Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches

Mainak Chaudhuri
Indian Institute of Technology
Kanpur 208016, INDIA
mainakc@iitk.ac.in

Jayesh Gaur
Intel Architecture Group
Bangalore 560103, INDIA
jayesh.gaur@intel.com

Nithiyanandan Bashyam
Intel Architecture Group
Bangalore 560103, INDIA
nithiyanandan.bashyam@intel.com

Sreenivas Subramoney
Intel Architecture Group
Bangalore 560103, INDIA
sreenivas.subramoney@intel.com

Joseph Nuzman
Intel Architecture Group
Haifa 31015, ISRAEL
joseph.nuzman@intel.com

ABSTRACT

The replacement policies for the last-level caches (LLCs) are usually designed based on the access information available locally at the LLC. These policies are inherently sub-optimal due to lack of information about the activities in the inner-levels of the hierarchy. This paper introduces cache hierarchy-aware replacement (CHAR) algorithms for inclusive LLCs (or L3 caches) and applies the same algorithms to implement efficient bypass techniques for exclusive LLCs in a three-level hierarchy. In a hierarchy with an inclusive LLC, these algorithms mine the L2 cache eviction stream and decide if a block evicted from the L2 cache should be made a victim candidate in the LLC based on the access pattern of the evicted block. Ours is the first proposal that explores the possibility of using a subset of L2 cache eviction hints to improve the replacement algorithms of an inclusive LLC. The CHAR algorithm classifies the blocks residing in the L2 cache based on their reuse patterns and dynamically estimates the reuse probability of each class of blocks to generate selective replacement hints to the LLC. Compared to the static re-reference interval prediction (SRRIP) policy, our proposal offers an average reduction of 10.9% in LLC misses and an average improvement of 3.8% in instructions retired per cycle (IPC) for twelve single-threaded applications. The corresponding reduction in LLC misses for one hundred 4-way multi-programmed workloads is 6.8% leading to an average improvement of 3.9% in throughput. Finally, our proposal achieves an 11.1% reduction in LLC misses and a 4.2% reduction in parallel execution cycles for six 8-way threaded shared memory applications compared to the SRRIP policy.

In a cache hierarchy with an exclusive LLC, our CHAR proposal offers an effective algorithm for selecting the subset of blocks (clean or dirty) evicted from the L2 cache that need not be written to the LLC and can be bypassed. Compared to the TC-AGE policy (analogue of SRRIP for exclusive LLC), our best exclusive LLC proposal improves average throughput by 3.2% while saving an average of 66.6% of data transactions from the L2 cache to the on-die interconnect for one hundred 4-way multi-programmed workloads. Compared to an inclusive LLC design with an identical hierarchy, this corresponds to an average throughput improvement of 8.2% with only 17% more data write transactions originating from the L2 cache.

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Design Styles

General Terms

Algorithms, design, measurement, performance

Keywords

Last-level caches, replacement policy, bypass algorithm

1. INTRODUCTION

The replacement policy for a particular level of a cache hierarchy is usually designed based on the access information (frequency, recency, etc.) available only at that level of the hierarchy. Such a level-centric design methodology for the last-level cache (LLC) fails to incorporate two important pieces of information. First, the reuses taking place in the inner-levels of a hierarchy are not propagated to the LLC. Past research showed that propagating even a small fraction of these reuses to the LLC can significantly increase the traffic in the on-die interconnect [7]. Second, the clean evictions from the inner-levels are usually not propagated to the LLC in an inclusive or a non-inclusive/non-exclusive hierarchy.

This paper, for the first time, studies the possibility of using an appropriately chosen subset of L2 cache evictions as hints for improving the replacement algorithms of an inclusive LLC (or L3 cache) in a three-level hierarchy. The central idea is that when the L2 cache residency of a block comes to an end, one can estimate its future liveness based on its reuse pattern observed during its residency in the L2 cache. Particularly, if we can deduce that the next reuse distance of such a block is significantly beyond the LLC reach, we can notify the LLC that this block should be marked a potential victim candidate in the LLC. An early eviction of such a block can help retain more blocks in the LLC with relatively shorter reuse distances. In a hierarchy with an exclusive LLC, this liveness information can be used to decide the subset of the L2 cache evictions that need not be allocated in the LLC.

To estimate the merit of making an inclusive LLC aware of the L2 cache evictions, we conduct an oracle-assisted experiment where the LLC runs the two-bit SRRIP policy [8] (this is our baseline in this paper). The two-bit SRRIP policy fills a block into the LLC with a re-reference prediction value (RRPV) of two and promotes it to RRPV of zero on a hit. A block with RRPV three (i.e., large re-reference distance) is chosen as the victim in a set. If none exists, the RRPVs of all

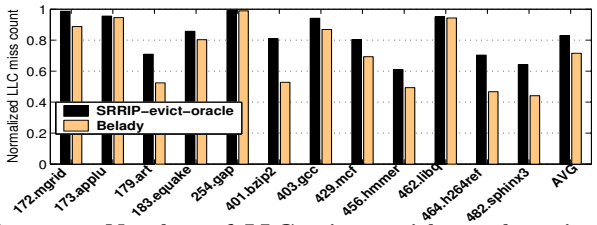

Figure 1: Number of LLC misses with oracle-assisted LLC replacement policies normalized to baseline SRRIP in an inclusive LLC.

the blocks in the set are increased in steps of one until a block with RRPV three is found. Ties are broken by victimizing the block with the least physical way id.[1] In our oracle-assisted experiment, on an L2 cache eviction of a data block[2], a next forward use distance oracle determines the relative order between the next forward use distances of the evicted block and the current SRRIP victim in the LLC set where the L2 cache victim belongs to. If the next forward use distance of the L2 cache victim is bigger, it is marked a potential victim by changing its RRPV to three in the LLC.

The left bar in each group of Figure 1 shows the number of LLC misses of this oracle-assisted policy normalized to the baseline SRRIP. The bar on the right in each group shows the number of LLC misses in Belady's optimal algorithm [1, 20] normalized to the baseline. These experiments are carried out on an offline cache hierarchy simulator that takes as input the entire L2 cache access trace of twelve single-threaded applications drawn from SPEC 2000 and SPEC 2006 suites. The L2 cache access trace of each application is collected for a representative set of one billion dynamic instructions chosen using the SimPoint toolset [22]. The cache hierarchy consists of 32 KB 8-way L1 instruction and data caches, a 256 KB 8-way L2 cache, and a 2 MB 16-way inclusive LLC. The L1 and L2 caches implement LRU replacement policy. Overall, Belady's optimal policy saves 28.5% of the LLC misses (refer to the AVG group), while the oracle-assisted SRRIP policy with L2 cache eviction hints can save about 17% of the baseline LLC misses. As a result, this policy can bridge close to two-third of the gap between Belady's optimal and the baseline SRRIP. Of course, this potential can be realized only if we can accurately learn which L2 cache evictions should be used to update the RRPV rank in the target LLC set and the solution to this problem forms the crux of our proposal.

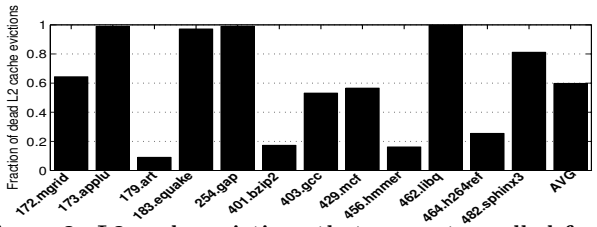

Figure 2: L2 cache evictions that are not recalled from the LLC in baseline SRRIP policy.

To further justify the data in Figure 1, we show in Figure 2 the percentage of blocks evicted from the L2 cache that are never recalled by the core from the time they are evicted from the L2 cache until they are evicted from the LLC. These data show that almost 60% of the blocks evicted from the L2 cache turn out to be dead in the LLC (refer to the AVG group). Such a block could be marked a potential LLC replacement candidate at the time it is evicted from the L2 cache provided we can separate it from the live blocks. Early replacement of such dead blocks can improve performance if the application has a good number of live blocks that can now stay longer in the LLC and enjoy additional reuses.

[1] SRRIP is known to outperform NRU and LRU [8].

[2] We do not apply our policy proposal to instruction blocks.

The fact that 60% of the L2 cache evictions are dead corresponds well with the already known fact that the blocks brought into the LLC have low use counts [21]. For the data in Figure 2, the average use count per LLC block is about 1.67 (reciprocal of dead percentage). In summary, the L2 cache eviction stream is rich in information regarding liveness of the cache blocks. Accurate separation of the live blocks from the dead ones in the L2 cache eviction stream can bridge a significant portion of the gap between the baseline and the optimal replacement policy for the LLC.

In Section 2, we present our cache hierarchy-aware replacement (CHAR) algorithms for inclusive LLCs. Figure 3 shows a high-level implementation of our CHAR algorithm for inclusive LLCs. The dead hint detector hardware is part of the L2 cache controller. It consumes the L2 cache eviction stream and identifies the eviction addresses that should be sent to the LLC as dead hints. As usual, it sends all dirty evictions to the LLC, some of which may be marked as dead hints. We note that the general framework of CHAR algorithms is not tied to any specific LLC replacement policy.

Section 2 also discusses how our CHAR proposal seamlessly applies to exclusive LLCs as well. In such designs, we use the dead hints to decide which blocks evicted from the L2 cache can be bypassed and need not be allocated in the exclusive LLC (blocks are allocated and written to an exclusive LLC when they are evicted from the L2 cache). This leads to bandwidth saving in the on-die interconnect and effective capacity allocation in the LLC. Further, we show that simple variants of our CHAR proposal can be used to dynamically decide if a block should be cached in exclusive mode or non-inclusive/non-exclusive mode in the LLC. The former mode optimizes the effective on-die cache capacity, while the latter trades cache capacity for on-die interconnect bandwidth by tolerating controlled amount of duplication of contents between the LLC and the L2 cache.

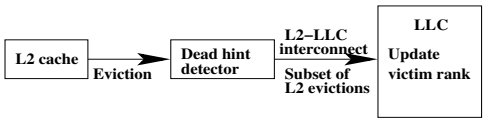

Figure 3: Implementation of CHAR algorithm.

Our execution-driven simulation methodology and results for single-threaded, multi-programmed, and shared memory parallel workloads are discussed in Sections 3 and 4. For inclusive LLCs, CHAR achieves 10.9%, 6.8%, and 11.1% reductions in LLC misses respectively for these workload classes compared to the baseline SRRIP policy. This leads to an average IPC improvement of 3.8% for the single-threaded applications, an average throughput improvement of 3.9% for the multi-programmed workloads, and an average reduction of 4.2% in parallel execution cycles for the shared memory applications. For exclusive LLCs, the best CHAR proposal achieves an average throughput improvement of 3.2% compared to the two-bit TC-AGE baseline (analogue of SRRIP for exclusive LLC) [5] while saving 66.6% data write transactions from the L2 cache to the on-die interconnect for the multi-programmed workloads.

1.1 Related Work

A large body of research work exists in the domain of replacement policies for inclusive LLCs. However, almost all of these consider the LLC in isolation and design the algorithms based on information available locally at the LLC. Only a few published studies explore hierarchy-aware (sometimes called global) replacement policies for inclusive LLCs. The first such study explored a number of global replacement policies where different types of access hints from the inner-level are sent to the LLC [26]. It showed that the advantage of such global schemes is limited to specific scenarios. A subsequent study further analyzes the limited utility of access hint-based global replacement schemes using a reuse distance argument [4].

A recent work shows that inner-level access hints can improve performance of an inclusive LLC significantly for a selected set of multi-programmed workloads if the on-die interconnect bandwidth is not a constraint [7]. This study also proposes two techniques, namely, early core invalidation (ECI) and query-based selection (QBS) to infer the temporal locality of inner-level accesses without sending access hints to the LLC. At the time of an LLC eviction, ECI invalidates the next LLC victim block from the L1 and L2 caches so that the LLC can observe any short-term temporal locality of this block before it is evicted from the LLC. QBS probes the L1 and L2 caches when an eviction decision is taken in the LLC to infer the usefulness of the current LLC victim and accordingly modifies the selection of LLC victims. In all the inclusive LLC configurations used in this paper, we keep the inclusion overhead low by maintaining an 8:1 capacity ratio between the LLC and the L2 cache [7], thereby eliminating most of the negative effects of inclusion victims. A recent work [27] explores an orthogonal dimension of the problem by proposing global cache management schemes to decide which level of the hierarchy in a two-level cache an incoming block should be placed in. Our proposal significantly differs from all these existing proposals. Our policy learns to identify a subset of blocks evicted from the L2 cache that can be made potential victim candidates in the LLC.

Our proposal shares some similarities with the dead block predictors. The existing dead block predictors predict the last access or the last burst of accesses to a block [6, 12, 13, 14, 16, 18]. These predictors usually require partial or full program counters (PC) to construct the necessary correlations with liveness of cache blocks. A recent work constructs a PC-less dead-on-fill predictor for use in cache bypassing by dynamically segmenting the LLC between referenced and not referenced blocks [11]. PC-less light-weight dead block predictors exploiting the fill order of LLC blocks have also been proposed [2]. Our basic CHAR proposal infers the death of an LLC block at the time it is evicted from the L2 cache and does not rely on program counter information. We briefly explore how to extend this basic CHAR design to take into account PC-based correlations.

Our proposal relies on estimation of reuse distance patterns in the L2 cache eviction stream. L2 cache eviction patterns have been used to arrive at bypass decisions and assign insertion ages to the non-bypassed blocks in the context of exclusive L3 caches [5]. LLC insertion and replacement policies based on static and dynamic re-reference interval prediction (SRRIP and DRRIP) have been explored [8]. A recent proposal improves the re-reference interval prediction of RRIP by exploiting LLC fill PC signatures (SHiP-PC), memory region signatures (SHiP-Mem), and instruction trace signatures (SHiP-ISeq) [24]. Another recent work shows how to extend RRIP to manage LLCs shared between CPU workloads and GPGPU workloads in a CPU-GPU heterogeneous environment [17]. Further, the PACMan family of policies is shown to outperform the RRIP policy in the presence of hardware prefetching by judicious design of RRPV insertion and update algorithms for prefetch fills and prefetch hits [25]. Prediction of reuse distances or next-use distances by correlating with program counters has also been studied [10, 19].

A recent proposal explores a set dueling-based solution to deliver performance close to an exclusive LLC while saving on-die interconnect bandwidth by dynamically switching the entire LLC between non-inclusive/non-exclusive and exclusive modes based on the outcome of the duel [23]. We show that our best CHAR proposal for exclusive LLC can dynamically decide the caching modes of different classes of blocks in the LLC at a fine grain.

2. CHAR ALGORITHMS

This section details our proposal on cache hierarchy-aware replacement (CHAR). The high-level flow diagram of CHAR is shown in Figure 3. Section 2.1 presents the design of the dead hint detector that identifies dead blocks in the L2 cache eviction stream in the context of an inclusive LLC. Section 2.2 explores the relationship of this design with PC-based dead block predictors. In Section 2.3, we discuss how the same dead hint detector design can be used in the context of an exclusive LLC.

2.1 Dead Hint Detector

The dead hint detector relies on the reuse behavior experienced by the blocks residing in the L2 cache to decide whether a block is likely to be recalled in future from the LLC after it is evicted from the L2 cache. To better learn a summary of this reuse behavior, we classify an L2 cache block (data block only) into one of five categories at the time of its eviction from the L2 cache. Such a classification is expected to separate the blocks with different liveness/death patterns.

2.1.1 Classification of L2 Cache Blocks

The classification of the L2 cache blocks is based on an approximate estimation of reuse distances inferred from their L2 cache usage patterns. This classification is invoked when a block is evicted from the L2 cache. The following four attributes $(A_0, \ldots, A_3)$ are used to classify an L2 cache block.

- (A_0) The type of the request that filled the block in the L2 cache (prefetch or demand).
- (A_1) The request that filled the block in the L2 cache was a hit or a miss in the LLC.
- (A_2) Number of demand uses (including the fill if it was a demand fill) enjoyed by the block during its residency in the L2 cache.
- (A_3) The L2 state of the block when it is evicted from the L2 cache.

Table 1 summarizes the class definitions, while Figure 4 shows how a block can transition from one class membership to another during its residency in the L2 cache. For example, a block belongs to C_0 if a) it is filled into the L2 cache by a prefetch request that misses in the LLC, b) it fails to experience any demand hit during its residency in the L2 cache, and c) it is evicted from the L2 cache in a clean state (E or S in a MESI protocol). On the other hand, a block is categorized as a C_2 block if a) it is filled into the L2 cache by a prefetch or a demand request that misses in the LLC, b) it enjoys exactly one demand use during its residency in the L2 cache (if it was filled by a demand request, it does not experience any demand hit), and c) it is evicted from the L2 cache in modified state.

Table 1: L2 cache block classification

Class	A_0	A_1	A_2	A_3
C_0	Prefetch	Miss	0	E/S
C_1	X	Miss	1	E/S
C_2	X	Miss	1	M
C_3	X	Miss	≥ 2	X
C_4	X	Hit	X	X

The C_0 class is relevant only if a hardware prefetcher is turned on and this class separates the potential premature or incorrect prefetches from rest of the blocks.[3] The remaining four classes separate the L2 cache blocks into different reuse distance bins. Beyond the L1 cache hits, a C_1 block is expected to have most of its natural reuse distances bigger than the reach of the L2 cache, since it fails to experience any demand hits while residing in the L2 cache. The C_2 blocks are

[3] In our simulation model, the prefetched blocks are brought into the LLC and the L2 cache, but not into the L1 cache.

similar to the C_1 blocks, except that the former class is modified and the latter is clean. We find that in several applications, separation of C_1 blocks from the C_2 blocks improves the identification of dead blocks in the L2 cache eviction stream. The C_3 blocks are likely to have a reuse cluster falling within the reach of the L2 cache. Finally, the C_4 blocks are likely to have a reuse cluster within the LLC reach.

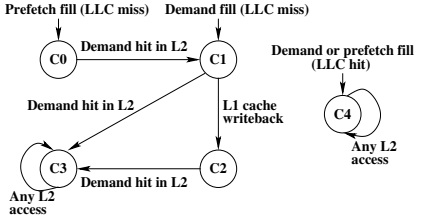

Figure 4: State transitions among the classes of the L2 cache blocks.

Our classification of the L2 cache blocks is inspired by the classification based on trip count and use count of cache blocks presented in an earlier study in the context of a cache hierarchy with an exclusive LLC [5]. According to the terminology used in that study, the set $C_0 \cup C_1 \cup C_2 \cup C_3$ contains the zero trip count blocks i.e., the blocks that are filled in the L2 cache for the first time during their residency in the hierarchy. The C_4 blocks have positive trip counts because they are recalled at least once from the LLC. As a result, C_4 is a subset of $C_0 \cup C_1 \cup C_2 \cup C_3$.

Table 2 shows how our five classes of cache blocks can be encoded in the L2 cache with just two extra state bits (S_1, S_0) per L2 cache block (as opposed to three bits per L2 cache block in [5]). Figure 4, through the class transitions, unambiguously defines the state transitions of these two bits on hits and writebacks to L2 cache blocks.[4]

Table 2: Class encoding in the L2 cache

State M	State S_1	State S_0	Class
X	0	0	C_0
0	0	1	C_1
1	0	1	C_2
X	1	0	C_3
X	1	1	C_4

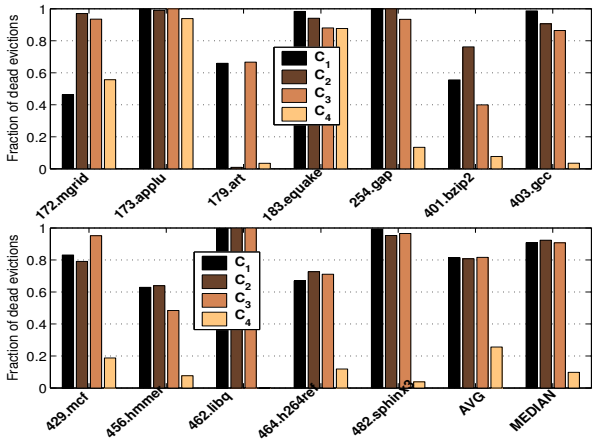

Figure 5: Fraction of dead evictions in each L2 cache block class.

Figure 5 quantifies the benefit of implementing the aforementioned classification of the L2 cache blocks. It shows the fraction of dead L2 cache evictions in each class for the non-prefetched baseline configuration on the single-threaded applications (32 KB 8-way LRU L1 caches, 256 KB 8-way LRU L2 cache, 2 MB 16-way SRRIP LLC). Naturally, the C_0 class is non-existent in non-prefetched executions. Overall, the C_1, C_2, and C_3 classes have very high fractions of dead blocks, while the C_4 blocks are mostly live (see the average and median groups of bars). By looking at the average or median data, one may be tempted to merge the C_1, C_2, and C_3 classes, given that they behave similarly. However, since C_4 is a subset of $C_1 \cup C_2 \cup C_3$, it is necessary to use a finer-grain partitioning of the set $C_1 \cup C_2 \cup C_3$ to identify the blocks that eventually get promoted to C_4. Also, there are applications where the C_1, C_2, and C_3 classes behave very differently. For example, in 172.mgrid, the likelihood of finding a dead block in C_1 evictions is below 0.5. The same is applicable to the C_3 evictions of 401.bzip2. On the other hand, the C_2 evictions coming out of the L2 cache in 179.art are primarily live. As a result, it is necessary to dynamically learn the reuse probability (same as live fraction) of each class of blocks evicted from the L2 cache and based on these probabilities we need to classify a block evicted from the L2 cache as dead or live. To save hardware resources necessary for this learning, our dead hint detector statically classifies all C_4 evictions as live because dead blocks are usually a minority in the C_4 class. The learning algorithm for the remaining four classes $(C_0, \ldots, C_3)$ is presented next.

2.1.2 *Learning Dead Evictions*

The goal of the learning algorithm is to estimate the reuse probabilities of the L2 cache block classes. If the estimated probability of a class falls below an appropriate threshold, the blocks belonging to that class are identified as dead when they are evicted from the L2 cache and this hint is propagated to the LLC. To effectively learn these probabilities, each class C_k $(k \in \{0, 1, 2, 3\})$ maintains two saturating counters, namely, the eviction counter (E_k) and the live counter (L_k). An eviction counter E_4 is also maintained for class C_4. All these counters reside in the L2 cache controller. Further, sixteen LLC sample sets per 1024 LLC sets are dedicated for learning E_k and L_k. These LLC sample sets always execute the baseline SRRIP replacement policy. The L2 cache controller is made aware of the hash function for determining if an L2 cache fill or eviction address maps to an LLC sample set. E_k keeps track of the number of L2 cache evictions belonging to class C_k and mapping to the LLC sample sets. L_k keeps track of which of these E_k evictions are recalled from the LLC (these are the live evictions). The total number of L2 cache evictions mapping to the LLC sample sets is maintained in a saturating counter N_E residing in the L2 cache controller.

Tables 3 and 4 show the actions of the L2 cache controller on an eviction and a fill, respectively. These actions are in addition to the usual ones such as sending a writeback to the LLC on evicting a dirty block from the L2 cache, etc.. The L2 cache eviction actions depend on whether the evicted address maps to an LLC sample set and the class of the evicted block. The L2 cache fill actions depend on whether the filled address maps to an LLC sample set and two attributes of the fill e.g., fill type (demand/prefetch) and hit/miss in the LLC. When a block mapping to one of the LLC sample sets is evicted from the L2 cache, its class id (k) and address (A) are sent to the LLC. The LLC stores the class id with the block. This storage is needed only for the sample sets in the LLC. The LLC, on a hit to a block in one of the sample sets, sends two pieces of additional information along with the fill message: the last stored class id (k) of the block and one bit signifying a hit in the LLC. This value of k is used to update L_k, as shown in the first and third rows of Table 4.

From Tables 3 and 4, we conclude that the estimated reuse probability of class C_k is L_k/E_k, which is the collective hit

[4] In our model, a block is filled into the L2 cache in either S or E state and it can transition to M state only if the L1 data cache writes the block back to the L2 cache.

Table 3: L2 cache block eviction flow (address A)

Class	Maps to LLC sample set	Does not map to LLC sample set
$C_k, k \in \{0,1,2,3\}$	E_k++, N_E++, send A and k to LLC	Invoke dead block detection algorithm
$C_k, k = 4$	E_4++, N_E++, send A and k to LLC	—

Table 4: L2 cache block fill flow (last class id k, if hit in LLC sample set)

Fill attribute	Maps to LLC sample set	Does not map to LLC sample set
Demand hit in LLC	L_k++ if $k \neq 4$, fill in L2 cache as C_4	Fill in L2 cache as C_4
Demand miss in LLC	Fill in L2 cache as C_1	Fill in L2 cache as C_1
Prefetch hit in LLC	L_k++ if $k \notin \{0,4\}$, fill in L2 cache as C_4	Fill in L2 cache as C_4
Prefetch miss in LLC	Fill in L2 cache as C_0	Fill in L2 cache as C_0

rate of the blocks belonging to class C_k, as learned from the LLC sample sets executing the baseline SRRIP policy. Notice, however, that a prefetch fill in the L2 cache does not update the live counter of C_0, even if the filled block hits in an LLC sample set (third row of Table 4). This is because such a block was filled into the L2 cache last time by a prefetch request and got evicted as a C_0 block without experiencing a demand hit. If this block is again prefetched into the L2 cache, we speculate that this is likely to be a premature or incorrect prefetch and do not update the live counter of C_0.

Finally, we present our dead block detection algorithm. This algorithm dynamically estimates an appropriate threshold t such that if $L_k/E_k < t$ or equivalently $L_k < tE_k$, we identify a block belonging to class C_k as dead when it is evicted from the L2 cache. One reasonable choice for t at any point in time during execution would be the baseline hit rate of the LLC. Such a dynamic choice would guarantee that if any class of blocks has hit rate lower than the current baseline LLC hit rate, the blocks in that class would be marked dead as and when they are evicted from the L2 cache. This algorithm would evict the less useful blocks from the LLC early and create more space in the LLC for the blocks that are contributing more heavily toward LLC hit rate. A highly accurate online estimate of the baseline LLC hit rate is E_4/N_E because E_4 approximates the number of LLC hits to the sampled sets and N_E approximates the total number of LLC accesses to the sampled sets.

To simplify the hardware, we approximate the E_4/N_E ratio such that t turns out to be a reciprocal of power of two. Therefore, we can synthesize $L_k < tE_k$ using a shifter and a comparator. The approximation is done by dividing the possible values of E_4 into four ranges, namely, $[0, N_E/8]$, $(N_E/8, N_E/4]$, $(N_E/4, N_E/2]$, and $(N_E/2, N_E]$. On each L2 cache eviction, we first determine the range the current value of E_4 falls in. If it falls in the lowermost range i.e., $[0, N_E/8]$, we approximate it to half of the upper bound of the range. If it falls in any of the remaining ranges, we approximate it to the lower bound of the range, which is also half of the upper bound of the range. Finally, we determine t as $\tilde{E_4}/N_E$ where $\tilde{E_4}$ is the approximate value of E_4. This leads to the following definition of t.

$$t = \begin{cases} 1/16 & \text{if } E_4 \leq N_E/8 \\ 1/8 & \text{if } N_E/8 < E_4 \leq N_E/4 \\ 1/4 & \text{if } N_E/4 < E_4 \leq N_E/2 \\ 1/2 & \text{if } E_4 > N_E/2 \end{cases} \quad (1)$$

The computation of t may require further tuning. For certain workload classes, $t = 1/2$ may turn out to be very aggressive. In such situations, we recommend merging the last two ranges in Equation (1) i.e., $t = 1/4$ if $E_4 > N_E/4$. Similarly, the last three ranges can be merged, if necessary. Also, a carefully chosen static value of t can offer reasonably good performance. A static value of 1/8 achieves excellent performance for our selection of workloads. In this study, we present results assuming Equation (1) for computing t dynamically.

2.1.3 Few Implementation Details

The L2 cache, on evicting a data block, first queries the L1 data cache. If the query hits in the L1 data cache, the L1 data cache retains the block (our L2 cache is non-inclusive/non-exclusive of the L1 caches). Only those L2 cache data evictions that do not hit in the L1 data cache are passed on to the dead hint detector. The dead hint detector receives the address and the class of each such L2 cache eviction. It decides if the block is dead in the LLC by determining the current value of t and applying the dead hint detection algorithm (only for blocks not mapping to LLC sample sets). If the block is identified as dead, its address is sent to the LLC in a special dead hint message. The LLC, on receiving a dead hint message, sets the RRPV of the block to three. It also clears the bit position corresponding to the evicting core in the coherence bitvector of the block, since neither L1 cache nor L2 cache of the evicting core has the block. This saves a future back-invalidation message.[5] If the LLC receives an eviction message from the L2 cache for a block mapping to one of its sample sets, it stores the class id of the block found in the eviction message.

The combined hardware overhead of our CHAR proposal involves two extra state bits per L2 cache block, ten counters in the L2 cache controller ($L_0, \ldots, L_3$, E_0, …, E_4, and N_E), negligible logic overhead to implement the additional actions in the L2 cache controller on eviction and fill (Tables 3 and 4), the logic of the dead hint detector in the L2 cache controller, three extra bits per LLC block for the sample sets to store class id, and handling of dead hints to non-sampled sets and eviction messages to sampled sets in the LLC. Note that our proposal does not involve any dynamic dueling between CHAR and baseline SRRIP policies, even though we use a small number of sampled sets in the LLC to dynamically learn the reuse behavior of the SRRIP policy. The L_k, E_k, and N_E counters are halved periodically whenever the total number of L2 cache evictions mapping to the LLC sampled sets reaches a value of 127. We also experimented with halving intervals of 255, 1023, and 2047, but did not see any significant swing in performance. With a halving interval of 127, we need eight bits for each of the ten counters and seven bits for the counter that keeps track of the halving interval.

The CHAR algorithms seamlessly apply to multi-core scenarios without any change, since each core would have its own CHAR hardware attached to its private L2 cache controller. The sample sets in the shared LLC would be shared by all the threads.

For shared memory programs, we do not apply CHAR to the shared blocks, since the current proposal does not have the global view necessary to detect the death of shared blocks. We rely on the inclusive LLC for detecting the read-shared and read/write-shared blocks. When a request from a core

[5] We simulate a bitvector-based directory coherence protocol. The L1 and L2 caches are private to each core and the LLC is shared. Each LLC tag is extended to maintain the coherence states and sharing bitvector in the inclusive LLC design.

hits in the LLC and the requested block is currently shared by another core (as indicated by the sharing bitvector directory), the LLC marks this block shared using an additional state bit per LLC block. Once marked, this bit remains set until the block is evicted from the LLC. Dead hints received for the blocks with this bit set are ignored by the LLC. Also, as soon as a block mapping to the LLC sample sets is identified as shared, it is upgraded to class C_4 in the LLC so that it does not update the L_k counters of any of the sharing cores. This policy will be referred to as CHAR-S.

2.1.4 Controlling Dead Hint Rate

Our dead hint generation algorithm tries to victimize the blocks belonging to the classes that have collective hit rates lower than the baseline LLC hit rate. However, there are situations where the overall baseline hit rate is reasonably high, but some of the classes are undergoing a phase transition and have low hit rates. The blocks belonging to these classes will be reused in near-future if we can retain them. CHAR cannot infer this by looking at the current hit rates of these classes and can hurt performance by sending premature dead hints to the LLC. The SPEC 2000 application 172.mgrid exhibits a few such phases. To address this problem, we incorporate a dead hint rate divider D with each core's L2 cache controller that sends out every D^{th} dead hint to the LLC. The value of D is always a power of two, and ranges between one and 256, inclusive. In all simulations, we initialize D to one. In the following, we discuss how D is adjusted dynamically.

We ear-mark sixteen LLC sample sets (different from the sixteen baseline samples) per 1024 LLC sets to monitor the relative number of hits experienced by CHAR compared to the baseline samples. This is done using a saturating counter residing in the L2 cache controller. The counter is initialized to the midpoint M of the range of the counter e.g., M is 2^{n-1} for an n-bit counter. The counter is incremented whenever the L2 cache receives a fill that hits in one of the CHAR sample sets of the LLC. The counter is decremented whenever the L2 cache receives a fill that hits in one of the baseline sample sets of the LLC.

After every eight halving intervals, we check the status of this counter. If the counter value is at least $M+T_{gb}$, we halve D to double the dead hint rate because CHAR is performing better than the baseline. Note, however, that the minimum value of D is one. If the counter value is less than or equal to $M-T'_{gb}$, we quadruple D provided the LLC hit rate is at least 3/8 i.e., E_4 is at least $N_E/4+N_E/8$; otherwise, D is left unchanged. The rationale is that we do not decrease the dead hint rate if the baseline LLC hit rate is anyway small. Note that this particular hit rate threshold (i.e., 3/8) may require tuning depending on the workload set under consideration. The guard-band thresholds, T_{gb} and T'_{gb}, are used to avoid potentially spurious adjustments in D and are set to 8 and 32 in this study. The dead hint rate division algorithm is conservative in the sense that it increases D much faster (though with a bigger guard-band) than it decreases D so that the performance penalty is low in the phases where CHAR becomes too aggressive. In multi-core configurations, the CHAR sample sets in the shared LLC are shared by all the threads and each private L2 cache controller maintains its own dead hint rate divider. To implement this algorithm, we use a 16-bit saturating counter for relative hit count monitoring (i.e., M is set to 2^{15}) and a 9-bit dead hint rate divider.

2.2 Code Space Correlation and CHAR-PC

Past research on dead block predictors has established a strong correlation between program counters (PCs) of the load/store instructions and the death patterns of the data blocks, as already discussed in Section 1.1. Figure 6 explores the relationship between the program counters of the instructions that bring the blocks of different classes into the L2 cache in the baseline configuration (32 KB 8-way LRU L1

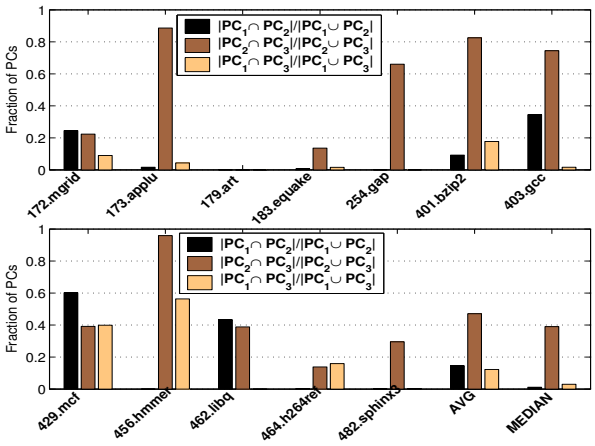

Figure 6: Relationship between L2 cache block classes and code space.

caches, 256 KB 8-way LRU L2 cache, and 2 MB 16-way SRRIP LLC) for the single-threaded programs with the hardware prefetcher disabled. We focus only on the classes C_1, C_2, and C_3 that can source dead hints in a non-prefetched execution. Let PC_k be the set of program counters that cause the blocks, which eventually get classified as C_k blocks, to be filled into the L2 cache. Figure 6 shows $|PC_k \cap PC_{k'}|$ as a fraction of $|PC_k \cup PC_{k'}|$ for all ordered pairs (k, k') with $k, k' \in \{1, 2, 3\}$. It is very encouraging to find that the load/store instructions that bring C_1 blocks into the L2 cache overlap very little with those that bring C_2 and C_3 blocks (see the average and median groups). The implication is that our PC-less classification of L2 cache blocks can effectively capture differing code space signatures for the pairs (C_1, C_2) and (C_1, C_3). However, the classes C_2 and C_3 share a sizable proportion of program counters. Further examination of this behavior revealed that the blocks that are written to and do not belong to C_4 are consumed in either near-future (C_3) or far-future (C_2). The dirty C_4 blocks have reuses in intermediate-future such that the reuse distance is within the LLC reach. Therefore, we conclude that our classification can successfully partition the dirty cache blocks based on their next use distance, which the code signature fails to do.

While our classification correlates well with code space signatures, further improvements may be possible if we can split a class of blocks based on L2 cache fill PCs. For example, suppose the set PC_1 has two program counters, say, PC_{11} and PC_{12}. The blocks filled into the L2 cache by the instruction at PC_{11} may behave differently from those filled by the instruction at PC_{12}. Clustering these both types of blocks into a single class would lower the overall prediction accuracy. To resolve this issue, we propose CHAR-PC, a PC-based extension to CHAR. CHAR-PC maintains, for each class C_k with $k \in \{1, 2, 3\}$, an eight-entry fully associative table T_k for storing the lower 14 bits of the PCs (after removing the lowest two bits) of the instructions that bring the C_k blocks into the L2 cache. Each entry of T_k has a valid bit, a 14-bit fill PC signature, and a three-bit saturating counter.

When a block is filled into the L2 cache, the 14-bit fill PC signature is stored with the L2 cache block. When a block mapping to an LLC sample set is evicted from the L2 cache, its class C_k is determined and T_k is looked up with its PC signature. If the signature is not found in T_k, an entry is allocated by invoking the not-recently-used (NRU) replacement policy and the saturating counter for that entry is initialized to zero. If the entry is found in T_k, the saturating counter for that entry is decremented by one. The fill PC signature and the class id of the evicted block are sent to the LLC. The LLC stores these along with the blocks mapping to the sample sets. When a block mapping to an LLC sample set is filled into the L2 cache as a result of an LLC hit, the

last class id (k) and the last fill PC signature of the block are supplied by the LLC. At this point, T_k is looked up with the last fill PC signature and if the entry is found, its saturating counter is incremented by one.

Finally, the dead block prediction takes place when a block not mapping to an LLC sample set and belonging to C_k with $k \in \{1, 2, 3\}$ is evicted from the L2 cache. T_k is looked up with the fill PC signature of the block. If the entry is found and the saturating counter has a value zero, the block is predicted dead and a dead hint is propagated to the LLC. The signature length, the saturating counter size, and the prediction threshold of zero have been borrowed from the SHiP-PC proposal [24]. In Section 4, we show that CHAR-PC, which exploits the cross product of our classification and L2 cache fill PC signature, improves performance beyond SHiP-PC.

2.3 Application to Exclusive LLC

A block is allocated in an exclusive LLC when it is evicted from the L2 cache and it is de-allocated from the LLC on a subsequent hit or replacement. Since every L2 cache eviction (clean or dirty) must be sent to the LLC for allocation, an exclusive LLC design consumes much bigger on-die interconnect bandwidth compared to an inclusive LLC design. However, the dead hints of the CHAR algorithm can be used to identify the blocks that can be dropped by the L2 cache and need not be sent to the LLC for allocation. This is known as selective cache bypassing. In the following, we discuss the working of the CHAR algorithm in an exclusive LLC. Our baseline exclusive LLC allocates all L2 cache evictions and decides the insertion age of a block based on the two-bit TC-AGE policy [5]. This policy is the analogue of SRRIP for exclusive LLCs. It inserts all C_0, C_1, C_2, and C_3 blocks at age two and the C_4 blocks at age zero. Note that a block is in C_4 class if and only if it has experienced at least one LLC hit. The LLC replacement policy is same as the inclusive SRRIP replacement policy (i.e., it victimizes a block with age three).[6]

In CHAR algorithm for exclusive LLC, every L2 cache eviction address is first sent to the LLC to update the coherence directory (the data is not sent at this point). The dead hint (one bit) for the block is sent to the LLC along with this message. If the block is not marked dead, the LLC requests the block from the L2 cache for allocation. If the block is marked dead, but there is an invalid way available in the target LLC set, a request for the block is sent by the LLC to the L2 cache. However, in this case, the block is filled at age three in the LLC. If the block is marked dead and there is no invalid way available in the target LLC set, the LLC sends a bypass command to the L2 cache. If the L2 cache receives a bypass command for a dirty block, it sends the block directly to the memory controller over the interconnect. On the other hand, if it receives a bypass command for a clean block, it drops the block and eliminates the data transaction. In summary, the CHAR algorithm allocates all live blocks in the LLC with an insertion age dictated by the TC-AGE policy. It also allocates a dead block at the highest possible age provided an invalid way is available in the target LLC set. All other dead blocks are either dropped (if clean) or sent to the memory controller (if dirty) by the L2 cache.

Since the CHAR algorithm classifies the C_4 blocks as live, these blocks are always allocated in the LLC. A block moves to the C_4 class as soon as it experiences a hit in the LLC and it remains in the C_4 class during its residency in the cache hierarchy. As a result, a C_4 block may get repeatedly allocated in and de-allocated from the LLC on its every trip between the L2 cache and the LLC. This leads to unnecessary wastage of on-die interconnect bandwidth.

[6] We use the term "age" instead of "RRPV" in this discussion to conform to the terminology used in the prior work on exclusive LLC management [5]. Age can be considered synonymous to RRPV in this discussion.

We address this inefficiency by observing that a block allocated in the LLC should not be de-allocated on a hit because it is a live block and will have to be allocated again when it is evicted from the L2 cache. In other words, live blocks should be cached in non-inclusive/non-exclusive mode to save on-die interconnect bandwidth. However, if a C_4 block is evicted from the LLC by the time it is evicted from the L2 cache, it must be re-allocated in the LLC at age zero so that the live blocks are retained in the cache hierarchy. Also, if such a block is evicted in the dirty state from the L2 cache, it must be sent to the LLC to update the LLC copy. In all other cases, the L2 cache can drop a C_4 block. Finally, the age of a C_4 block in the LLC is changed to zero when it is evicted from the L2 cache so that the block, being live, gets the highest level of protection in the LLC.

We explore three variations of this basic policy that eliminates the unnecessary data transactions involving the C_4 blocks. While these policies do not de-allocate a block from the LLC on a hit, they update the age of the block in the LLC differently at the time of the hit. The first policy, CHAR-C4, leaves the age of the block unchanged at the time of hit. The second policy, CHAR-C4-MAX, updates the age of the block to the maximum possible i.e., three. The third policy, CHAR-C4-MIN, updates the age of the block to the minimum possible i.e., zero. All three policies continue to fill cache blocks from memory directly into the L2 cache and a block is considered for allocation in the LLC only when it is evicted from the L2 cache and the dead hint detector identifies it as live. Among these three policies, CHAR-C4-MIN offers the best protection for C_4 blocks and is expected to have the highest bypass rate (percentage of blocks dropped by the L2 cache) because it reduces the chance of a C_4 block getting evicted from the LLC before it is evicted from the L2 cache. On the other hand, CHAR-C4-MAX is expected to have a bypass rate that is higher than CHAR but lower than CHAR-C4 and CHAR-C4-MIN. Since all these three policies reduce the effective on-die cache capacity, their relative performance depends on the sensitivity of the workloads toward cache capacity and on-die interconnect bandwidth.

3. EVALUATION METHODOLOGY

We carry out our execution-driven simulations on a MIPS ISA simulator. We model an out-of-order issue core running at 4 GHz with 4-wide front-end and 8-wide commit. The core model has an 18-stage pipeline with 128-entry re-order buffer, 160-entry integer and floating-point register files, and a hybrid branch predictor similar to the Alpha 21264 tournament predictor. The front-end of the core has a 256-set 4-way branch target buffer, a 32-entry return address stack, and a 32-entry branch stack allowing 32 in-flight branch instructions. The issue unit of the core has a 32-entry integer queue, a 32-entry floating-point queue, and a 64-entry load/store queue. The execution unit of each core is equipped with six integer ALUs, three FPUs, and two load/store units.

Each core has private L1 and L2 caches. The L1 instruction and data caches are 32 KB 8-way with LRU replacement policy. In the inclusive LLC model, the capacity ratio between the LLC and the L2 cache is maintained at 8:1 in all our configurations to keep the inclusion overhead low [7]. Our single-threaded applications are simulated on a 256 KB 8-way L2 cache with LRU replacement policy and a 2 MB 16-way LLC. The 4-way multi-programmed workloads are simulated on a 4-core model with each core having a 256 KB 8-way private L2 cache. The LLC is 8 MB 16-way and shared among all the cores. The 8-way threaded shared memory programs are simulated on an 8-core model with each core having a 128 KB 8-way private L2 cache. The LLC is 8 MB 16-way and shared among all the cores. In all the multi-core configurations, the LLC banks and the cores sit on a bidirectional ring that has a single-cycle hop time. Each hop has a core and an LLC bank. The L2 cache of a core connects to the router in each ring

hop through a 32-entry outgoing message queue. The LLC in the single-threaded and multi-programmed models has banks of size 2 MB with access latency of eight cycles (tag+data). The model used to evaluate the shared memory programs has 1 MB LLC banks with seven-cycle access latency (tag+data). Each LLC bank can keep track of 16 outstanding misses. All the levels in the cache hierarchy have 64-byte block size.

Table 5: Storage overhead of CHAR

States	ST	Mprog	Shm
States in L2C	8192 bits	32768 bits	32768 bits
Counters in L2C	112 bits	448 bits	896 bits
LLC samples	1536 bits	6144 bits	6144 bits
TOTAL	1.20 KB	4.80 KB	4.86 KB

Table 5 summarizes the extra storage overhead of CHAR (as discussed in Sections 2.1.3 and 2.1.4) for the three cache configurations used to evaluate single-threaded (ST), multi-programmed (Mprog), and shared memory (Shm) workloads. It is clear that this overhead is a small percentage (less than 0.1%) of the total storage devoted to the cache hierarchy. The CHAR-S implementation would need one extra shared bit per LLC block in the Shm configuration. This would increase the overhead of the Shm configuration to 20.86 KB, which is still less than 0.3% of the LLC capacity.

We model an aggressive memory system with four single-channel memory controllers clocked at 2 GHz with a round robin mapping of consecutive cache blocks on the controllers. Each controller implements the FR-FCFS scheduling policy and connects to a DIMM (64-bit interface) built out of 8-way banked DDR3-1600 DRAM chips. The 800 MHz DRAM part has burst length of eight and 9-9-9-27 (tCAS-tRCD-tRP-tRAS) access cycle parameters.

We model per-core multi-stream stride prefetchers that keep track of sixteen simultaneous streams and prefetch into the LLC and the L2 cache. The prefetcher attached to the L2 cache injects a prefetch request on an L2 cache miss as well as on a demand hit to a prefetched L2 cache block.

The exclusive LLC model uses the same configuration as the inclusive LLC model. Since there is no inclusion overhead in exclusive LLC models, the per-core private L2 cache capacity is increased to 512 KB (with a corresponding increase in access latency) to improve baseline performance.

Table 6: Baseline inclusive LLC MPKI

mgrid	applu	art	equake	gap	bzip2
2.92	4.62	6.51	13.93	3.60	1.18
gcc	mcf	hmmer	libq	h264ref	sphinx3
5.35	28.84	0.34	11.38	0.31	7.43

We select twelve single-threaded applications (shown in Figure 1) from SPEC 2000 and SPEC 2006 suites. The selected twelve applications represent a fair distribution of LLC miss savings achievable by Belady's optimal algorithm. The savings range from 1.1% in gap to 55.9% in sphinx3 (see Figure 1). The applications also represent a wide range of LLC misses per kilo instructions (MPKI) for the baseline SRRIP policy, as shown in Table 6. We simulate one billion dynamic instructions chosen using the SimPoint toolset [22] from each application executed on the ref input set.

The single-threaded applications are mixed to prepare one hundred 4-way heterogeneous multi-programmed workloads. In each workload, each thread is executed for 500 million representative dynamic instructions. If a thread finishes executing its set of instructions early, it continues to run beyond that point so that the LLC contention can be modeled correctly. However, performance is reported based on the first 500 million instructions retired by each thread.

We pick six shared memory kernels and applications for preliminary evaluation in this paper. These are FFT (256K complex double points), Radix-sort (4M points, radix 32), and Ocean (514×514 grid) from SPLASH-2, Art (MinneSPEC input [15]) and Equake (MinneSPEC input, ARCHduration 0.5) from SPEC OMP, and FFTW (4096×16×16 complex double points) [3]. All the applications are executed in entirety on eight cores (one thread per core).

4. SIMULATION RESULTS

We present the results for the single-threaded applications on the inclusive LLC model first with the hardware prefetcher disabled to understand how each application performs in isolation. Next we discuss the multi-core results for heterogeneous workload mixes as well as shared memory applications with and without the hardware prefetcher enabled. We conclude this section with a discussion of the results for the exclusive LLC model with the hardware prefetcher enabled.

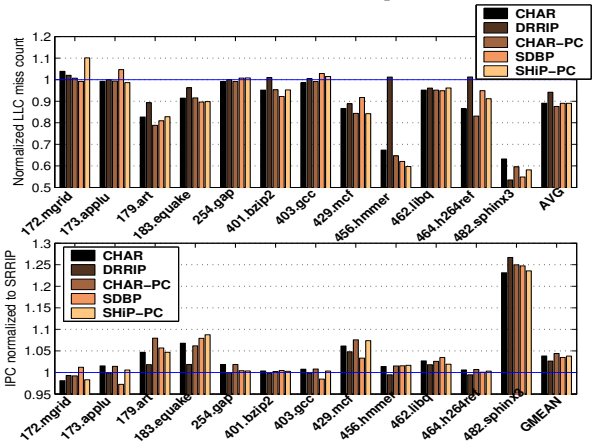

Figure 7: Upper panel: Number of LLC misses normalized to SRRIP. Lower panel: IPC normalized to SRRIP.

4.1 Inclusive LLC Model

In the following, we discuss the performance of CHAR on the inclusive LLC model.

4.1.1 Single-threaded Applications

Figure 7 compares the performance of CHAR, DRRIP [8], CHAR-PC, SDBP [12], and SHiP-PC [24]. The last three policies require the program counter of the load/store instructions. SDBP is the state-of-the-art dead block prediction technique that correlates death of a cache block with the last-touch PC of the block. We do not exercise the LLC bypass component of SDBP so that strict inclusion is maintained between the LLC and the L2 cache. SHiP-PC is the state-of-the-art PC-correlated LLC insertion policy that improves the re-reference interval prediction of SRRIP and DRRIP. The upper panel of Figure 7 compares the policies in terms of the number of LLC misses (lower is better) normalized to SRRIP, while the lower panel shows normalized IPC (higher is better). CHAR enjoys noticeable reductions in LLC misses in art, equake, bzip2, mcf, hmmer, libquantum, h264ref, and sphinx3. On average, CHAR reduces the number of LLC misses by 10.9%, while DRRIP exhibits a 5.8% reduction in LLC misses. The reduction in LLC misses achieved by CHAR-PC is 12.4%. Interestingly, CHAR achieves reductions in LLC misses similar to the SDBP and SHiP-PC policies without requiring any program counter information. This further confirms the finding of Section 2.2 that our L2 cache block classification captures the code space signatures quite well. Referring back to Figure 1, we find that CHAR and CHAR-PC still leave significant room for improvement.

As shown in the lower panel of Figure 7, the IPC gains correspond well with the LLC miss reductions. The IPC improvements achieved by CHAR and CHAR-PC are 3.8% and 4.4%, respectively. The corresponding improvements of DRRIP, SDBP, and SHiP-PC are 2.7%, 3.5%, and 3.8%, respec-

tively. Given the small benefit of CHAR-PC in comparison with CHAR and the added complexity of CHAR-PC, we do not pursue CHAR-PC any further.

Table 7: LLC hit distribution

Policy	C_1	C_2	C_3	C_4	Ins.	Total
CHAR	0.11	0.08	0.04	0.76	0.01	1.00
DRRIP	0.07	0.05	0.01	0.75	0.01	0.89
SRRIP	0.09	0.05	0.01	0.52	0.01	0.68
SRRIP-Or	0.26	0.13	0.21	0.78	0.04	1.42
Belady	0.34	0.17	0.14	0.85	0.02	1.52

To further understand the sources of LLC hits in CHAR and DRRIP, Table 7 shows the average distribution of LLC hits normalized to CHAR among the L2 cache block classes and instructions for the non-prefetched execution of the single-threaded applications. To collect this data, every block evicted from the L2 cache is classified as C_1, C_2, C_3, C_4, or instruction and this is also recorded in the LLC. A subsequent LLC hit to such a block belonging to a category increments the hit count of the corresponding category. We show the average of this hit distribution for CHAR, DRRIP, SRRIP, oracle-assisted SRRIP (SRRIP-Or), and Belady's optimal policy normalized to CHAR. The last two rows correspond to the two policies discussed in Figure 1. Compared to SRRIP, both DRRIP and CHAR significantly improve the volume of hits enjoyed by the C_4 blocks (recall that the C_4 blocks are predominantly live). DRRIP achieves this while sacrificing some of the hits in the C_1 class. The RRPV duel of DRRIP inserts *all* blocks in the LLC with RRPV of either two or three during an execution phase. Therefore, it is expected that during the phases dominated by a mix of C_1, C_2, and C_3 blocks, DRRIP would probably insert all blocks with RRPV three, even though inserting some of these with RRPV two could have improved performance. CHAR can enjoy such selectivity because it learns to send dead hints based on the reuse probabilities of different classes of blocks. As a result, it improves the volume of hits across all the four classes. Overall, DRRIP and SRRIP experience 11% and 32% less LLC hits compared to CHAR. The last two rows of Table 7 point out that further characterization of the C_1, C_2, and C_3 classes is necessary to exploit the remaining performance potential.

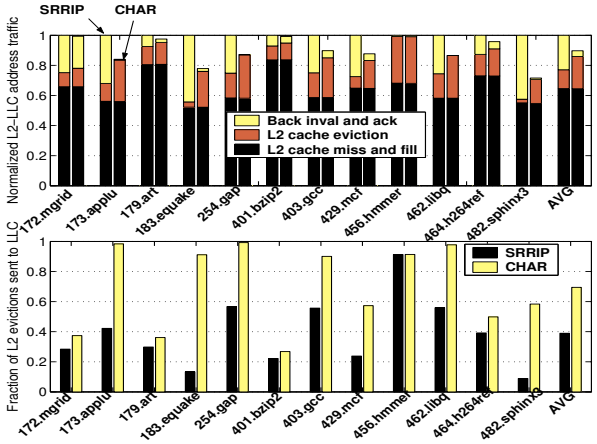

Figure 8: Upper panel: Address traffic in L2-LLC interconnect normalized to SRRIP. Lower panel: L2 cache eviction addresses sent to LLC.

Before closing the discussion on the single-threaded applications, we examine the L2-LLC address traffic in the presence of the dead hints generated by CHAR. We divide the traffic into three parts: a) L2 cache miss requests and the L2 cache fills, b) L2 cache evictions (in SRRIP, these are only the dirty block evictions, while in CHAR, these also include the eviction messages to the LLC sample sets and the dead hints to the non-sample sets), c) back-invalidations and their acknowledgments exchanged at the time of an LLC eviction.

The upper panel of Figure 8 shows the address traffic in the L2-LLC interconnect normalized to SRRIP. Both SRRIP and CHAR have the same volume of traffic due to L2 cache misses and L2 cache fills. Although CHAR increases the L2 cache eviction traffic (as expected), it dramatically reduces the traffic due to back-invalidations. The reduction in back-invalidations in CHAR is observed for two reasons. First, due to less number of LLC misses in CHAR, LLC evictions are less in number leading to a lower volume of back-invalidations. Second, a dead hint message clears the coherence bitvector position of the core sending the dead hint. When such a block is evicted from the LLC, no back-invalidation is generated. On average, CHAR enjoys 10% less address traffic on the L2-LLC interconnect compared to SRRIP. The lower panel of Figure 8 further quantifies the fraction of L2 cache eviction addresses sent to the LLC by SRRIP and CHAR. On average, SRRIP sends 38.9% of all L2 cache evictions to the LLC (these are dirty cache block evictions) and CHAR sends 69.5% of all L2 cache evictions to the LLC. It is important to note that a dead hint is a dataless message and inflates only the address traffic, unlike the dirty block evictions.

4.1.2 Multi-core Workloads

For heterogeneous workload mixes, we present three different metrics to summarize the performance and fairness of several policies. Normalized average IPC (or throughput) i.e., $\frac{\sum_i IPC_i^{NEW}}{\sum_i IPC_i^{SRRIP}}$ summarizes the improvement in average throughput of the mix when a policy "NEW" replaces the baseline SRRIP, where IPC_i is the IPC of thread i ($0 \leq i \leq 3$). We use $min_i \frac{IPC_i^{NEW}}{IPC_i^{SRRIP}}$ as a conservative fairness metric that captures the minimum throughput improvement enjoyed by any thread in a mix. A policy "NEW" is considered at least as fair as the baseline SRRIP policy if this metric is at least 1.0 meaning that no thread has suffered from a slowdown due to introduction of the new policy. Finally, we evaluate the normalized LLC miss count as $\frac{\sum_i M_i^{NEW}}{\sum_i M_i^{SRRIP}}$, where M_i is the number of LLC misses experienced by thread i during the execution of its first 500 million instructions. For shared memory applications, we execute each application completely and compare the policies in terms of the execution time of the parallel computation.

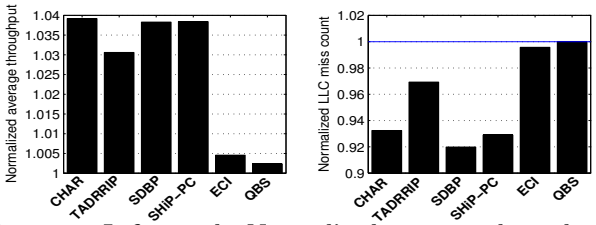

Figure 9: Left panel: Normalized average throughput comparison (higher is better). Right panel: Normalized average LLC miss count comparison (lower is better).

Figure 9 summarizes the normalized average throughput and LLC miss counts delivered by CHAR, TADRRIP [8], SDBP [12], SHiP-PC [24], ECI [7], and QBS [7]. These data represent the average across one hundred heterogeneous mixes with the hardware prefetcher disabled. CHAR improves average throughput by 3.9% and reduces the LLC misses by 6.8%. Surprisingly, CHAR performs slightly better than SDBP and SHiP-PC even though the latter policies use the PCs for the load/store instructions that access the LLC and fill into the LLC, respectively. The SDBP and SHiP-PC policies significantly reduce the LLC miss counts of several non-memory-intensive mixes and as a result, these two policies save more

LLC misses compared to CHAR, but they fail to convert these savings into throughput improvement. These results once again underscore the effectiveness of CHAR in capturing the code space signatures of LLC access patterns. CHAR does not have the complexity of carrying the PC information through the load-store pipeline, the L1 and L2 cache controllers, the on-die interconnect, and the LLC controller. Also, the SDBP and SHiP-PC policies need additional storage to maintain the PC signatures and the predictor tables.

Thread-aware DRRIP (TADRRIP) improves throughput by 3.1% and reduces the LLC miss count by 3.1%, on average. In Section 4.1.1, we have already explained how the fine-grain cache block classification of CHAR helps it enjoy more LLC hits compared to the DRRIP policy.

Finally, for completeness, we discuss the results of ECI and QBS (these policies were discussed in Section 1.1). For our workload mixes and simulated configurations, we do not expect ECI or QBS to deliver noticeable performance improvements. Only 2.5% of the back-invalidations sent at the time of LLC evictions in baseline SRRIP hit in the L1 or the L2 cache. As a result, at the time of an LLC eviction, the block has already been evicted from the core caches with a likelihood of 0.975. Hence, it is impossible for QBS to infer much about the temporal locality of the block by querying the core caches. Overall, QBS improves the average throughput by only 0.23% compared to the baseline SRRIP. For ECI, only 1.5% of early-invalidated blocks are recalled by the cores before they are evicted from the LLC leading to an average throughput improvement of 0.46% compared to the baseline SRRIP. The original study [7] reported that QBS performs better than ECI and we believe that we observe the opposite trend due to selection of different workload mixes.

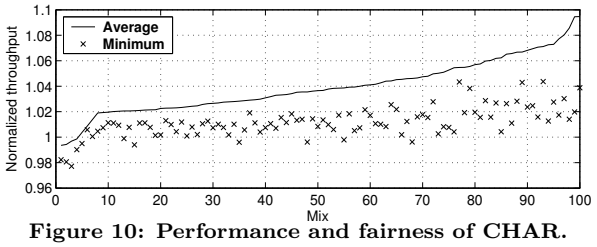

Figure 10: Performance and fairness of CHAR.

Figure 10 shows the details of the normalized average and minimum throughput delivered by CHAR for each of the mixes. The mixes are arranged in the increasing order of throughput improvement. The average throughput profile ranges from a loss of 1.0% to an improvement of 9.5%. The minimum throughput profile also looks very encouraging as only 11 of the 100 mixes have minimum improvement below 1.0. The minimum throughput improvement of any thread ranges from 0.98 to 1.04, averaging at 1.01. Overall, CHAR remains at least as fair as SRRIP while offering a throughput improvement of 3.9% and an LLC miss saving of 6.8%.

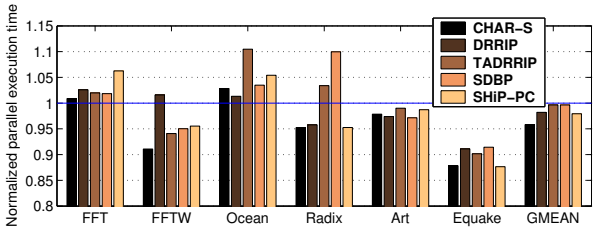

Figure 11: Comparison of parallel execution time.

Figure 11 presents the parallel execution time (lower is better) of CHAR-S, DRRIP, TADRRIP, SDBP, and SHiP-PC normalized to SRRIP for the shared memory applications. CHAR-S achieves noticeable savings in FFTW, Radix-sort, Art, and Equake. On average, it saves 4.2% execution time compared to the baseline. This performance gain comes from an 11.1% reduction in LLC misses. The reductions in execution time achieved by DRRIP, TADRRIP, SDBP, and SHiP-PC are 1.8%, 0.4%, 0.4%, and 2.1%, respectively.

4.1.3 Interaction with Hardware Prefetching

All the results presented up to this point pertain to configurations with the hardware prefetcher disabled. In this section, we evaluate the performance of our policy proposal in the presence of a multi-stream hardware prefetcher. This would also validate the efficiency of our algorithm in identifying and managing the blocks belonging to the C_0 class.

Table 8: Comparison between CHAR and PACMan-DYN-Global relative to SRRIP

Metric	CHAR	PACMan-DYN-Global
Throughput improvement	5.3%	2.8%
LLC demand misses saved	15.0%	11.5%
DRAM requests	8.8% less	4.8% more

Prefetching alone improves the average throughput of SRRIP by 20.6% for the hundred heterogeneous mixes. Table 8 further compares CHAR and PACMan-DYN-Global [25] for these mixes in terms of average throughput improvement, the number of LLC demand misses saved, and the total volume of DRAM requests relative to SRRIP with the hardware prefetcher enabled. PACMan is a family of RRPV insertion and RRPV update policies that was shown to outperform DRRIP and SDBP in the presence of hardware prefetching [25]. CHAR improves average throughput by 5.3% and significantly improves the effectiveness of the prefetcher by reducing the LLC demand miss count by 15.0% compared to the prefetched SRRIP. We find that compared to SRRIP, the CHAR algorithm evicts 11.3% less C_0 blocks from LLC. This is primarily due to dynamic learning of the usefulness of C_0 blocks in the LLC. The prefetched blocks that experience demand hits much later than they are prefetched into the hierarchy must be identified and retained in the LLC. These are a subset of the C_0 blocks. CHAR dynamically monitors the collective reuse probability of the C_0 blocks and learns to retain them if they are useful.

PACMan-DYN-Global delivers a throughput improvement of 2.8% compared to prefetched SRRIP. This policy dynamically selects one of the two algorithms, namely, PACMan-H and PACMan-HM. None of these policies upgrade the RRPV of a block on prefetch hits in the LLC. As a result, the blocks that can potentially enjoy multiple prefetch hits in the LLC followed by demand hits in the L2 cache get prematurely evicted from the LLC. Note that even though CHAR does not increment the live counter of C_0 class if a C_0 block experiences a prefetch hit in the LLC, it does increment the live counters of C_1, C_2, and C_3 classes if a block belonging to any of these three experiences a prefetch hit in the LLC. Further, CHAR always upgrades the RRPV of a block in the LLC on a prefetch hit following the SRRIP policy. A common pattern experienced by a live block in CHAR is the following. The block is prefetched into the L2 cache (filled as C_0 in the L2 cache). It enjoys demand hit(s) in the L2 cache and is evicted as a C_1, C_2, or C_3 block from the L2 cache. Later the block is again prefetched from the LLC and it enjoys further demand hits. While CHAR can retain such blocks in the LLC, PACMan-DYN-Global fails to do so because it does not upgrade the RRPVs on prefetch hits in the LLC.

Overall, we find that PACMan-DYN-Global is too aggressive in filtering prefetch-induced LLC pollution and in the process it often loses useful prefetched blocks early resulting in an eventual increase in the memory controller congestion because several prematurely evicted useful prefetches will have to be fetched/prefetched again. This fact is substantiated by the last row of Table 8. CHAR saves 8.8% DRAM requests

compared to SRRIP, while PACMan-DYN-Global increases the memory traffic by 4.8%. The increased pressure on the memory controllers leads to loss in performance for several mixes when running with PACMan-DYN-Global.

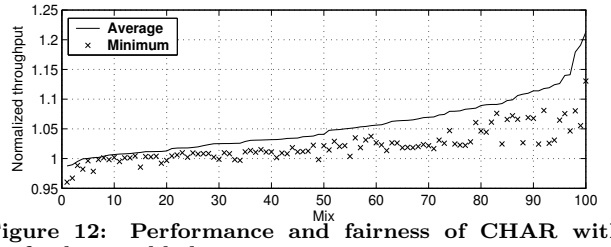

Figure 12: Performance and fairness of CHAR with prefetcher enabled.

Figure 12 shows the details of throughput improvement and fairness of CHAR for the heterogeneous mixes relative to SRRIP with the hardware prefetcher enabled. The mixes are arranged in the increasing order of throughput improvement. The throughput profile ranges from a loss of 1.2% to an improvement of 21.3%. The average throughput improvement achieved by CHAR is 5.3%. The minimum throughput improvement of any thread ranges from 0.96 to 1.13, averaging at 1.02. Only 16 mixes have minimum throughput improvement below 1.0. These data confirm that even in the presence of prefetching, CHAR continues to achieve significant performance improvement compared to SRRIP.

For the shared memory parallel programs, prefetching alone reduces the execution time of SRRIP by 15.6%. CHAR-S achieves a further 2.2% reduction in execution time relative to prefetched SRRIP. Thread-oblivious PACMan-DYN [25] and thread-aware PACMan-DYN-Global respectively achieve 1.1% and 0.4% reduction in parallel execution time.

4.2 Exclusive LLC Model

We summarize our results for the exclusive LLC model in Figure 13. All the simulations are done with the hardware prefetcher enabled. The leftmost panel shows the average throughput of one hundred 4-way multi-programmed mixes for a non-inclusive (NI) LLC model [7, 23] and TC-AGE, CHAR, CHAR-C4, CHAR-C4-MAX, and CHAR-C4-MIN policies implemented on the exclusive LLC model. The throughput results are normalized to an inclusive LLC model. The inclusive, non-inclusive, and exclusive LLC models have identically designed cache hierarchies (32 KB 8-way private L1 instruction and data caches, 512 KB 8-way private L2 cache, 8 MB 16-way shared LLC). The inclusive and the non-inclusive LLCs implement the two-bit SRRIP policy. The NI LLC model is identical to the inclusive LLC model, except that on an LLC eviction the NI model does not invalidate the copy of the block in the L2 and L1 caches. The performance gain of the NI model over the inclusive model arises from elimination of back-invalidations, while the performance gap between the NI model and the TC-AGE policy running on the exclusive LLC model stems from the added effective capacity of the exclusive LLC. Overall, CHAR-C4 delivers the best performance improving the average throughput by 8.2% over the inclusive model and 3.2% over the TC-AGE exclusive model. The CHAR, CHAR-C4-MAX, and CHAR-C4-MIN policies deliver performance close to CHAR-C4.

The middle panel of Figure 13 shows the bypass fractions for CHAR, CHAR-C4, CHAR-C4-MAX, and CHAR-C4-MIN. This fraction corresponds to the fraction of L2 cache evictions dropped by the L2 cache controller and not sent to the LLC or memory controllers in the exclusive LLC model. While CHAR bypasses 14.8% blocks on average, the other three policies that switch the C4 blocks to non-inclusive/non-exclusive mode enjoy more than 60% bypass rates. CHAR-C4 bypasses 66.6% of the L2 cache evictions.

The rightmost panel of Figure 13 quantifies the number of data write transactions from the L2 cache to the on-die

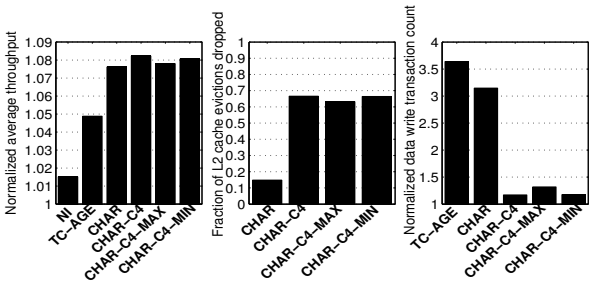

Figure 13: Throughput, bypass fraction, and L2 cache to interconnect data traffic in exclusive LLC model for multi-programmed workloads with prefetcher enabled.

interconnect in the exclusive LLC policies relative to the inclusive LLC model. For an exclusive LLC, these are the data transactions to LLC or memory controllers originating from L2 cache evictions that could not be bypassed. For an inclusive LLC, these are dirty writebacks from the L2 cache to the LLC. While TC-AGE generates 3.6 times data write transactions relative to the inclusive LLC model, the CHAR-C4 policy generates only 17% more writes to the LLC. As mentioned in Section 2.3, the TC-AGE policy does not exercise any bypass algorithm. Overall, CHAR-C4 is the best policy among the ones we have evaluated for an exclusive LLC. It improves the average throughput by 8.2% while generating only 17% more data writes to the LLC compared to an identically sized inclusive LLC hierarchy.

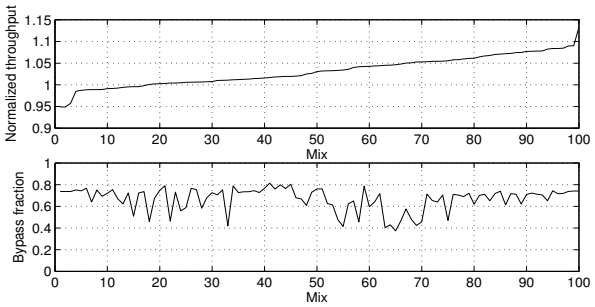

Figure 14: Throughput normalized to TC-AGE and bypass fraction of CHAR-C4 policy.

Figure 14 shows the details of throughput (upper panel) and bypass fraction (lower panel) of the CHAR-C4 policy for the heterogeneous mixes. The throughput is normalized to the TC-AGE policy. The mixes are arranged in the increasing order of normalized throughput. The throughput profile varies from a loss of 5.1% to a gain of 13.3%, while the bypass fraction is between 0.37 and 0.81. In other words, the CHAR-C4 policy can save 37% to 81% (average 66.6%) on-die data write traffic originating from the L2 cache evictions while delivering a throughput improvement of up to 13.3% (average 3.2%) relative to TC-AGE.

5. SUMMARY

This paper proposes hierarchy-aware replacement and allocation/bypass policies for LLCs in a three-level cache hierarchy. The proposal uses a carefully chosen subset of the L2 cache evictions as hints to improve the quality of replacement and allocation in inclusive and exclusive LLCs, respectively. When a data block is evicted from the L2 cache, it is passed on to a dead hint detector. The dead hint detector decides if the block should be marked a potential victim in an inclusive LLC, provided the block has already been evicted from the L1 data cache. A similar decision is used to identify L2 cache evictions that need not be allocated in an exclusive LLC and can be bypassed. Central to the dead hint detector logic is an approximate reuse distance-based classification of L2 cache blocks and estimation of reuse probability of each of

these classes. If the reuse probability of a class falls below a threshold, any block belonging to that class is marked a potential victim/bypass candidate for inclusive/exclusive LLC when the block is evicted from the L2 cache.

We evaluate our policy proposal on single-threaded, multi-programmed, and shared memory workloads. In an inclusive LLC, our cache hierarchy-aware algorithm offers an average throughput improvement of 5.3% for one hundred 4-way multi-programmed mixes compared to a baseline SRRIP policy with a well-tuned multi-stream hardware prefetcher enabled. Our best proposal for exclusive LLC improves the average throughput of one hundred 4-way multi-programmed mixes by 8.2% compared to an identical inclusive LLC hierarchy while bypassing 66.6% of the L2 cache evictions. As a result of this high bypass rate, this policy introduces only 17% more data write transactions into the on-die interconnect compared to an identical inclusive LLC hierarchy.

6. ACKNOWLEDGMENTS

This research effort is funded by Intel Corporation. The authors thank Praveen Vishakantaiah from Intel India and Koby Gottlieb from Intel Israel for their financial support and continued encouragement.

7. REFERENCES

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

[2] M. Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 401–412, December 2009.

[3] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. In *Proceedings of the IEEE*, **93**(2): 216–231, February 2005.

[4] R. V. Garde, S. Subramaniam, and G. H. Loh. Deconstructing the Inefficacy of Global Cache Replacement Policies. In *7th Annual Workshop on Duplicating, Deconstructing, and Debunking*, held in conjunction with the *35th International Symposium on Computer Architecture*, June 2008.

[5] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 81–92, June 2011.

[6] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 209–220, May 2002.

[7] A. Jaleel et al. Achieving Non-Inclusive Cache Performance with Inclusive Caches. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 151–162, December 2010.

[8] A. Jaleel et al. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 60–71, June 2010.

[9] A. Jaleel et al. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques*, pages 208–219, October 2008.

[10] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache Replacement Based on Reuse Distance Prediction. In *Proceedings of the 25th International Conference on Computer Design*, pages 245–250, October 2007.

[11] S. Khan, Z. Wang, and D. A. Jimènez. Decoupled Dynamic Cache Segmentation. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, pages 235–246, February 2012.

[12] S. Khan, Y. Tian, and D. A. Jimènez. Dead Block Replacement and Bypass with a Sampling Predictor. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 175–186, December 2010.

[13] S. Khan et al. Using Dead Blocks as a Virtual Victim Cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 489–500, September 2010.

[14] M. Kharbutli and Y. Solihin. Counter-based Cache Replacement and Bypassing Algorithms. In *IEEE Transactions on Computers*, **57**(4): 433–447, April 2008.

[15] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. In *Computer Architecture Letters*, **1**(1), January 2002.

[16] A-C. Lai, C. Fide, and B. Falsafi. Dead-block Prediction & Dead-block Correlating Prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, June/July 2001.

[17] J. Lee and H. Kim. TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, pages 91–102, February 2012.

[18] H. Liu et al. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 222–233, November 2008.

[19] R. Manikantan, K. Rajan, and R. Govindarajan. NUcache: An Efficient Multicore Cache Organization Based on Next-Use Distance. In *Proceedings of the 17th IEEE International Symposium on High-performance Computer Architecture*, pages 243–253, February 2011.

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

[21] M. K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[22] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[23] J. Sim et al. FLEXclusion: Balancing Cache Capacity and On-chip Bandwidth with Flexible Exclusion. In *Proceedings of the 39th IEEE/ACM International Symposium on Computer Architecture*, pages 321–332, June 2012.

[24] C-J. Wu et al. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 430–441, December 2011.

[25] C-J. Wu et al. PACMan: Prefetch-Aware Cache Management for High Performance Caching. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 442–453, December 2011.

[26] M. Zahran. Cache Replacement Policy Revisited. In *6th Annual Workshop on Duplicating, Deconstructing, and Debunking*, held in conjunction with the *34th International Symposium on Computer Architecture*, June 2007.

[27] M. Zahran and S. A. McKee. Global Management of Cache Hierarchies. In *Proceedings of the 7th Conference on Computing Frontiers*, pages 131–140, May 2010.