherence directory to keep track of the blocks that are present in the private caches, but not in the LLC [61]. Prior studies have also suggested a similar directory organization having two separate coherence directory structures for non-inclusive LLCs [64]. Irrespective of whether there are two separate on-chip coherence directory structures or there is a unified on-chip coherence directory structure, the coherence protocol for a non-inclusive cache hierarchy is complicated due to the presence of the fourth case. In summary, an inclusive LLC simplifies the intra-chip directory-based coherence protocol significantly in a CMP.

On the other hand, the inclusion victims generated by an inclusive LLC lead to a number of significant drawbacks. If the inclusion victims are in active use at the time of victimization, the number of private cache and LLC misses can increase significantly leading to performance degradation. Given a fixed LLC capacity, the number of inclusion victims is a function of the private cache capacity and the LLC replacement policy. This dependence is captured in Figure 1, which compares the performance achieved by different inclusive (I) and non-inclusive (NI) cache hierarchy configurations for two different LLC replacement policies, namely LRU and Hawkeye [18]. The Hawkeye replacement policy dynamically classifies LLC blocks into cache-friendly and cache-averse by learning from the behavior of the Belady's MIN algorithm [4], [34] at run-time. The data in Figure 1 are generated by simulating an eight-core CMP with a three-level cache hierarchy having an 8 MB 16-way shared LLC, 32 KB 8-way per-core instruction and data L1 caches, and multiple configurations of the private per-core L2 cache. The configurations studied for the per-core L2 cache are 256 KB 8-way, 512 KB 8-way, and 768 KB 12-way with increasing lookup latency. This study uses 72 multi-programmed workload mixes composed using the SPEC CPU 2017 applications.[1]

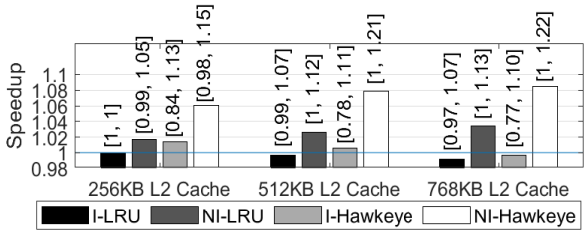

Fig. 1. Performance comparison between inclusive and non-inclusive LLCs.

The speedup numbers in Figure 1 are normalized relative to the configuration with 256 KB L2 cache and inclusive LLC running the LRU policy (I-LRU). On top of each bar, we show the speedup range observed in the workload mixes. We make two important observations from this figure. First, for a given L2 cache capacity, the non-inclusive LLC performs better than the inclusive LLC and this performance gap is much larger in Hawkeye than LRU. The lower ends of the speedup ranges observed for I-Hawkeye show that some of the mixes suffer from very large performance degradation. Thus, the inclusive LLC prevents Hawkeye from realizing its full potential. Second, with increasing L2 cache capacity, the non-inclusive LLC performance improves while the inclusive LLC performance slowly degrades due to gradually increasing volumes of inclusion victims. Thus, the inclusive LLC rules out the use of large L2 caches.

To substantiate the aforementioned two points, Figure 2 shows the normalized inclusion victim counts for the inclusive LLC (normalized to 256 KB L2 cache and I-LRU). To understand the trends in Hawkeye, we have also included the data for an offline implementation of the MIN replacement policy. The MIN replacement policy picks LLC victims based on oracle knowledge of the future global access stream to the L1 cache.[2] We observe that for a given L2 cache capacity, both Hawkeye and MIN policies generate a much larger number of inclusion victims than LRU. The MIN policy victimizes the LLC block that has the furthest reuse in the future among all blocks in a set. In the presence of a circular access pattern, this often leads to the victimization of a block that is recently filled/used. In a circular access pattern, a group of blocks $\{B_i\}$ mapping to the same LLC set is accessed as $(B_1, B_2, \ldots, B_N, B_1, B_2, \ldots)$ with N being larger than the LLC associativity. As can be seen, the block which is accessed most recently has the furthest reuse among the blocks resident in the LLC set. Such recently used LLC victims are highly likely to be resident in the private caches leading to generation of inclusion victims. The Hawkeye policy also inherits this behavior by learning from MIN. In general, any LLC replacement policy attempting to approach the optimal behavior would generate a large volume of inclusion victims as can be inferred by observing the volume of inclusion victims experienced by I-MIN in Figure 2. We also observe from these data that the volume of inclusion victims increases with increasing L2 cache capacity, as expected.

For complete understanding of the aforementioned trends, Figure 3 shows the LLC miss counts for different configurations and LLC replacement policies (normalized to 256 KB L2 cache and I-LRU). The trends correspond well with the speedup numbers. One

[1] Please see Section IV for more details on the simulation environment.

[2] Although the Hawkeye policy tries to learn the optimal behavior based on the portion of the access stream visible locally to the LLC only, it is impossible to correctly model the behavior of MIN for an inclusive LLC based on the LLC access stream. This is because the accesses to the inclusion victims may alter the LLC access stream. Since the global access stream (i.e., the stream of accesses to the L1 cache) remains unaffected by the choice of LLC victims at least for a given instruction schedule, we use that as the input to MIN.

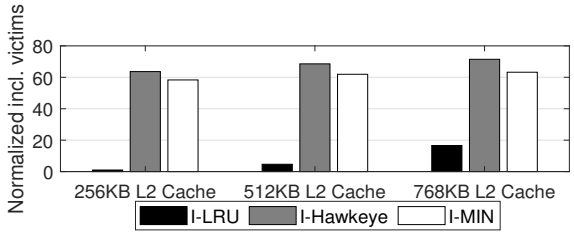

Fig. 2. Normalized count of inclusion victims.

interesting observation is that the numbers of LLC misses in NI-LRU and NI-Hawkeye decrease slightly with increasing L2 cache capacity. This is because as the L2 cache increases in size, a subset of accesses that were missing previously in both L2 cache and LLC now hit in the L2 cache. Figure 4 shows the normalized L2 cache miss counts for different configurations and LLC replacement policies (normalized to 256 KB L2 cache and I-LRU). As expected, the L2 cache miss count is independent of the LLC replacement policy for non-inclusive LLC. For inclusive LLC, the L2 cache miss count is more compared to non-inclusive LLC and varies with LLC replacement policy depending on the volume of inclusion victims.

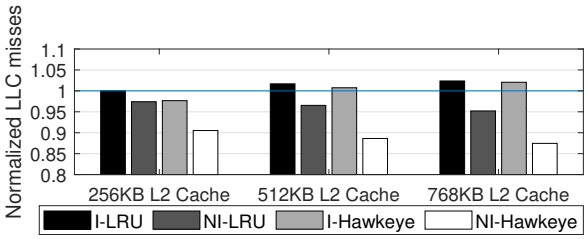

Fig. 3. Normalized LLC miss count.

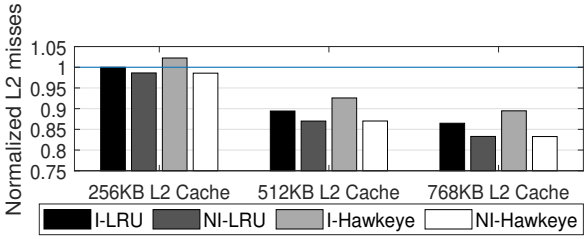

Fig. 4. Normalized L2 cache miss count.

In summary, this preliminary study indicates that the inclusive LLC can be a significant performance problem. It increases the LLC and L2 cache miss counts through generation of inclusion victims. The LLC replacement policies attempting to approach the optimal behavior of the MIN policy fail to achieve their full potential due to a large volume of inclusion victims. The inclusive LLC requires the private cache capacity to be a reasonably small fraction of the LLC capacity to keep the inclusion victim count low. We also note that the inclusion overhead in the LLC arising from replication of blocks between the LLC and the private caches is influenced by the private cache capacity. However, this is inherent in the inclusion property and can be resolved only by making the LLC increasingly exclusive. In this paper, we focus on the problem of inclusion victims only, which is not inherent in the inclusion property, but is only an artifact of how we architect and manage inclusive LLCs.

Apart from the performance problems, inclusion victims can be used in CMPs to enhance the success probability of LLC eviction-based cross-core timing side-channel attacks [10], [11], [16], [23], [29], [30], [38], [39], [44], [51], [54], [62], [65]. An attacker process P scheduled on core C can, in general, evict any block from the shared inclusive LLC. It can engineer a certain memory access pattern that evicts certain blocks belonging to process P' scheduled on core C'. This would lead to invalidation of the copies of these blocks from the private caches of C' as inclusion victims. The future accesses of P' to these blocks would miss in the private caches and the LLC. Thus, these accesses become "visible" to the attacker process. The attacker process can figure out which blocks have been accessed by the victim process P' if the blocks are shared between P and P'; otherwise it can compute the LLC set indices touched by P' by finding out if any of the blocks previously accessed by P have been evicted from the LLC. Either way, P can decipher the access pattern of P' or parts of the physical addresses touched by P' and these can be achieved by P through measurement of the latency of its accesses. In the absence of the inclusion victims, such attacks would be noisy and difficult to mount with low likelihood of success.

B. Contributions and Sketch of the Study

The central contribution of this paper is the design of an inclusive cache hierarchy that is free of inclusion victims. To the best of our knowledge, this is the first inclusive cache hierarchy design to offer such a guarantee. The centerpiece of the design is an inclusive LLC, referred to as the Zero Inclusion Victim (ZIV) LLC, that never generates any inclusion victim while retaining all the advantages of the inclusive LLC such as a simpler coherence substrate in a CMP environment compared to the non-inclusive design alternatives. The ZIV LLC considers the root cause of inclusion victims to be the constraint that an LLC victim must be found from the set pointed to by the set indexing function. The crux of the solution employed by the ZIV LLC lies in efficiently and minimally enabling a global LLC replacement policy only when necessary to avoid generation of inclusion victims (Section III). Our simulation results on multi-programmed and multi-threaded workloads show that the ZIV LLC gracefully supports large L2 caches (e.g., half of LLC capacity) and performs close to a non-inclusive LLC for both LRU and Hawkeye replacement policies (Sections IV and V). The ZIV LLC also comfortably outperforms the related proposals and its performance lead grows with increasing L2 cache capacity. Additionally, by eliminating inclusion victims, the ZIV LLC achieves the much-needed isolation between the core caches and the evictions from an inclusive LLC; inclusive LLC evictions can no longer be used to control the inner-level cache contents. We, however, leave the security analysis of the ZIV LLC to future work. We focus only on the performance aspects of the ZIV LLC in this study.

II. Related Work

The inclusive LLCs have been studied from performance as well as security viewpoints. The non-inclusive cache with an inclusive directory (NCID) proposes to make the data array of the LLC non-inclusive, but maintains an inclusive and over-provisioned tag array [64]. The tag array is decoupled from the data array and maintains pointers to associate a tag to the corresponding data block. This design allows eviction of data blocks from the LLC without generating inclusion victims because the over-provisioned tag array can continue to maintain the tag of the evicted block. This design resembles a non-inclusive LLC equipped with an over-provisioned coherence directory. Unfortunately, such a design cannot enjoy the coherence protocol simplifications offered by an inclusive LLC because it needs to handle the case where a lookup can hit in the inclusive directory, but miss in the non-inclusive cache as already discussed in Section I-A. Similar ideas have been proposed for supporting fill bypass in inclusive LLCs [13]. Specifically, the tags

of the bypassed blocks can be maintained in a separate tag directory to make the overall tag space inclusive.

The temporal locality-aware (TLA) inclusive LLC study is closer in theme to ours as it keeps the tag as well as data arrays inclusive [20]. TLA proposes three techniques to mitigate the performance impact of inclusion victims. The first technique, referred to as temporal locality hint (TLH), informs the LLC about the accesses to the private caches so that the LLC may know about the blocks that have been recently accessed by the cores. This technique, however, requires very high LLC lookup and interconnect bandwidth. The second technique, referred to as early core invalidation (ECI), forcefully invalidates the next LRU victim V from the core caches at the time of replacing the current LRU victim from an LLC set and observes if V receives LLC hits. This helps protect such LLC blocks from getting replaced prematurely. This technique can, however, increase the private cache misses depending on the application behavior. The third technique, referred to as query-based selection (QBS), queries the private caches to find out if an LLC victim candidate is currently resident in the private caches. If yes, the block is moved to the MRU position within the target LLC set and the next victim candidate is considered. This technique has been shown to be the best among the three. Unfortunately, none of these techniques offer any guarantee regarding freedom from inclusion victims. If QBS fails to find a victim that is not resident in the private caches, it victimizes the LRU block in the LLC set, thereby generating an inclusion victim. We note that while QBS was originally proposed for LRU only, it can be combined with any other LLC replacement policy.

The cache hierarchy-aware replacement (CHAR) algorithm makes the replacement policy of the inclusive LLC aware of the problem of inclusion victims [7]. When a block is evicted from the L2 cache, this proposal decides whether it can be prioritized for victimization in the LLC based on the observed access history of the L2 cache victims. In effect, this algorithm attempts to attach higher LLC victimization priority to a subset of blocks already evicted from the L2 cache, thereby reducing the volume of inclusion victims while maintaining high quality in the choice of LLC victims. We will discuss CHAR further in Section III, as its mechanism of inferring liveness of the blocks evicted from the L2 cache can be used in our proposal to improve the efficiency of the basic ZIV LLC design.

A recent study has explored, in detail, the joint influence of inclusion policies and prefetching techniques on LLC management policies [1]. This study shows that a vast majority of the recently proposed LLC management policies deliver good performance only in non-inclusive LLCs and suffer from performance degradation in inclusive LLCs due to inclusion victims.

As already pointed out, inclusion victims help reduce the noise and enhance the success rate of eviction-based timing side-channel attacks in the shared LLC. A large number of proposals has explored mitigation techniques for security vulnerabilities related to shared caches [12], [24]–[26], [31], [40]–[43], [55]–[57], [60], [67]. These proposals improve shared cache security through line locking [25], [56], static or dynamic way-grain cache partitioning [26], [40], [55], combination of line locking and partitioning [31], randomized and dynamic re-mapping of addresses to cache sets [56], [57], dynamic encryption of cache addresses [42], [43], OS-assisted copy-on-access [67], prefetching inclusion victims [41], back-filling cross-core LLC victims into inner-level caches (breaks inclusion and hence, applicable to non-inclusive LLCs only) [12], and restricting certain subsets of inclusion victims [24], [60]. The last category of techniques is closer to our proposal and we discuss these in more detail here.

The relaxed inclusion cache (RIC) [24] relaxes inclusion and hence, eliminates inclusion victims for read-only data (covering read-only cipher data) and thread-private data (covering privately modified sensitive data). Relaxing inclusion for these data types does not require additional cache coherence support because these are either read-only or thread-private. Thread-private modified data must be flushed out of the private caches when a process migrates from one core to another. RIC requires custom compiler support for identifying the read-only and thread-private data. Unfortunately, these analyses are often very conservative. RIC cannot offer security for read/write shared critical data.

The secure hierarchy-aware cache replacement policy (SHARP) [60] proposes to modify the LLC replacement policy. On an LLC miss, it first tries to evict a block that is not resident in the private caches. If there is no such block in the target LLC set, it tries to evict a block that is resident in the requesting core's private cache only. If there is no such block in the target LLC set, it victimizes a random block from the target LLC set. Although SHARP successfully lowers the volume of inclusion victims arising from cross-core LLC conflicts (important for plugging cross-core information leakage), it cannot guarantee freedom from this class of inclusion victims. In contrast, our proposal rids the inclusive LLC of inclusion victims altogether.

III. Design of the ZIV LLC

We present the detailed design of the ZIV LLC in this section. We start with a discussion of the baseline cache hierarchy (Section III-A). Section III-B presents a high-level description of the design proposal. Sections III-C, III-D, and III-E discuss the three primary components of the ZIV LLC in detail.

A. Baseline CMP Cache Hierarchy

The baseline on-chip cache hierarchy is assumed to have a shared inclusive LLC and per-core private caches. The LLC is banked and the banks are distributed over the on-chip interconnect, the exact topology of which is not important for the discussion. The private caches are kept coherent using a directory-based MESI coherence protocol. For space-efficiency, the coherence directory structure is decoupled from the LLC as in a sparse directory [14], [27]. The sparse directory is organized as a tagged set-associative structure. Each entry of the sparse directory keeps track of one privately cached block. To minimize sparse directory eviction, it is sized to have double the number of entries as the number of tags in the last-level private caches (e.g., L2 cache for private L1 and L2 caches) aggregated over all cores. This is referred to as a $2\times$ sparse directory. To improve sparse directory bandwidth, the sparse directory is sliced and a slice of the directory is associated with each LLC bank. The sparse directory slice associated with an LLC bank is responsible for keeping track of all privately cached blocks that map to that LLC bank; the LLC bank is referred to as the home bank of these blocks. A request coming from the private cache hierarchy looks up the home LLC bank and the home sparse directory slice in parallel. Depending on the outcomes of these lookups, the subsequent coherence actions are decided. To keep the sparse directory up-to-date and to avoid extraneous invalidations triggered by coherent writes, a private cache eviction is always notified to the home sparse directory slice [33], [37]. The eviction notices are dataless messages except when a dirty block is evicted generating a traditional writeback to the home LLC bank. Keeping the sparse directory up-to-date can significantly simplify the previous related proposals such as QBS [20] and SHARP [60] because a sparse directory lookup can reveal whether an LLC block is resident in the private caches. A sparse directory entry is freed (i.e., invalidated) when all copies of the block it is tracking are evicted from the private caches.

B. Overview of the ZIV LLC Design

The inclusion victims are generated at the time of LLC replacements due to the way set-associative caches are architected and managed. When filling a block B in the LLC, a replacement candidate must be picked from the LLC set B maps to. This restricts the choice of replacement candidates. The ZIV LLC design exploits the observation that in an inclusive LLC, there is at least one block which is not present in the private caches. This observation follows from the fact that the aggregate private cache capacity must be less than the LLC capacity for an inclusive LLC configuration so that the LLC is left with some space outside the inclusion overhead. As a result, if a global replacement policy could be enabled to replace an LLC block that is not present in the private caches, the cache hierarchy could be made free of inclusion victims. Such a policy needs to be invoked only when the baseline replacement policy selects a victim that is resident in the private caches. The ZIV LLC incorporates enough support into the cache hierarchy to enable a minimal global replacement policy that can be implemented efficiently and invoked selectively. The ZIV design leaves the coherence protocol and the basic architecture of the LLC unaltered.

Figure 5 walks through the functional flow of the ZIV LLC at a high level. An LLC fill to address $A1$ looks up the home LLC bank and the home sparse directory slice. A new directory entry $E1$ is allocated in the target sparse directory set. The LLC block with address $A2$ is replaced from the LLC set to make room for $A1$. This completes the baseline sparse directory and LLC allocation flow. Next, the sparse directory entry of $A2$ is looked up, as is done in the baseline, to determine if copies of the evicted block are resident in the private caches. If there is no copy in the private caches (indicated by a sparse directory miss when looked up with address $A2$), nothing more needs to be done. If there are copies of $A2$ in the private caches, the baseline would have generated back-invalidations at this time. Instead, the ZIV LLC relocates $A2$ to a different LLC set. To do this, a relocation set (RS) is found out such that RS contains at least one block that is not resident in the private caches. Such a set is guaranteed to exist. Once such a set is identified, the ZIV design replaces a block, say $A3$, which is not resident in the private caches and inserts $A2$ in its place in the relocation set. The ZIV LLC leverages the sparse directory entry $E2$ of $A2$ to record the new location of $A2$. The supports for finding a relocation set, relocating a block to a different LLC set, and replacing an appropriate block from the relocation set are the three primary components of the ZIV LLC design. We discuss the architectural support needed to relocate a block from one LLC set to another in Section III-C. In Section III-D, we focus on how to find a "good" relocation set efficiently. Section III-E presents the support needed to implement an appropriate replacement algorithm in the relocation set for inserting a relocated block.

In this context, we note that the Z-cache [46], which uses a skew-associative cache structure [47], employs block relocation. However, the purpose of block relocation in Z-cache is entirely different from ours. Z-cache needs relocation to support a multi-level replacement policy, thereby improving upon the basic replacement decision of the skew-associative caches.

C. Support for Block Relocation

Throughout this section, let us suppose that an LLC block B is relocated from its original location of way $W1$ in set $S1$ of bank K to way $W2$ in set $S2$ of bank K. The relocation process involves reading the block out from the source and writing it at the destination. To distinguish a relocated block from the non-relocated ones within an LLC set, a new $Relocated$ state is added to each

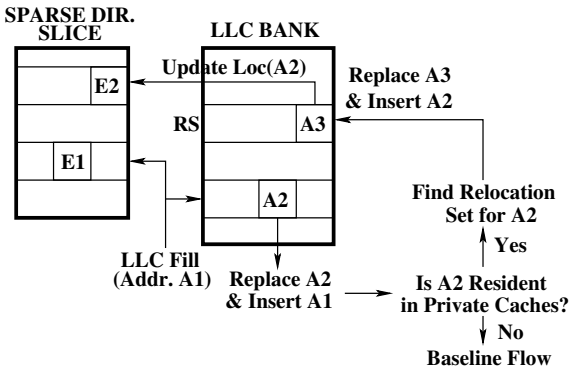

Fig. 5. Overview of the functional flow in the ZIV LLC.

LLC block. As already mentioned, the sparse directory entry E_B of B records the new location of B. In general, it is possible to relocate a block from its home LLC bank to another LLC bank. As a result, each sparse directory entry is extended with a new $Relocated$ state along with the space to hold the tuple <bank id, set id, way id>, which for block B would be $< K, S2, W2 >$ after relocation. Subsequently, block B may observe the following three events while in the $Relocated$ state: (i) an access from a core arising from a private cache miss, (ii) eviction of a copy of it from a private cache, and (iii) its replacement from set $S2$ causing another relocation. In the following, we discuss how the ZIV LLC design handles these three events.

1) Handling Private Cache Misses: A private cache miss request looks up the sparse directory and the LLC in parallel. For an access to the aforementioned relocated block B, the LLC lookup will be to its original pre-relocation set $S1$. The lookup considers only the blocks with the $Relocated$ state off. For B, the lookup misses in LLC set $S1$, but hits in the sparse directory. The sparse directory entry E_B has $Relocated$ state on and emits the current location of B. Next, B is looked up directly in the LLC data array by computing its location using $(S2, W2)$. In the background, the replacement states of set $S2$ are updated appropriately due to this access. Importantly, the critical path of an access to a relocated block has latency equal to $\max$(LLC tag array latency, sparse directory array latency)+LLC data array latency. This is longer compared to the sequential tag and data array latency of an LLC lookup to a non-relocated block only if the sparse directory array latency is bigger than the LLC tag array latency within a bank. With the help of CACTI [15] we find that for a $2\times$ sparse directory and a 1 MB 16-way LLC bank implemented using 22 nm technology nodes, the critical path of an access to a relocated block is lengthened by 1, 2, and 3 cycles respectively for the configurations with 256 KB, 512 KB, and 768 KB L2 caches. This additional delay is a very small fraction of the total round-trip LLC access latency. Also, an access to a relocated block can come to the LLC from only a new sharer core for the block because the existing sharers can access the block from their private caches. Therefore, this latency impact is restricted to only those relocated blocks that get shared by at least two cores.

We note that the latency difference between the accesses to relocated and non-relocated blocks can be observed by a process to infer whether a particular block has suffered an LLC conflict. This, in turn, can be exploited to set up timing side-channels or covert channels. However, in this case, the latency delta is so small that it will be impossible to distinguish it from the latency fluctuations that happen due to various non-deterministic latency components (such as queuing delays) on the round-trip path between the cores and the LLC banks.

2) Handling Private Cache Evictions: When a private cache eviction message reaches the home LLC bank, the sparse directory is looked up and the evicting core is removed from the directory entry. At this point, if the directory entry indicates that there is no more copy of the block left in the private caches, the directory entry should be invalidated. If the directory entry state is $Relocated$, the corresponding LLC block is also invalidated, thereby ending the life of a relocated block. Thus, for the aforementioned relocated block B, this would invalidate the tag at LLC way $W2$ in set $S2$. It is evident that a relocated block that is never shared between two cores and is accessed by only one core C would never receive an access request in the LLC; this is because when C evicts the copy of the block from its private caches, the relocated block is invalidated from the LLC. So, the next access to the block will miss in the LLC. Note that this does not pose a performance problem in the ZIV LLC because the baseline would have evicted this block from the LLC generating inclusion victims, which the ZIV LLC avoids by relocating the block.

If the eviction message from the private cache carries a dirty block back to the LLC (the traditional writeback), the usual action would be to update the LLC block. For relocated blocks, the writeback is directly sent to the memory controller, since the relocated block is being invalidated. On the other hand, if the eviction message from the private cache does not carry any data (i.e., a dataless eviction notification) and the relocated block is being invalidated and is dirty, a writeback is sent to the memory controller.

3) Relocating a Relocated Block: If an LLC block in the $Relocated$ state is replaced, it must be relocated again to a different set because a relocated block is guaranteed to generate inclusion victim(s) on victimization and this must be avoided. The same relocation protocol is followed. However, one added complication related to updating the relocated block's sparse directory entry needs to be addressed. Let us suppose that the aforementioned relocated block B is evicted from set $S2$ and needs to be relocated again. The new location of B needs to be updated in sparse directory entry E_B. However, there is no way to access E_B because we cannot generate the original address of B which is needed to look up the sparse directory. We only have the tag of B, but not the remaining bits of its block address. Provisioning each LLC block with the space to maintain these remaining bits would require a prohibitive amount of storage. We, however, observe that the tag part of a relocated LLC block, in fact, has no use because an access to a relocated block is always initiated through a lookup to the sparse directory, which has its own tag array. Therefore, we can use the tag part of a relocated LLC block to record the location of its sparse directory entry i.e., the tuple <home bank id, set id of E_B, way id of E_B> for the relocated block B. For a 48-bit physical address and an 8 MB 16-way LLC with 64-byte blocks, the LLC tag length is 29 bits assuming simple hash functions for set indexing. LLCs with complex hash functions may require more bits for tags. Now, assuming eight LLC banks, we have up to 26 bits to encode the total number of entries in a sparse directory slice (i.e., set id and way id of a directory entry). Even with 128 LLC banks, we have 22 bits for this purpose. This is far more than what is needed for a practically sized sparse directory slice. For example, our 2× sparse directory has 8192 (1024 sets × 8 ways), 16384 (2048 sets × 8 ways), and 24576 (2048 sets × 12 ways) entries per slice respectively for 256 KB, 512 KB, and 768 KB L2 cache configurations. Therefore, we can comfortably encode the location of the sparse directory entry E_B for a relocated block B using the LLC tag space. When B is relocated again, we can locate E_B using the tag of B and update E_B to record the new location of B.

4) Metadata Overhead: The metadata overhead for supporting block relocation has two components. First, each LLC block needs a new $Relocated$ state, which can be implemented using 2 KB of additional storage for a 1 MB LLC bank with 64-byte blocks. This state can also be implemented with the help of the unused state <Valid=0, Dirty=1>. Second, each sparse directory entry needs a new $Relocated$ state along with bits to store the location of an LLC block. For an 8 MB LLC with 64-byte blocks, this amounts to 18 additional bits per directory entry. The baseline sparse directory entry for an 8-core CMP has a bitvector of size 8 bits and 2 or 3 state bits depending on the cache coherence protocol used (MESI, MOESI, etc.). Therefore, each sparse directory entry gets expanded from 10/11 bits to 28/29 bits. Using CACTI [15] we have verified that a lookup to the expanded sparse directory can be comfortably hidden under the parallel LLC access. In our simulated CMP, each sparse directory slice has 8192, 16384, or 24576 entries when the per-core L2 cache size is 256 KB, 512 KB, or 768 KB respectively. Thus, per LLC bank we need additional 18 KB, 36 KB, or 54 KB of storage for the sparse directory. Adding 2 KB for maintaining the $Relocated$ state per LLC block, we see that the total storage overhead per 1 MB LLC bank ranges from 20 KB to 56 KB, which is less than 6% of the LLC bank capacity.

D. Finding Relocation Sets

A relocation set must satisfy the property that it has at least one block that is not resident in private caches. We will refer to this property as `NotInPrC`. Satisfying this property alone may not, however, make an LLC set a "good" relocation set because a block that is not in any private cache may still have an imminent access. Replacing such a block may hurt performance compared to the baseline. Therefore, it may be desirable to augment this property with more locality-centric properties such as the LRU block in the set is not resident in private caches if the LLC implements LRU policy. We will refer to this property as `LRUNotInPrC`. If the LLC implements the Hawkeye policy, an equivalent property would be that the set has a cache-averse block that is not resident in private caches. The Hawkeye policy uses a three-bit re-reference prediction value (RRPV) [19] to distinguish between the cache-averse (RRPV=7) and cache-friendly (RRPV<7) blocks. Among the cache-friendly blocks, a higher RRPV indicates a higher age (i.e., less recently used). So, this property can be stated as: the set has a block with maximum RRPV that is not resident in private caches. This property, referred to as `MaxRRPVNotInPrC`, can also be used with other LLC replacement policies that employ RRPVs to grade the blocks in a set [19], [59]. The most general desirable property of a relocation set would be that the set has a block that is likely to be dead and not resident in private caches. We will refer to this property as `LikelyDeadNotInPrC`. In the following, we first discuss the general architectural support needed to find a relocation set having a certain property. We note that if the original LLC set can satisfy the target relocation property, there is no need for a relocation. In that case, the replacement policy of the relocation set is invoked directly on the original LLC set to find a different LLC victim.

1) General Architectural Support: Each LLC set is augmented with a bit called the property bit. The property bit of a set is turned on if the set satisfies a property. If multiple properties need to be tracked, each property would need one bit associated with each LLC set. Corresponding to a particular property `P`, the property bits of all sets in an LLC bank together constitute the property vector (PV) for property `P`. This arrangement is shown in Figure 6. If the PV has no bit turned on, that is recorded in a separate bit called $\overline{emptyPV}$. This bit is used to avoid unnecessary PV scans. This bit is computed through a two-level OR logic where the first level has

N k-bit OR gates and the second level has one N-bit OR gate. For an LLC bank with 1024 sets, both N and k could be 32. This logic is triggered when a bit in PV flips from 1 to 0. When a bit in PV flips from 0 to 1, $\overline{emptyPV}$ is also set to 1.

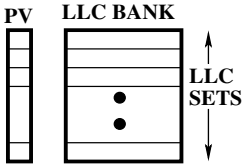

Fig. 6. Property vector (PV) corresponding to a particular property.

For each PV, there is a $nextRS$ register which points to the next round-robin bit location in the PV that is 1. This register effectively holds the index of the next relocation set to be used. A round-robin selection is important to uniformly distribute the relocation load across all eligible LLC sets. Algorithm 1 shows how the decoded $nextRS$ is computed based on the PV and the decoded current RS. Decoded $nextRS$ is a one-hot vector that has a 1 at bit position pointed to by $nextRS$ and 0 in all other positions (for example, a decoded $nextRS$ equal to $00\ldots01000$ implies a $nextRS$ value of 3). Algorithm 1 relies on the fact that the least significant set bit in a binary string x is at the same position as the only set bit in the binary string obtained by ANDing x with its 2's complement. This logic is used in lines 4 and 5 in Algorithm 1 to isolate the next set bit in the PV relative to the position of the current RS. Algorithm 1 is triggered whenever a new relocation operation starts or when $\overline{emptyPV}$ flips from 0 to 1. The algorithm, when triggered, precomputes the decoded $nextRS$ for each PV (there could be multiple PVs corresponding to different properties) given the decoded RS that is last/currently used.

The average repeat interval of relocation operations in an LLC bank as observed in our evaluation is much higher than the latency to execute Algorithm 1 (we discuss detailed synthesis results later in this section). However, there are phases of frequent relocations when the relocation intervals are observed to be smaller than the latency to execute this algorithm. To handle situations where the decoded $nextRS$ is not ready and a new relocation request has arrived at an LLC bank, we maintain an eight-entry FIFO buffer in each LLC bank to hold the blocks waiting to be relocated. This buffer also decouples the actual relocation datapath from the rest of the relocation logic and makes the interface modular. Each buffer entry holds a full LLC block including the address, data, and states. With 64-byte blocks, the total storage needed for this FIFO buffer is slightly over 512 bytes per LLC bank.

To carry out a relocation, the logic associated with the FIFO buffer arbitrates for the LLC bank's write port and contends with the LLC fill logic for LLC write bandwidth. LLC fills are always given higher priority than a pending relocation. To avoid complicating the coherence protocol, we mark the directory entry busy for a block waiting to be relocated. A private cache miss request to such a block is negatively acknowledged. The directory entry comes out of the busy state after the relocation completes. In very rare occasions, the relocation FIFO may fill up (never observed in our evaluation). In such a situation, the LLC controller stops handling the private cache miss requests. We note that stalling private cache miss requests cannot lead to a deadlock because the progress of pending relocations does not depend on the progress of private cache miss requests.

In extremely rare situations, it may happen that all blocks in an LLC bank are resident in private caches. This situation may arise if there is an extremely uneven distribution of active blocks across

Algorithm 1: Computation of decoded $nextRS$

Input: PV and $decoded\_RS$
Output: $decoded\_nextRS$

```
/* Generate mask = 11...100...0 with
   the cross-over from 0 to 1
   happening right after the current
   RS position                          */
```
1 $mask \leftarrow ((\sim decoded\_RS) + 1)\ \&\ (\sim decoded\_RS)$

```
/* Extract upper and lower portions
   of PV split right after the
   current RS position                  */
```
2 $upperPV \leftarrow PV\ \&\ mask$
3 $lowerPV \leftarrow PV\ \&\ \sim mask$

```
/* Find the next set bit position in
   upperPV                              */
```
4 $decoded\_nextRS\_upper \leftarrow upperPV\ \&\ ((\sim upperPV) + 1)$

```
/* Find the next set bit position in
   lowerPV                              */
```
5 $decoded\_nextRS\_lower \leftarrow lowerPV\ \&\ ((\sim lowerPV) + 1)$

```
/* Compute the final output             */
```
6 **if** *decoded_nextRS_upper == 0* **then**
7 $\quad decoded\_nextRS \leftarrow decoded\_nextRS\_lower$
8 **else**
9 $\quad decoded\_nextRS \leftarrow decoded\_nextRS\_upper$

LLC banks. While we have not encountered this in our evaluation, this situation is handled by relocating a block from one LLC bank to another. The home LLC bank may query its one-hop neighbor LLC banks first to find a relocation set. If no relocation set is found, it can query randomly picked LLC banks. Once the destination LLC bank is found, the relocation in that bank proceeds according to the aforementioned intra-bank relocation protocol. We note that relocations to non-home banks increase the latency difference between the accesses to the relocated and non-relocated blocks. If such a relocation is due to a cross-core conflict in the LLC, it can be decided whether the newly filled LLC block or the LLC victim should be relocated. Such a decision allows us to balance the volume of non-home relocations across all cores. Another alternative could be to choose the relocation candidate (from among the newly filled LLC block and the LLC victim) randomly following a certain probability distribution.

Having discussed the general relocation protocols, we now turn to discuss the additional support needed for implementing specific relocation set properties.

2) Support for `Invalid`*:* The `Invalid` property bit of an LLC set is turned on if the set has an invalid way. The `Invalid` PV is always consulted first at the time of deciding a relocation set. The metadata overhead is 1024 bits for the PV per 1 MB 16-way LLC bank.

3) Support for `NotInPrC`*:* Each LLC block is provisioned with a $NotInPrC$ state bit to record if the block is resident in private caches or not. When a private cache eviction notice or a writeback comes to the home sparse directory slice and the directory entry indicates that no other copy of the block is resident in the private caches, the $NotInPrC$ bit of the LLC block is set to 1. On

an access from a core to an LLC block, the $NotInPrC$ bit of the block is reset. An LLC set's `NotInPrC` property bit is set to 1 if at least one LLC block in the set has $NotInPrC$ bit set to 1. The metadata overhead per 1 MB 16-way LLC bank is 2 KB for the per-block $NotInPrC$ state bit and 1024 bits for the PV. The ZIV LLC design implementing the `NotInPrC` property for identifying relocation sets also needs to have the `Invalid` PV adding another 1024 bits per LLC bank to the metadata overhead. The $nextRS$ of the `NotInPrC` PV is used for relocation if the `Invalid` PV is empty; otherwise the $nextRS$ of the `Invalid` PV is used for relocation.

4) Support for `LRUNotInPrC`*:* Three PVs need to be maintained: one for `Invalid`, one for `NotInPrC`, and one for `LRUNotInPrC`. Each LLC block also needs the $NotInPrC$ state bit. On an access to an LLC set or on a replacement from an LLC set, if the block entering the LRU position has the $NotInPrC$ state bit set to 1, the LLC set's property bit for `LRUNotInPrC` is set to 1; otherwise the property bit is set to 0. When updating the $NotInPrC$ state bit of an LLC block, if the block is in the LRU position, the LLC set's property bit for `LRUNotInPrC` is also updated accordingly.

When selecting a relocation set, the first priority is given to the set pointed to by $nextRS$ of `Invalid`; if the `Invalid` PV is empty, the $nextRS$ of `LRUNotInPrC` gets the next higher priority. If the PV for `LRUNotInPrC` is also empty, the set pointed to by $nextRS$ of the `NotInPrC` PV is picked as the relocation target. At each of the three priority levels, the original source LLC set is first checked to see if it satisfies the corresponding relocation set property of that priority level (i.e., `Invalid` first, `LRUNotInPrC` next, and then `NotInPrC`); if yes, no relocation is required and the relocation set's victim selection algorithm is executed in the original LLC set to pick a different LLC victim. Thus, the relocation set selection order is as follows: original LLC set satisfying `Invalid` (this check is done anyway as part of the baseline LLC replacement policy), global LLC set satisfying `Invalid`, original LLC set satisfying `LRUNotInPrC`, global LLC set satisfying `LRUNotInPrC`, original LLC set satisfying `NotInPrC`, global LLC set satisfying `NotInPrC`. The metadata overhead per 1 MB 16-way LLC bank is 2 KB for the per-block $NotInPrC$ state bit and 3072 bits for the three PVs.

5) Support for `MaxRRPVNotInPrC`*:* The support for this property is similar to `LRUNotInPrC`. The PV for `LRUNotInPrC` is replaced by the PV for `MaxRRPVNotInPrC`. In the operations discussed for `LRUNotInPrC`, if we replace the LRU position by maximum RRPV, we obtain the operations to be done for `MaxRRPVNotInPrC`. The metadata overhead is same as `LRUNotInPrC`.

6) Support for `LikelyDeadNotInPrC`*:* In this case also, we need three PVs: one for `Invalid`, one for `NotInPrC`, and one for `LikelyDeadNotInPrC`. Each LLC block is provisioned with a $LikelyDead$ state bit and a $NotInPrC$ state bit. If an LLC set has at least one block with both $LikelyDead$ and $NotInPrC$ state bits set to 1, the LLC set's property bit for `LikelyDeadNotInPrC` is set to 1. When selecting a relocation set, the set pointed to by $nextRS$ of the `LikelyDeadNotInPrC` PV is picked provided the `Invalid` PV is empty; if the `LikelyDeadNotInPrC` PV is empty, the set pointed to by $nextRS$ of the `NotInPrC` PV is picked. At each priority level, the original source LLC set is first checked to see if it satisfies the corresponding relocation set property of that priority level; if yes, no relocation is required and the relocation set's victim selection algorithm is executed in the original LLC set to pick a different LLC victim.

We have already discussed how the $NotInPrC$ state of an LLC block is operated on. To update the $LikelyDead$ state of an LLC block, we need to identify the likely dead LLC blocks. A large body of work has explored ways to identify likely dead blocks in the LLC; we would like to reuse one of these proposals. However, to suite our purpose, we would like to identify the likely dead blocks only from among the blocks that are not resident in the private caches. There is no use of identifying other dead blocks because we cannot replace them from the LLC. The cache hierarchy-aware replacement (CHAR) proposal [7] comes closest to this requirement. This proposal attempts to infer likely death of a block from LLC's viewpoint when it is evicted from the private cache of a core. The inferred dead blocks are prioritized for replacement in the LLC. In our proposal, however, the LLC does not exercise the dead block-based replacement as the baseline policy. Instead, it needs to identify just enough likely dead blocks in the LLC to support relocation because the LLC sets satisfying the `LikelyDeadNotInPrC` property are important for supporting relocation only and serve no other purpose. Therefore, our adaptation of CHAR can use a dynamically adjusted confidence level for dead block inference depending on the relocation demand. We discuss our design in the following.

When a block is evicted from the L2 cache, if it is not present in the L1 caches[3], an eviction notice or a writeback is sent to the home LLC bank as in the baseline. At this time, we invoke CHAR's death inference mechanism and if the block is inferred dead, one bit in the header of the eviction notice or writeback message is used to convey this information to the LLC bank. When an LLC bank receives a private cache eviction notice or a writeback message, it checks the dead inference bit in the message and accordingly sets the $LikelyDead$ state of the corresponding LLC block and the `LikelyDeadNotInPrC` property bit of the LLC set provided the block is not shared (a shared block is not inferred dead). When an LLC block is accessed due to a private cache miss, its $LikelyDead$ state is reset to 0.

CHAR categorizes a block evicted from the L2 cache into a number of groups based on attributes such as (i) whether the block was brought to the private caches through a prefetch or a demand request, (ii) whether the block was filled into the private caches through an LLC hit or miss, (iii) the number of L2 cache demand reuses experienced by the block, and (iv) whether the block is dirty. For each group, CHAR collects the number of L2 cache evictions and the number of recalls from the LLC using two counters. If the ratio of the recall counter to the eviction counter of a group is below a threshold τ, that means the blocks in that group are rarely recalled from the LLC and hence, are likely to be dead. Thus, if a block evicted from the L2 cache is classified to be in that group, it is inferred dead.

Our adaptation of CHAR adjusts the death inference threshold τ dynamically. We restrict τ to the values of the form $\frac{1}{2^d}$ so that the comparison $RecallCount/EvictionCount < \tau$ can be implemented as $(RecallCount << d) < EvictionCount$. We initialize d to six in all L2 cache controllers and LLC banks. When a relocation request in an LLC bank finds that the PV for `LikelyDeadNotInPrC` is empty and $d > 1$, this event is recorded in a threshold request bitvector (TRBV) of length equal to the number of cores; all bit positions of TRBV are set to 1. The LLC bank also decrements d by one. After this, when the LLC bank receives a private cache eviction notice or a writeback from a core i and if TRBV[i] is 1, it piggybacks the new value of d in the acknowledgment message for the eviction notice or writeback. At this time, TRBV[i] is reset to 0.

[3] The private L1 and L2 caches are non-inclusive in our CMP model.

When an L2 cache controller receives a new value of d, it overwrites its own value of d with this new value provided the new value is less than its own value. Thus, d can only decrease starting from six down to one. Since different LLC banks can have different d values at run-time, this check is needed for correct dynamics in threshold setting. An LLC bank maintains a sufficiently long interval between two consecutive decrements in d to make sure that the effect of the new threshold value has been taken into account before requesting another change. In our design, this interval is set to 4096 private cache eviction notices. Overall, if the number of relocations is low, the dead block inference operates with a small threshold τ and is likely to be more accurate. It is necessary to periodically reset d back to six in all LLC banks and L2 caches to take care of phase changes. The high-level flow is summarized in Figure 7.

CHAR requires two state bits per L2 cache block for carrying out the classification of the blocks into groups [7]. So, the primary overhead of supporting the `LikelyDeadNotInPrC` property stems from the following: (i) two additional state bits in each L2 cache block (2 KB overhead per core for 512 KB L2 cache), (ii) two additional state bits in each LLC block (4 KB overhead for each 1 MB LLC bank), (iii) three PVs (3072 bits per LLC bank), (iv) one additional header bit in private cache eviction notice and writeback messages for conveying dead inference outcome, (v) three bits in eviction notice and writeback acknowledgment messages for updating d, and (vi) one bit in private cache miss response to convey LLC hit/miss needed for the block classification algorithm used by CHAR. Total storage overhead per LLC bank and per core taken together is 6.375 KB.

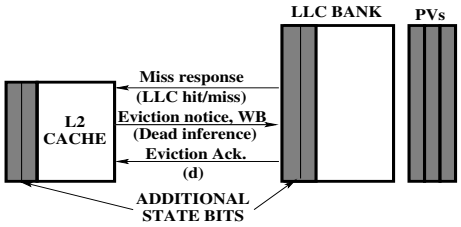

Fig. 7. High-level flow in our adaptation of CHAR. The additional storage is shown in grey. The additional pieces of information piggybacked in the messages are shown within parentheses.

7) Support for `MaxRRPVLikelyDeadNotInPrC`*:* We implement the property `MaxRRPVLikelyDeadNotInPrC` so that we can combine the inferences of Hawkeye and CHAR. Hawkeye is based on LLC access stream only, but uses optimal behavior for training the inference mechanism. On the other hand, CHAR takes into account L2 cache reuses, but employs a rudimentary inference mechanism. Implementing this property requires four PVs: `Invalid`, `MaxRRPVNotInPrC`, `LikelyDeadNotInPrC`, and `NotInPrC`. In relocation set selection, the $nextRS$ of `Invalid` is given the highest priority, the $nextRS$ of `MaxRRPVNotInPrC` is given the next priority (this choice attempts to replace one of the cache-averse blocks ear-marked by Hawkeye provided it is not privately cached) if the `Invalid` PV is empty; $nextRS$ of `LikelyDeadNotInPrC` is given the next priority if the `MaxRRPVNotInPrC` PV is empty; $nextRS$ of `NotInPrC` is picked if all other PVs are empty. At each priority level, the source original LLC set is first checked to see if it satisfies the corresponding relocation set property of that priority level (i.e., `Invalid`, `MaxRRPVNotInPrC`, `LikelyDeadNotInPrC`, `NotInPrC` in that order) before looking for a global relocation set for that level. The overhead of implementing this property includes all the overhead of implementing the `LikelyDeadNotInPrC` property. It requires one additional PV (1024 bits per LLC bank).

8) Critical Path and Area Estimates: We synthesize the module that updates $\overline{emptyPV}$ and PV, and computes the decoded $nextRS$ for the ZIV LLC design. We use a 45 nm TSMC process and the Synopsys Design Compiler in the ultra-optimization mode (uses `compile_ultra` command). The critical path through a purely combinational implementation of the logic meets the timing target of three cycles at 4 GHz clock frequency. The `NotInPrC` set property needs two PVs and the associated logic to update $\overline{emptyPV}$ and decoded $nextRS$ for each PV. Its total area overhead per 1 MB LLC bank is 0.045 mm^2. The `LRUNotInPrC` and the `LikelyDeadNotInPrC` set properties need three PVs; its total area overhead per 1 MB LLC bank is 0.078 mm^2. The `MaxRRPVLikelyDeadNotInPrC` set property needs four PVs and its total area overhead per 1 MB LLC bank is 0.099 mm^2.

E. Replacement Policy in Relocation Sets

Once a relocation set is selected, an appropriate block needs to be replaced from this set to make room for the relocated block. Since the relocation set is selected based on a certain underlying priority order among the implemented LLC set properties, the same priority order is used within the relocation set to evict a block. For all cases, an invalid block, if available, is evicted first from the relocation set honoring the highest priority of the `Invalid` set property. If there is no invalid way in the relocation set, the next lower priority level dictates the replacement policy within the relocation set as discussed below.

If the LLC implements the `NotInPrC` or `LRUNotInPrC` set property, the replacement policy in the relocation set would victimize the $NotInPrC$ block closest to the LRU position. On the other hand, if the LLC implements the `MaxRRPVNotInPrC` set property, the replacement policy in the relocation set would victimize the $NotInPrC$ block with as high an RRPV as possible. When the LLC implements the `LikelyDeadNotInPrC` set property and the baseline LLC policy is LRU, the replacement policy in the relocation set would victimize the $LikelyDead$ block closest to the LRU position; if there is no $LikelyDead$ block in the set, it would victimize the $NotInPrC$ block closest to the LRU position. If, on the other hand, the LLC implements the `MaxRRPVLikelyDeadNotInPrC` set property, the replacement policy in the relocation set would first attempt to victimize the $NotInPrC$ block with RRPV=7 (corresponds to one of the cache averse blocks identified by Hawkeye provided it is not privately cached); if there is no such block in the set, it would victimize the $LikelyDead$ block with as high an RRPV as possible; if there is no $LikelyDead$ block in the set, it would victimize the $NotInPrC$ block with as high an RRPV as possible. The LLC controller needs to have support for executing two different replacement algorithms: one for baseline replacement invoked in the case of traditional LLC fills and another for filling relocated blocks into the relocation sets.

F. Handling Sparse Directory Eviction

The ZIV LLC design relies on the sparse directory structure for storing the location of a relocated LLC block. Since the sparse directory is a set-associative tagged structure, it may have to replace valid entries if needed. The traditional protocol for handling a sparse directory eviction back-invalidates the privately cached copies of the block that the evicted directory entry is tracking. Additionally, in the ZIV LLC design, if the evicted sparse directory entry is tracking a relocated block, the relocated block must also be invalidated from the LLC. This invalidation is necessary because there would be no space to track the location of the relocated block after the directory entry is evicted.

The recently proposed Zero Directory Eviction Victim (ZeroDEV) protocol shows how to eliminate the back-invalidations when a sparse directory entry is evicted [6]. It accommodates the evicted directory entries in the LLC. When a directory entry is evicted from the LLC, it avoids generation of back-invalidations by incorporating novel extensions to the baseline cache coherence protocol. The ZIV LLC design seamlessly integrates with the ZeroDEV protocol. In Section V, we show that this integrated design maintains its performance leads even in the configurations with under-provisioned sparse directory structures.

G. Summary of the ZIV LLC Design

The ZIV design rids the inclusive LLC of inclusion victims by relocating the LLC victims that have copies resident in the private caches. We have proposed several design options of varying complexity for choosing a "good" LLC set where such a victim can be relocated to. The relocation set properties are the primary performance determinants of the ZIV LLC design because otherwise all ZIV design variants are free of inclusion victims. The relocation set properties determine the quality of the LLC victim in the case a relocation is needed. The ZIV LLC leverages the on-chip coherence directory for maintaining the new location of the relocated LLC blocks. The metadata and logic area overheads of the ZIV LLC design are small.

IV. Simulation Environment

We use the Multi2Sim simulation infrastructure [52] to evaluate our proposal. Table I lists the parameters of the simulated CMP. We use CACTI [15] to determine the lookup latency of the cache arrays shown in Table I; the round-trip latency for LLC lookup is a few tens of cycles as it additionally includes interconnect latency, waiting time at interface queues, etc.. To evaluate the scalability of our proposal, we also model a 128-core system having a 32 MB 16-way shared LLC, 128 KB 8-way per-core L2 cache and 32 KB 8-way per-core instruction and data L1 caches. We use this model to evaluate a server application.

TABLE I
Baseline simulation environment

CPU core (eight in number, dynamically scheduled, x86, 4 GHz)
224-entry ROB, 128-entry LSQ, iL1 & dL1 cache: 32 KB/8-way/LRU, L2 cache: 256 KB/8-way/LRU/4 cycles, 512 KB/8-way/LRU/5 cycles, 768 KB/12-way/LRU/6 cycles
Shared LLC, sparse directory, interconnect
LLC: 8 MB/16-way/8 banks/LRU or Hawkeye/2-cycle tag lookup/ 5-cycle data access/64-byte block. Sparse dir.: 2×, 8-way, 1-bit NRU. Interconnect: 2D mesh, 1 ns routing delay, 0.5 ns link latency.
Main memory (modeled using DRAMSim2 [45])
Two single-channel DDR3-2133 controllers, 64-bit channel, BL=8, two ranks per channel, x8 DRAM devices, eight banks, 1 KB row buffer per bank, latency parameters: 14-14-14-35

We evaluate five ZIV LLC designs implemented using different relocation set properties discussed in Section III-D. The properties `NotInPrC`, `LRUNotInPrC`, and `LikelyDeadNotInPrC` are evaluated in the context of the LRU policy, while the properties `MaxRRPVNotInPrC` and `MaxRRPVLikelyDeadNotInPrC` are evaluated in the context of the Hawkeye policy. We compare our proposals to QBS [20] and SHARP [60] implemented on top of LRU and Hawkeye policies. For both, the baseline policy determines the order of search for the victims in each step (e.g., LRU to MRU or maximum RRPV to minimum RRPV).

We conduct our evaluations on multi-programmed as well as multi-threaded applications. For homogeneous multi-programming (multiple copies of the same application run on the CMP), we select all 36 application-input pairs from the SPEC CPU 2017 suite and they use the ref inputs. For heterogeneous multi-programming, we create 36 random workload mixes by drawing eight different application-input pairs at a time. We ensure that each application-input pair is represented an equal number of times in the heterogeneous workload mixes to avoid any bias toward any particular application. Each application is run for a representative segment of 500M dynamic instructions selected using SimPoint [49]. The early-finishing applications continue to run until each application completes its representative segment. Statistics are reported by considering only the representative segment of each application. A small set of multi-threaded applications is selected with varying performance-sensitivity to inclusion victims. We draw canneal, facesim, vips from the PARSEC suite [5] and 316.applu from the the SPEC OMP 2001 suite. We also evaluate TPC-E on MySQL configured with a 10 GB database, 2 GB buffer pool, 100 clients and run on a 128-core system for five billion instructions. The TPC-E simulation is done by replaying an instruction trace of the MySQL server collected using Pin [32].

V. Simulation Results

We analyze the performance of the multi-programmed and multi-threaded workloads in Sections V-A and V-B, respectively. Section V-C characterizes the frequency of block relocation and its energy expense.

A. Performance Analysis of Multi-programmed Workloads

All results presented in this section are normalized to the configuration having 256 KB L2 cache and an inclusive LLC with LRU policy. This allows us to easily compare different sets of results along different dimensions. We first discuss the results when the baseline LLC policy is LRU. Figure 8 shows the performance results. For each L2 cache capacity, the leftmost two bars correspond to the baseline inclusive and non-inclusive LLCs. The next two bars show the performance of QBS and SHARP running with an inclusive LLC. The last three bars are ZIV LLC designs (we have shortened `LikelyDeadNotInPrC` to `LikelyDead`). With a 256 KB L2 cache, QBS and SHARP deliver performance close to the non-inclusive LLC, but fail to scale their performance up as the L2 cache capacity increases. This is because the number of LLC blocks not resident in the private caches drops rapidly with increasing L2 cache capacity and as a result, it becomes increasingly important to select the appropriate LLC victims from this small set of LLC blocks that are not resident in the private caches. The ZIV LLC designs implementing the `NotInPrC` and `LRUNotInPrC` relocation properties perform close to QBS and SHARP. However, `NotInPrC` and `LRUNotInPrC` guarantee freedom from inclusion victims, while QBS and SHARP do not offer any such guarantee.

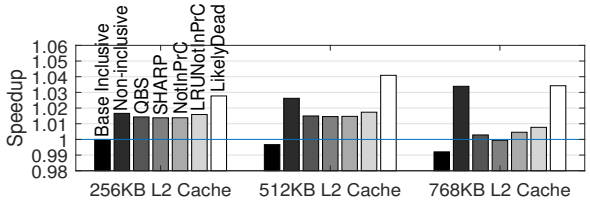

Fig. 8. Performance of multi-programmed workloads with LRU as the baseline LLC policy.

The ZIV LLC design implementing the `LikelyDead` property performs the best across the board. Surprisingly, this design even outperforms the non-inclusive LLC for 256 KB and 512 KB L2 cache capacity points. To understand this, we note that there are

three reasons for performance improvement in any ZIV LLC design compared to the baseline inclusive LLC design: (i) elimination of inclusion victims, (ii) good selection of LLC victims from relocation sets, and (iii) bigger set of choices in LLC victim selection at the time of relocation due to the global nature of relocation set selection. Any additional performance improvement compared to non-inclusive LLC arises from (ii) and (iii). On the other hand, to achieve (i), the ZIV LLC design would sometimes be forced to choose LLC victims that are poorer in quality compared to the baseline LLC policy. This effect is visible in the performance of `NotInPrC` and `LRUNotInPrC` for the 512 KB and 768 KB L2 cache capacity points and would be more prominent when we discuss the results for the Hawkeye policy. To separate the contributions of (ii) and (iii), we found that when we let the non-inclusive LLC change its replacement policy from LRU to the `LikelyDead` policy without any relocation (replaces the $LikelyDead$ block closest to the LRU position), it bridged roughly half of the gap between non-inclusive LLC and the ZIV LLC design exercising the `LikelyDead` property for the 256 KB and 512 KB L2 cache capacity points. The remaining performance gap can be attributed to (iii). Overall, the ZIV LLC with the `LikelyDead` property scales gracefully all the way to 768 KB L2 cache capacity either meeting or surpassing the performance of a non-inclusive LLC. However, since its performance with a 768 KB L2 cache is worse than that with a 512 KB L2 cache, these results do not justify supporting a 768 KB L2 cache with a ZIV inclusive LLC.

An interesting comparison point is the inclusive LLC design that selects a $LikelyDead$ block (e.g., the one closest to the LRU position) inferred by CHAR as the victim in the target LLC set if the victim picked by the baseline policy has privately cached copies; if there is no $LikelyDead$ block in the target set or if the baseline victim has no privately cached copies, the baseline LLC policy is used. While this policy reduces the volume of inclusion victims significantly compared to the baseline, it is not free of inclusion victims. We refer to this policy as CHAROnBase; this policy can be implemented on top of any baseline LLC policy. We find that across the board, CHAROnBase performs better than QBS, SHARP, and the ZIV LLC designs exercising the `NotInPrC` and `LRUNotInPrC` properties. However, this policy falls significantly short of the ZIV LLC design exercising the `LikelyDead` property for the configurations with 512 KB and 768 KB L2 cache. The performance of CHAROnBase suffers due to the fact that the baseline policy is used if there is no $LikelyDead$ block in the target set, even though there may be $LikelyDead$ blocks in other LLC sets. This result further underscores the importance of choosing a good global victim in the ZIV LLC design as the L2 cache capacity grows.

Figure 9 shows the detailed speedup achieved by the ZIV LLC design exercising the `LikelyDead` property for the configuration with 512 KB L2 cache. For the homogeneous mixes, we also note the top few applications. The average speedup and the speedup range for each type of mix is also noted. The heterogeneous mixes tend to benefit more because the memory-intensive applications in a mix can generate a lot of inclusion victims for all the applications in the mix, thereby significantly degrading the performance of the applications that fit in their private caches. We note that 12% of LLC misses, on average, require relocation with the maximum being 33% across all these mixes.

Figure 10 shows the normalized LLC misses (upper panel) and the L2 cache misses (lower panel) averaged across the multi-programmed workloads. The LLC miss data closely follow the performance trend: the ZIV LLC design exercising the `LikelyDead` property saves more LLC misses than the non-inclusive LLC for the 256 KB and 512 KB L2 cache capacity points. The L2 cache miss

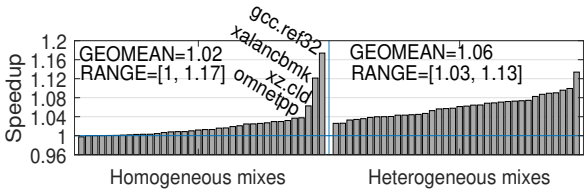

Fig. 9. Performance of the ZIV LLC design exercising the `LikelyDead` property for the configuration with 512 KB L2 cache.

data show that QBS, SHARP, and the ZIV LLC designs all save nearly the same number of L2 cache misses as the non-inclusive design across all three L2 cache capacity points. These results indicate that both QBS and SHARP successfully eliminate most of the inclusion victims, although they cannot guarantee freedom from inclusion victims. As already pointed out, these two proposals suffer in terms of performance due to their choice of LLC victims.

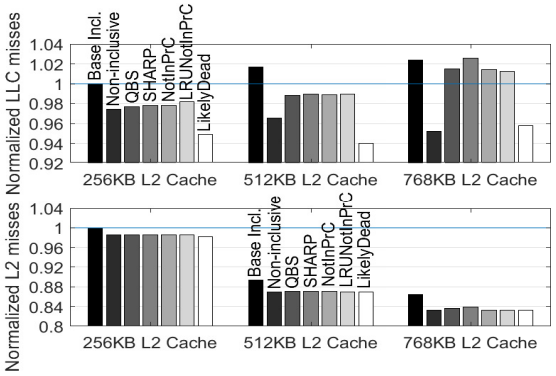

Fig. 10. Normalized LLC and L2 cache misses for the multi-programmed workloads with LRU as the baseline LLC policy.

Next, we discuss the performance results when the LLC exercises Hawkeye as the baseline policy. Figure 11 shows the performance results. As already mentioned, these results are also normalized to the configuration having 256 KB L2 cache and inclusive LLC with LRU policy. For each L2 cache capacity, the rightmost two bars present the performance of the ZIV LLC designs implementing the `MaxRRPVNotInPrC` (shortened to `MRNotInPrC`) and `MaxRRPVLikelyDeadNotInPrC` (shortened to `MRLikelyDead`) properties. The general performance trends are similar to what we have seen with the LRU baseline. Across the board, the ZIV LLC design implementing the `MRLikelyDead` property offers the best performance coming close to the non-inclusive LLC performance for the 256 KB and 512 KB L2 cache capacity points. This property combines the likely dead inference of CHAR with the cache-friendly/cache-averse classification of Hawkeye and offers roughly a percentage more performance on average compared to the ZIV LLC design exercising the `MRNotInPrC` property that relies only on the classification done by Hawkeye.

We also observe that the `MRLikelyDead` property cannot outperform the non-inclusive LLC design for any L2 cache capacity point. This is unlike what we have seen when the baseline policy was LRU. This is because the `MRLikelyDead` property is not sufficiently good to compensate the performance loss arising from the occasional poor choice of LLC victims needed to eliminate the inclusion victims; the cost of these low-quality victims is very high in this case because the baseline Hawkeye policy is significantly better than LRU. The results at the 768 KB L2 cache capacity point show that we need much smarter relocation properties to

justify this L2 cache capacity. Overall, the ZIV LLC design with the `MRLikelyDead` property scales well in terms of performance up to 512 KB L2 cache capacity. Figure 12 shows the detailed speedup for the ZIV LLC design implementing the `MRLikelyDead` property for the configuration with 512 KB L2 cache.

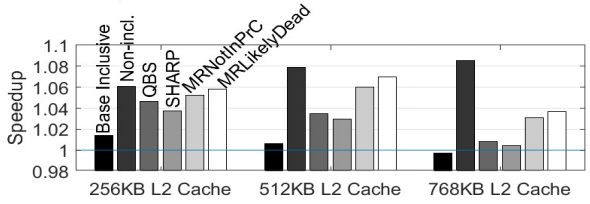

Fig. 11. Performance of multi-programmed workloads with Hawkeye as the baseline LLC policy.

Figure 13 shows the normalized LLC misses (upper panel) and L2 cache misses (lower panel) for the multi-programmed workloads when the baseline policy is Hawkeye. The trends in the LLC miss count follow the performance trend closely. The L2 cache miss trends are same as what we saw when the baseline LLC policy was LRU.

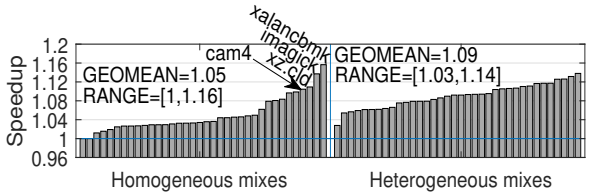

Fig. 12. Performance of the ZIV LLC design implementing the `MRLikelyDead` property for the configuration with 512 KB L2 cache.

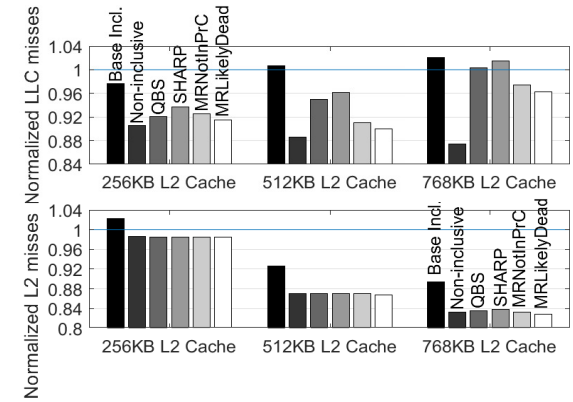

Fig. 13. Normalized LLC and L2 cache misses for the multi-programmed workloads with Hawkeye as the baseline LLC policy.

Sensitivity to LLC Capacity: The results discussed so far have used an 8 MB LLC for an 8-core CMP. Figure 14 summarizes the average performance of the multi-programmed workloads for a configuration with a 16 MB LLC and per-core 1 MB L2 cache. The results are normalized with respect to the same baseline used so far, namely the configuration with an 8 MB inclusive LLC exercising the LRU policy and 256 KB L2 cache per core. The group of bars on the left shows the results when the 16 MB LLC's replacement policy is LRU, while the group of bars on the right corresponds to the Hawkeye policy. For the LRU policy, the ZIV LLC design exercising the `LikelyDead` property continues to surpass the non-inclusive design in performance as we observed for an 8 MB LLC. For the Hawkeye policy, the ZIV LLC designs exercising the `MRNotInPrC` and `MRLikelyDead` properties perform close to the non-inclusive design.

Fig. 14. Performance of the multi-programmed workloads for a configuration with a 16 MB LLC shared among 8 cores and per-core 1 MB L2 cache.

Sensitivity to Sparse Directory Size: Figure 15 shows the performance of the multi-programmed workloads as the sparse directory size is varied from $2\times$ to $\frac{1}{4}\times$ for the configuration with 8 MB LLC, Hawkeye as the baseline LLC policy, and 256 KB per-core L2 cache. The left half shows the performance variation when using the traditional MESI coherence protocol. The baseline inclusive LLC, non-inclusive LLC, and the ZIV LLC design with the `MRLikelyDead` property suffer from performance degradation as the sparse directory size is reduced; as expected, the non-inclusive LLC gradually loses its performance advantage over the baseline inclusive LLC due to the back-invalidations arising from the sparse directory evictions. However, the ZIV LLC design continues to closely follow the performance of the non-inclusive LLC. The right half of the figure shows that the ZeroDEV coherence protocol [6] succeeds in making the performance nearly invariant of the sparse directory size. It achieves this by eliminating the back-invalidations that arise from sparse directory evictions. These results confirm that the ZIV LLC design integrates seamlessly with the ZeroDEV protocol.

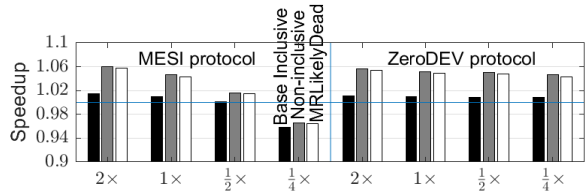

Fig. 15. Sensitivity to sparse directory size for the multi-programmed workloads with Hawkeye as the baseline LLC policy and 256 KB L2 cache.

B. Performance Analysis of Multi-threaded Workloads

Figures 16 and 17 present the performance results for the multi-threaded applications for the LRU and Hawkeye baselines, respectively. The results for canneal, facesim, vips, and 316.applu correspond to the configuration with 8 MB LLC and per-core 512 KB L2 cache. The results for TPC-E also correspond to a configuration with per-core L2 cache capacity being half of per-core LLC capacity, but since it is run on a 128-core system, it has a 32 MB shared LLC and 128 KB per-core L2 cache. The results in both the figures are normalized to the LRU baseline. For the LRU baseline (Figure 16), canneal, facesim, and vips show very little performance-sensitivity to inclusion victims. For 316.applu and TPC-E, the ZIV design with the `LikelyDead` property performs better than the non-inclusive LLC design.

With the Hawkeye baseline (Figure 17), both the ZIV LLC designs (the rightmost two bars for each application) perform close to the non-inclusive LLC design. We observe that for facesim and vips the QBS and SHARP proposals perform worse than the inclusive LLC design. These two applications show small performance-sensitivity toward inclusion victims (shown by the small performance difference between baseline inclusive and non-inclusive LLCs) because the inclusion victims are mostly harmless from performance viewpoint for these two applications. However, these applications have a lot of LLC reuses in the baseline inclusive LLC. QBS and SHARP sacrifice a lot of these LLC hits to save

inclusion victims leading to large performance losses. The TPC-E results validate the scalability of our proposal to larger core-counts. We also observe that the additional LLC latency incurred for accessing the shared relocated blocks in the ZIV LLC designs (see Section III-C1) has very little performance impact as nullifying this additional latency affects performance by a negligible amount.

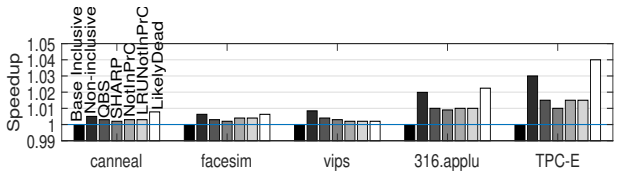

Fig. 16. Performance of multi-threaded workloads with LRU as the baseline LLC policy.

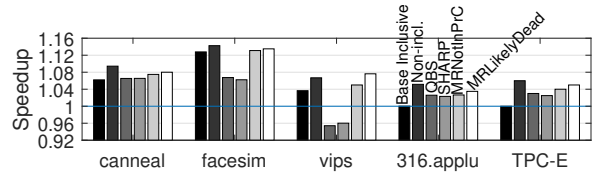

Fig. 17. Performance of multi-threaded workloads with Hawkeye as the baseline LLC policy.

C. Relocation Statistics and Energy Expense

Figure 18 shows the cumulative distribution of the interval lengths (in CPU cycles) between consecutive relocations in an LLC bank observed across all LLC banks for all multi-programmed and multi-threaded workloads in the configuration with 512 KB per-core L2 cache and 8 MB LLC (for TPC-E, 128 KB per-core L2 cache and 32 MB LLC). The log of relocation interval length is shown on the x-axis. The distributions are shown for three ZIV LLC designs (from left to right) exercising the (i) `LikelyDead` property (operating with LRU baseline), (ii) `MRNotInPrC` property (operating with Hawkeye baseline), (iii) `MRLikelyDead` property (operating with Hawkeye baseline). We make two important observations from these data. First, for all three designs, the fraction of relocation intervals that are less than five cycles is extremely small. Recall that the combinational logic that computes the decoded $nextRS$ has a latency of three cycles. Therefore, for a vast majority of cases, the decoded $nextRS$ would be ready long before the need for the next relocation arises. Second, for the `MRNotInPrC` and `MRLikelyDead` designs, the knee of the distribution shifts significantly to the left compared to the `LikelyDead` design indicating that the Hawkeye baseline needs more frequent relocations and has, in general, a bigger fraction of smaller relocation intervals. This is expected given the much higher volume of inclusion victims in the Hawkeye baseline.

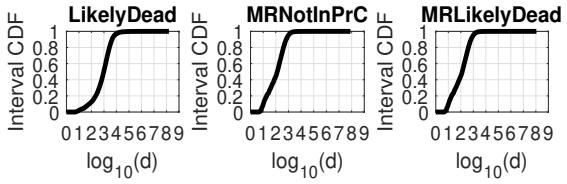

Fig. 18. Cumulative distribution of relocation intervals (in CPU cycles) in the configuration with 512 KB L2 cache.

Figure 19 shows the energy expense of relocation as an addition to energy per instruction (EPI) for the multi-programmed workloads. The primary energy expense of the ZIV LLC arises from block relocation and the additional dynamic and leakage energy expended in the widened sparse directory. Block relocation involves reading a block out of the LLC and writing it to the relocation set. Using CACTI we estimate the energy expense assuming 22 nm technology nodes. Figure 19 shows that as the L2 cache capacity increases, the EPI contribution also increases due to an increased number of relocations needed to keep the cache hierarchy free of inclusion victims. Across all configurations, the contribution to EPI is at most 12 pJ for the multi-programmed workloads; for the multi-threaded applications, it is at most 8 pJ (details not shown). These increments in EPI are small fractions of typical EPI numbers that range from several tens of pico-Joules to nano-Joules as seen in different types of instructions across different processors [35], [48], [53]. Using CACTI and Micron DDR3 power calculator [36], we further compare this EPI addition against the average EPI saved in the L2 caches, LLC, and DRAM as a result of fewer L2 cache and LLC misses and reduced execution time. For the configuration with 512 KB L2 cache, the ZIV LLC exercising the `MRLikelyDead` property saves average EPI of 0.5 pJ in the L2 cache and the LLC and 14.6 pJ in DRAM for the multi-programmed workloads. Since this configuration expends about 12 pJ of additional EPI in the ZIV LLC design, our proposal enjoys an overall advantage of at least about 3 pJ EPI over the inclusive baseline. We expect additional EPI saving in other system components due to reduced execution time.

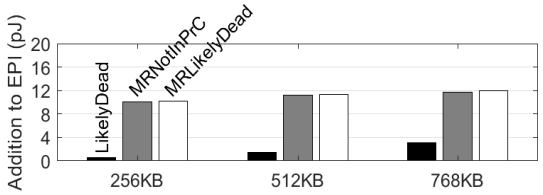

Fig. 19. Contribution to average EPI for the multi-programmed workloads.

VI. Summary

We have presented the Zero Inclusion Victim (ZIV) LLC design that enables for the first time an inclusive LLC with a guarantee of freedom from inclusion victims. The crux of the design involves invoking a block relocation protocol whenever the baseline LLC policy selects a victim that could generate inclusion victims. We have presented a set of different block relocation policies with varying complexity and performance goals. The policies differ primarily in terms of the properties satisfied by the target relocation set where an offending LLC victim is relocated to. The best ZIV LLC design allows integration of reasonably large private caches (e.g., half the LLC capacity) while delivering performance close to a non-inclusive LLC for different LLC replacement policies as shown by our evaluation on multi-programmed and multi-threaded workloads.

Our evaluation has shown that as the baseline LLC policy gets better, it becomes more challenging to design good relocation set properties that can help the inclusive LLC perform close to the non-inclusive LLC. Further, as the private cache capacity increases, this gets even more challenging. While the current study has laid the foundation for designing ZIV LLCs, an important future work would be to explore the design of better relocation set properties. One can compute the optimal relocation victim from among the LLC blocks that are not resident in the private caches for a given private cache capacity. Future work needs to explore how close one can get to this oracle-assisted optimal selection. Another important future work would be to analyze the security guarantees of the ZIV LLC design, given that such a design has enabled the much-needed isolation between the private caches and the inclusive LLC evictions.

Acknowledgment

The author would like to thank the anonymous reviewers for all the feedback.

REFERENCES

[1] L. Backes and D. A. Jimenez. The Impact of Cache Inclusion Policies on Cache Management Techniques. In *MEMSYS*, 2019.

[2] J-L. Baer and W-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *ISCA*, 1988.

[3] L. A. Barroso, K. Gharachorloo, and A. Nowatzyk. Method and System for Exclusive Two-level Caching in a Chip-multiprocessor. *US Patent 6725334B2*, granted April 2004.

[4] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. In *IBM Systems Journal*, **5**(2): 78–101, 1966.

[5] C. Bienia, et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[6] M. Chaudhuri. Zero Directory Eviction Victim: Unbounded Coherence Directory and Core Cache Isolation. In *HPCA*, 2021.

[7] M. Chaudhuri, et al. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *PACT*, 2012.

[8] P. Conway, et al. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. In *IEEE Micro*, **30**(2):16–29, March/April 2010.

[9] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *ISCA*, 2011.

[10] D. Gruss, et al. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security*, 2015.

[11] D. Gullasch, at al. Cache Games — Bringing Access-based Cache Attacks on AES to Practice. In *Security & Privacy*, 2011.

[12] V. Gupta, et al. Seclusive Cache Hierarchy for Mitigating Cross-Core Cache and Coherence Directory Attacks. In *DATE*, 2021.

[13] S. Gupta, H. Gao, H. Zhou. Adaptive Cache Bypassing for Inclusive Last Level Caches. In *IPDPS*, 2013.

[14] A. Gupta, et al. Reducing Memory and Traffic Requirements for Scalable Directory-based Cache Coherence Schemes. In *ICPP*, 1990.

[15] HP Labs. CACTI: An Integrated Cache and Memory Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model. Available at http://www.hpl.hp.com/research/cacti/.

[16] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing — and its Application to AES. In *Security & Privacy*, 2015.

[17] R. Iyer. On Modeling and Analyzing Cache Hierarchies using CASPER. In *MASCOTS*, 2003.

[18] A. Jain and C. Lin. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *ISCA*, 2016.

[19] A. Jaleel, et al. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *ISCA*, 2010.

[20] A. Jaleel, et al. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *MICRO*, 2010.

[21] A. Jaleel, et al. High Performing Cache Hierarchies for Server Workloads: Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches. In *HPCA*, 2015.

[22] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *ISCA*, 1994.

[23] M. Kayaalp, et al. A High-resolution Side-channel Attack on Last-level Cache. In *DAC*, 2016.

[24] M. Kayaalp, et al. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In *DAC*, 2017.

[25] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security*, 2012.

[26] V. Kiriansky, et al. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO*, 2018.

[27] B. O'Krafka and A. Newton. An Empirical Evaluation of Two Memory-efficient Directory Methods. In *ISCA*, 1990.

[28] K. M. Lepak and R. D. Isaac. Mostly Exclusive Shared Cache Management Policies. *US Patent 7640399B1*, granted December 2009.

[29] M. Lipp, et al. ARMageddon: Last-level Cache Attacks on Mobile Devices. In *USENIX Security*, 2016.

[30] F. Liu, et al. Last-level Cache Side-Channel Attacks are Practical. In *Security & Privacy*, 2015.

[31] F. Liu, et al. CATalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In *HPCA*, 2016.

[32] C. Luk, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[33] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. In *CACM*, **55**(7):78–89, July 2012.

[34] R. L. Mattson, et al. Evaluation Techniques for Storage Hierarchies. In *IBM Systems Journal*, **9**(2): 78–117, 1970.

[35] M. McKeown, et al. Power and Energy Characterization of an Open Source 25-Core Manycore Processor. In *HPCA*, 2018.

[36] Micron Technology, Inc.. Calculating Memory System Power for DDR3. *Micron Technical Note TN-41-01*, 2007.

[37] V. Nagarajan, et al. "A Primer on Memory Consistency and Cache Coherence". *Synthesis Lectures in Computer Architecture*, Morgan & Claypool Publishers, February 2020.

[38] Y. Oren, et al. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, 2015.

[39] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference on Topics in Cryptology*, 2006.

[40] D. Page. Partitioned Cache Architecture as a Side-channel Defence Mechanism. In *IACR Eprint archive*, 2005.

[41] B. Panda. Fooling the Sense of Cross-Core Last-Level Cache Eviction Based Attacker by Prefetching Common Sense. In *PACT*, 2019.

[42] M. K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *MICRO*, 2018.

[43] M. K. Qureshi. New Attacks and Defense for Encrypted-address Cache. In *ISCA*, 2019.

[44] T. Ristenpart, et al. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *CCS*, 2009.

[45] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. In *IEEE Computer Architecture Letters*, **10**(1): 16–19, January-June 2011.

[46] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *MICRO*, 2010.

[47] A. Seznec. A Case for Two-Way Skewed-Associative Caches. In *ISCA*, 1993.

[48] Y. S. Shao and D. M. Brooks. Energy Characterization and Instruction-level Energy Model of Intel's Xeon Phi Processor. In *ISLPED*, 2013.

[49] T. Sherwood, et al. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*, 2002.

[50] J. Sim, et al. FLEXclusion: Balancing Cache Capacity and On-chip Bandwidth via Flexible Exclusion. In *ISCA*, 2012.

[51] W. Song and P. Liu. Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC. In *USENIX RAID*, 2019.

[52] R. Ubal, et al. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *PACT*, 2012.

[53] E. Vasilakis. "An Instruction Level Energy Characterization of ARM Processors". *Technical Report FORTH-ICS*, TR-450, March 2015.

[54] P. Vila, B. Kopf, and J. F. Morales. Theory and Practice of Finding Eviction Sets. In *Security & Privacy*, 2019.

[55] Y. Wang, et al. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *DAC*, 2016.

[56] Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *ISCA*, 2007.

[57] Z. Wang and R. B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *MICRO*, 2008.

[58] J. Wang, et al. Reducing Data Movement and Energy in Multilevel Cache Hierarchies without Losing Performance: Can you have it all? In *PACT*, 2019.

[59] C-J. Wu, et al. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *MICRO*, 2011.

[60] M. Yan, et al. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *ISCA*, 2017.

[61] M. Yan, et al. Attack Directories, Not Caches: Side Channel Attacks in a Non-inclusive World. In *Security & Privacy*, 2019.

[62] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security*, 2014.

[63] M. Zahran, K. Albayraktaroglu, and M. Franklin. Non-Inclusion Property in Multi-level Caches Revisited. In *IJCA*, **14**(2), June 2007.

[64] L. Zhao, et al. NCID: A Non-inclusive Cache, Inclusive Directory Architecture for Flexible and Efficient Cache Hierarchies. In *CF*, 2010.

[65] Y. Zhang, et al. Cross-tenant Side-channel Attacks in PaaS Clouds. In *CCS*, 2014.

[66] Y. Zheng, B. T. Davis, and M. Jordan. Performance Evaluation of Exclusive Cache Hierarchies. In *ISPASS*, 2004.

[67] Z. Zhou, M. K. Reiter, and Y. Zhang. A Software Approach to Defeating Side Channels in Last-level Caches. In *CCS*, 2016.