Characterizing Multi-threaded Applications for Designing Sharing-aware Last-level Cache Replacement Policies

Ragavendra Natarajan
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota 55455, USA
natar@cs.umn.edu

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

***Abstract*—Recent years have seen a large volume of proposals on managing the shared last-level cache (LLC) of chip-multiprocessors (CMPs). However, most of these proposals primarily focus on reducing the amount of destructive interference between competing independent threads of multi-programmed workloads. While very few of these studies evaluate the proposed policies on shared memory multi-threaded applications, they do not improve constructive cross-thread sharing of data in the LLC In this paper, we characterize a set of multi-threaded applications drawn from the PARSEC, SPEC OMP, and SPLASH-2 suites with the goal of introducing sharing-awareness in LLC replacement policies. We motivate our characterization study by quantifying the potential contributions of the shared and the private blocks toward the overall volume of the LLC hits in these applications and show that the shared blocks are more important than the private blocks. Next, we characterize the amount of sharing-awareness enjoyed by recent proposals compared to the optimal policy. We design and evaluate a generic oracle that can be used in conjunction with any existing policy to quantify the potential improvement that can come from introducing sharing-awareness. The oracle analysis shows that introducing sharing-awareness reduces the number of LLC misses incurred by the least-recently-used (LRU) policy by 6% and 10% on average for a 4MB and 8MB LLC respectively. A realistic implementation of this oracle requires the LLC controller to have the capability to accurately predict, at the time a block is filled into the LLC, whether the block will be shared during its residency in the LLC. We explore the feasibility of designing such a predictor based on the address of the fill and the program counter of the instruction that triggers the fill. Our sharing behavior predictability study of two history-based fill-time predictors that use block addresses and program counters concludes that achieving acceptable levels of accuracy with such predictors will require other architectural and/or high-level program semantic features that have strong correlations with active sharing phases of the LLC blocks.**

I. Introduction

Dynamic sharing of the last-level cache (LLC) among the processing cores of a chip-multiprocessor (CMP) has become a popular design choice in the industry due to better utilization of the cache space. Multi-threaded shared memory applications executing on such a CMP can enjoy fast communication between the threads mapped on different cores through the shared LLC. This inter-core reuse of shared data can be significantly improved by introducing sharing-awareness in replacement or insertion policies of the LLC. However, most replacement policy proposals for shared LLCs focus on reducing cross-thread destructive interference in multi-programmed workloads and pay no special attention to improving cross-thread constructive sharing in multi-threaded shared memory programs. Even though some of these proposals evaluate their policies on multi-threaded workloads, these policies treat private and shared data equally. In this paper, we squarely focus on the problem of introducing sharing-awareness in replacement policies for shared LLCs. With this goal in mind, we characterize a set of shared memory parallel applications drawn from the PARSEC [5], SPEC OMP [1], and SPLASH-2 [38] suites executing on an eight-core CMP with an inclusive three-level cache hierarchy. Our characterization is directed toward understanding the properties of an application that can positively or negatively influence the design of a sharing-aware LLC replacement policy.

Fundamentally, data sharing is the manifestation of the reuses that happen between the threads of a multi-threaded application. An LLC management policy that can accurately estimate reuse distances would naturally be sharing-aware. However, online accurate estimation of reuse distance is difficult and most existing policies try to indirectly estimate the next-use distances of cache blocks through easily implementable heuristics. In this paper, we explore the characteristics of multi-threaded applications that can be exploited to incorporate better treatment for shared cache blocks in these existing LLC management heuristics.

Our study begins with an analysis of the reuse behavior of the shared cache blocks that experience cross-thread reuses in multi-threaded applications. This analysis shows that the shared cache blocks contribute more LLC hits than the private cache blocks in these applications (Section IV). Next, we evaluate the degree of sharing-awareness enjoyed by recent LLC management proposals. This study reveals how these proposals compare against Belady's optimal replacement policy [3], [28] in terms of sharing-awareness (Section V). To improve the sharing-awareness of the existing proposals, we design a generic oracle that can be used in conjunction with any replacement policy. This oracle has future knowledge about the sharing pattern of the application and puts extra weight on shared data reuses (Section VI). The oracle also helps us identify a common design element that all good sharing-aware LLC replacement policies must possess. This design element is a predictor that predicts at the time a block is filled into the LLC whether the block will be shared by at least two cores during its residency in the LLC. We analyze how data is shared and utilized in the LLC by multi-threaded applications and highlight the implications of these characteristics on the design of such a predictor (Section VII). We explore two avenues for designing this predictor. First, we examine if the modes of sharing e.g., read-only and read-write have any correlation with the volume of the LLC hits experienced by the shared cache blocks. Second, we conduct a predictability study of two history-based fill-time predictors that make use of the address of the cache block being filled and the program counter of the instruction that caused the fill into the LLC. We conclude that other architectural features or high-level program semantics or a combination of both are required to achieve an acceptable level of prediction accuracy. These features should be such that they help the architecture identify the active sharing phases of a cache block so that the block can be treated differently by the LLC replacement policy during these phases.

This paper makes the following contributions.

- We show that the cross-thread reuses of the shared LLC blocks are more important than the intra-thread reuses of the private LLC blocks in multi-threaded applications.
- We show that the amount of data sharing in the LLC is greatly influenced by the LLC replacement policy.
- Our analysis based on a generic oracle shows that introducing sharing-awareness in the existing LLC replacement policies can significantly improve their performance.
- We analyze how the data in the LLC is utilized and shared in multi-threaded applications. We highlight the implications of these characteristics on the design of sharing-aware policies.

II. Related Work

Replacement and insertion policies for LLC blocks have been researched extensively. The insertion policies decide the age of a block at the time the block is filled into the LLC [13], [14], [32]. The dynamic insertion policy (DIP) adapts to the changing application behavior by deciding whether to insert a new block into the LLC at the least-recently-used (LRU) or the most-recently-used (MRU) position [32]. The thread-aware DIP extends this idea to control block insertion into a shared LLC for each thread of a multi-programmed workload [14]. These policies continue to use LRU as the replacement policy. The recently proposed re-reference interval prediction policy assigns an n-bit re-reference prediction value (RRPV) to each LLC block at the time of insertion into the LLC. The block with the largest i.e., $2^n - 1$ RRPV is predicted to have a large re-reference interval and is selected as the victim. On a hit, the RRPV of the block is updated to zero anticipating a short re-reference interval. The static re-reference interval prediction (SRRIP) policy assigns RRPV of $2^n - 2$ to all newly inserted blocks anticipating an intermediate re-reference interval. The dynamic re-reference interval prediction (DRRIP) policy adapts to the changing application behavior by dynamically choosing the RRPV of a new block from the set $\{2^n - 1, 2^n - 2\}$. Thread-aware DRRIP extends DRRIP to decide the insertion RRPV of the heterogeneous threads in a multi-programmed workload. None of these proposals evaluate the policies on shared memory multi-threaded workloads. A recent study shows that DRRIP is more effective than thread-aware DRRIP for shared memory multi-threaded workloads where the threads are more homogeneous due to the single-program-multiple-data (SPMD) nature of the workloads [7]. In this paper, we explore sharing-awareness of the two-bit (i.e., $n = 2$) SRRIP and DRRIP policies. In this study, we focus only on inclusive LLCs. We note that there have been studies that explore insertion policies for exclusive LLCs as well [7], [11].

Replacement policy proposals attempt at approximating Belady's optimal policy, which victimizes the block with the largest next-use distance within a set [3], [28]. This is the optimal dead block within a set. The LRU replacement policy speculates that the block with the largest next-use distance would be the one accessed least recently. Replacement policies proposing improvements on LRU attempt to identify more accurate dead block candidates by correlating block reuses, reuse distances and death of a block with program counters of the instructions that access the block [12], [17], [18], [19], [20], [22], [25], [39], or by incorporating techniques to gain some look-ahead into the cache access stream to estimate the next-use distances [10], [27], [34], or by exploiting other properties of access patterns that do not make use of program counters [7], [8]. However, very few of these proposals evaluate the policies on shared memory multi-threaded workloads [7], [8], [36] and they do not address the problem of improving cross-thread sharing in the LLC. In this paper, we explore the sharing-awareness of SHiP-PC, a recently proposed policy that identifies the probable dead blocks in a set by correlating the reuses of a block with the program counter of the instruction that fills the block into the LLC [39]. A recent work (CSHARP) proposes to offer extra protection to the dirty shared blocks to improve the quality of LLC replacement for multi-threaded applications [31]. As a part of our workload characterization, we explore the influence of the read-write shared blocks on the volume of LLC hits. Even though the hardware-managed policies have not paid much attention to shared data, compiler transformations to enhance the locality of shared data in multi-threaded workloads have been proposed [16].

Policies to dynamically partition a shared LLC among the competing threads of a CMP have been proposed [26], [33], [35], [40]. However, these policies assume that the threads are independent (as in a multi-programmed workload) and do not take into account any cross-thread data sharing. Apart from these, there is a proposal that partitions each set in the shared LLC into private and shared ways by dynamically choosing one of the four predefined partitions [9]. Since the possible partitions are statically predefined, the policy can only approximately match the optimal need of the workloads. Also, the policy, at the time of filling a block into the LLC, needs to infer whether the block will be shared during its residency in the LLC. Accordingly, the policy assigns the block to the shared or the private partition of the target set. The proposal uses a simple heuristic for this purpose that infers a block being filled into the LLC to be shared if it has already been shared in the past or if it is being filled by a core that is different from the core which filled the block during its last residency in the LLC. Once a cache block is identified as shared, this information is tracked and the block is inferred as shared during all subsequent fills into the LLC. We will refer to this policy as sharing-aware-partitioning (SA-Partition). We show that for the workloads considered in this paper, this simple heuristic is ineffective in inferring whether a block will be private or shared during its current residency in the LLC.

The PARSEC, SPLASH-2, and SPEC OMP benchmark suites are widely used in the community and there have been prior works that characterize the behavior of these applications on CMP systems [1], [2], [4], [5], [6], [38]. These studies analyze various characteristics of these applications including memory characteristics such as working set sizes, amount of sharing, true/false sharing, and the nature of cross-thread communication. The shared LLC behavior of the emerging recognition, mining, synthesis (RMS), and bioinformatics workloads has been evaluated in detail on CMP systems in prior studies [15], [23], [24], [29]. The effectiveness of compiler-directed software prefetching on these workloads has been evaluated [30]. These studies conclude that these applications, in general, are memory intensive, have large working set sizes, and enjoy good locality. These studies also show that a shared LLC per-

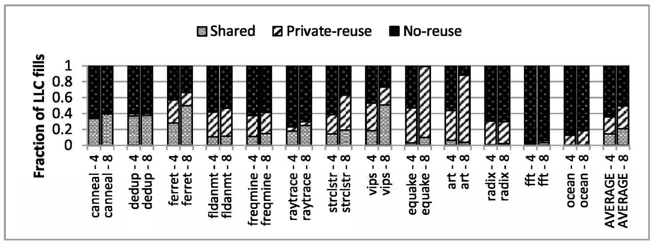


Fig. 1: Distribution of the LLC fills based on the reuse categories for 4 MB and 8 MB LLCs with Belady's optimal replacement policy.

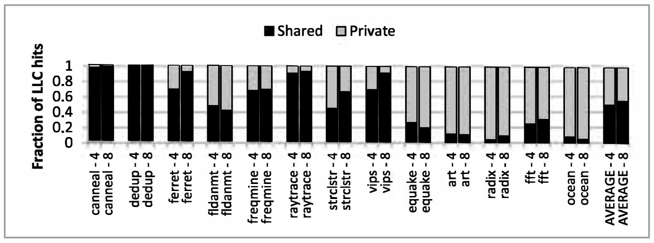


Fig. 2: Distribution of the LLC hits enjoyed by the private and shared cache blocks for 4 MB and 8 MB LLCs with Belady's optimal replacement policy.

forms better than an iso-capacity configuration of private per-core LLCs. Our work differs from these prior characterization studies in that we explore the influence of LLC replacement policies on the data sharing behavior of multi-threaded applications.

III. Characterization Methodology

We use the Multi2Sim simulator [37] to generate LLC access traces from the applications. We model a CMP with eight single-threaded x86 cores. Each core has private L1 and L2 caches and the LLC is shared among all the cores. The L1 instruction and data caches are 32KB 8-way set-associative and use the LRU replacement policy. The per-core unified L2 cache is 128KB 8-way set-associative and use the LRU replacement policy. The load/store micro-ops that miss in the L2 cache are issued to the shared LLC and recorded in our LLC access trace. Each element of the trace contains the requested address, the requesting core id, the program counter of the instruction that generated the LLC access, and the request type (i.e., instruction or data fetch and load or store). An offline LLC model digests each workload trace and generates the statistics of the desired characteristics. We model a 16-way set-associative shared inclusive LLC and experiment with two different capacities, namely, 4 MB and 8 MB. The block size in all caches is 64 bytes.

We select eight applications from the PARSEC suite (canneal, ferret, fluidanimate, freqmine, vips, streamcluster, raytrace, dedup), three from the SPLASH-2 suite (fft, ocean contiguous, and radix), and two from the SPEC OMP suite (art and equake). The PARSEC applications are run for the entire regions of interest. We use the `simmedium` input sets provided with PARSEC for all applications except canneal. For canneal, we use the `simlarge` input set. For the SPEC OMP applications, we use the MinneSPEC [21] inputs (for equake, the ARCHduration is set to 0.5) and simulate the entire parallel regions of the applications. The SPLASH-2X applications distributed with the PARSEC suite are used with the `simmedium` input sets and simulated for the entire regions of interest.

IV. Importance of Cross-thread Reuses in Multi-threaded Applications

A fill that brings a cache block into a shared LLC can be classified into one of three categories, namely, no-reuse fill, private-reuse fill, and shared fill. A no-reuse fill brings a cache block that does not experience any reuse during its residency in the LLC. A private-reuse fill brings a cache block that experiences LLC reuses only from the thread that brought the block to the LLC. A shared fill brings a cache block that enjoys reuses from multiple threads during its residency in the LLC. Figure 1 shows the distribution of the LLC fills for each application for 4MB (left bar in each group) and 8MB (right bar in each group) LLCs running Belady's optimal replacement policy. We use Belady's optimal policy in our

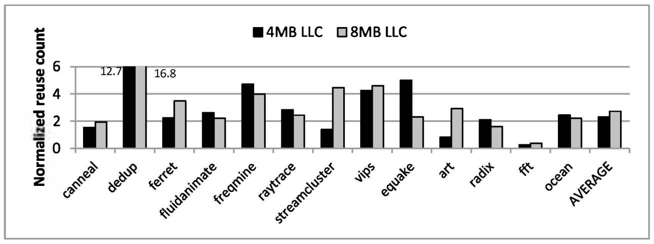


Fig. 3: Reuse count per shared LLC fill normalized to the reuse count per private-reuse LLC fill for 4 MB and 8 MB LLCs with Belady's optimal replacement policy.

analysis to understand the true nature of the LLC reuses in these applications. On average, the majority of the LLC fills are no-reuse fills in both 4MB (64%) and 8MB (50%) LLCs. The average number of private-reuse and shared fills is 29% and 21% for an 8MB LLC. Applications such as canneal, dedup, ferret, raytrace, and vips experience more shared fills than private-reuse fills. The average number of private-reuse and shared fills for a 4MB LLC is 21% and 15%, respectively. These results show that the shared fills constitute a significant fraction of the useful LLC fills in multi-threaded applications.

To further understand the sources of the LLC hits, Figure 2 compares the number of LLC hits to the private and shared cache blocks in 4MB and 8MB LLCs running Belady's optimal policy. On average, 51% and 56% of the LLC hits are to the shared cache blocks in 4MB and 8MB LLCs, respectively. The fraction of the LLC hits to the shared cache blocks decreases with decreasing LLC capacity, since the likelihood of a cache block being evicted before experiencing the accesses from all its sharers is higher in a smaller-capacity LLC. Shared cache blocks enjoy a significant fraction of the LLC hits (more than 90%) in applications such as canneal, dedup, ferret, raytrace, and vips in an 8MB LLC. The trends are similar in a 4MB LLC. These results suggest that a shared fill, which brings a shared cache block into the LLC, is more valuable than a private-reuse fill in multi-threaded applications. To further quantify this aspect, we compare the average number of reuses experienced by a private-reuse fill to that of a shared fill. Figure 3 shows the average reuse count per shared fill normalized to the average reuse count per private-reuse fill in these applications for 4MB and 8MB LLCs running Belady's optimal policy. On average, a shared LLC fill experiences 2.3 and 2.7 times more reuses than a private-reuse fill in 4MB and 8MB LLCs, respectively. On average, a shared LLC fill experiences an order of magnitude more reuses than a private-reuse fill in dedup for both 4MB and 8MB LLCs. These results indicate that the shared LLC fills are more valuable than the private-reuse fills in several multi-threaded applications. A sharing-aware LLC replacement policy can identify and protect the cache blocks brought in by the shared fills.

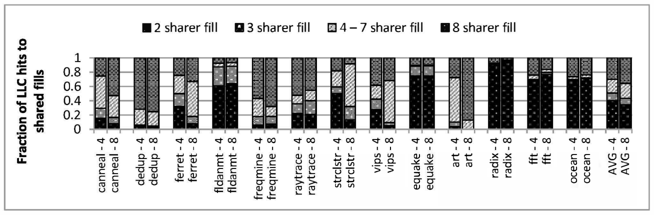


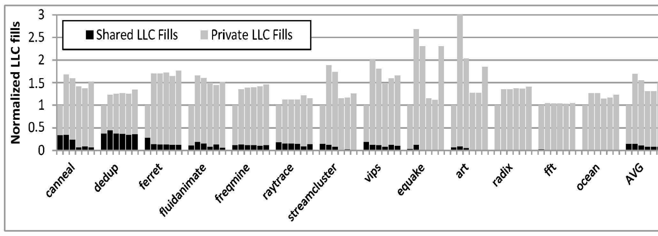
Fig. 4: Distribution of the LLC hits enjoyed by the shared fills of different sharing degree categories for 4 MB and 8 MB LLCs with Belady's optimal replacement policy.

Figure 4 further investigates the reuse behavior of the shared fills classified based on the number of sharers. A shared fill is classified into one of the four categories based on the number of sharers the filled block experiences before it is evicted from the LLC. Figure 4 shows the contribution of each category to the LLC hits enjoyed by the shared fills. On average, the 2-sharer, 3-sharer, 4-7-sharer, and 8-sharer fills contribute 34%, 9%, 21%, and 36% respectively of all LLC hits to the shared blocks in an 8MB LLC. These numbers for a 4MB LLC are 40%, 10%, 20%, and 30%, respectively. These results show that not all shared fills are equally important and the 2-sharer and 8-sharer fills together contribute to about 70% of all the cross-thread LLC reuses. In canneal, dedup, ferret, freqmine, raytrace, streamcluster, vips, and art, the LLC fills that experience more than three sharers contribute to majority of the cross-thread LLC reuses. In summary, the results discussed in this section bring out three important facts. First, the shared fills experience more LLC hits than the private fills. Second, each shared fill enjoys more than twice the number of LLC hits than a private fill, on average. Third, the importance of a shared fill depends on the degree of sharing that the filled block experiences during its residency in the LLC.

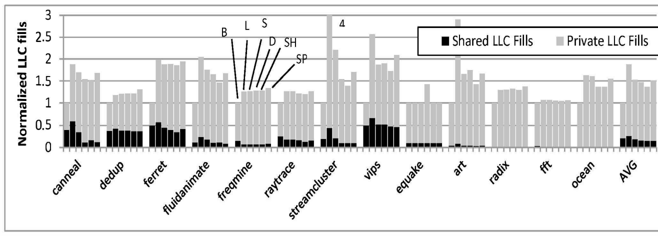
V. Quantifying Sharing-awareness

The LLC replacement policy can influence the amount of cross-thread sharing in the LLC. If the replacement policy prematurely evicts the shared blocks before they are accessed by all their sharers, the amount of sharing can decrease significantly. On the other hand, if the accesses from the sharers to a shared block are very far apart, retaining such a block until all the sharers access the block may, in fact, be suboptimal and hurt performance. As an example, consider a cache block which is supposed to be accessed by k distinct sharers in a certain phase of execution. A policy that is not sharing-aware may cause k private fills of this cache block in the worst case leading to a zero fraction of shared fills and only one sharer per fill on average (the private blocks are defined to have one sharer). On the other hand, depending on the inter-core sharing distance, a policy that is more sharing-aware may have less than k fills, some of which are shared fills leading to a non-zero fraction of shared fills and larger than one sharer per fill on average. The fraction of shared fills and the average number of distinct sharers per fill are expected to be good indicators of sharing-awareness of a policy.

To evaluate the impact of LLC management policies on data sharing, Figure 5 compares Belady's optimal policy, LRU policy, two-bit SRRIP policy, two-bit DRRIP policy, SHiP-PC policy, and SA-Partition policy in terms of the volume of LLC fills partitioned into shared and private fills normalized to Belady's optimal policy. The upper and lower panels show the results for a 4 MB LLC and an 8 MB LLC, respectively. As expected, the data in Figure 5 show that Belady's optimal policy has the lowest number of LLC



(a) 4MB LLC



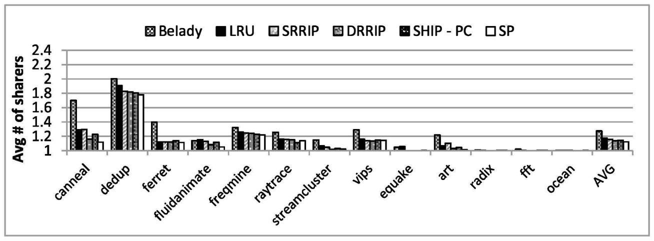
(b) 8MB LLC

Fig. 5: Number of shared and private LLC fills in the LRU (L), SRRIP (S), DRRIP (D), SHiP-PC (SH) and SA-Partition (SP) replacement policies normalized to the number of LLC fills by Belady's (B) optimal replacement policy in 4 MB and 8 MB LLCs.

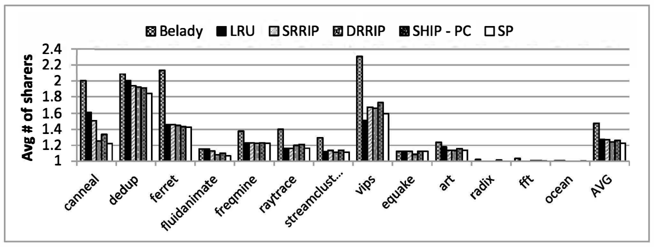
fills (same as LLC misses). The results also show that SHiP-PC is the best performing among the remaining policies, while LRU is the worst on both 8 MB and 4 MB LLCs. We also note that on average the SA-Partition policy suffers from as many LLC misses as the SRRIP policy, which it uses as the baseline policy (the original proposal of SA-Partition used LRU as the baseline policy, which we replace by SRRIP because SRRIP outperforms LRU by a significant margin). Only streamcluster gains significantly from the partitioning technique of this policy. The performance of the SA-Partition policy depends on the quality of its heuristic to predict the shared fills, which we have already discussed. In Section VII, we show that this heuristic cannot offer acceptable levels of accuracy.

As mentioned before, the sharing-awareness of a replacement policy is indicated by the fraction of shared fills made by the policy. A higher fraction of shared fills corresponds to a larger volume of cross-thread sharing in the LLC. Belady's algorithm has optimal sharing-awareness because it has knowledge about all future reuses. On average, for an 8 MB LLC, 21% of LLC fills in Belady's policy are shared. This figure is 14% for a 4 MB LLC. Although the LRU policy on an 8 MB LLC exhibits a larger volume of shared fills compared to the optimal policy, the number of shared fills as a fraction of all fills in the LRU policy is much smaller. The other policies also exhibit significantly lower fraction of shared fills when compared to Belady's policy.

Another way of quantifying sharing-awareness of an LLC management policy is to measure the average number of distinct sharers per fill into the LLC. Figure 6 compares various policies in terms of this metric. For an 8 MB LLC, the average number of sharers per fill observed by Belady's policy, LRU, SRRIP, DRRIP, SHiP-PC, and SA-Partition is 1.48, 1.27, 1.27, 1.24, 1.26. and 1.23. These figures for a 4 MB LLC are 1.27, 1.17, 1.16, 1.14, 1.14, and 1.12. As expected, with decreasing LLC capacity the average number of sharers per fill decreases. We see considerable variability in the average number of sharers per LLC fill for different policies in applications such as canneal, dedup and vips. This indicates that the LLC replacement policy can significantly affect the amount of data sharing in the LLC. From the figure it is clear that, compared

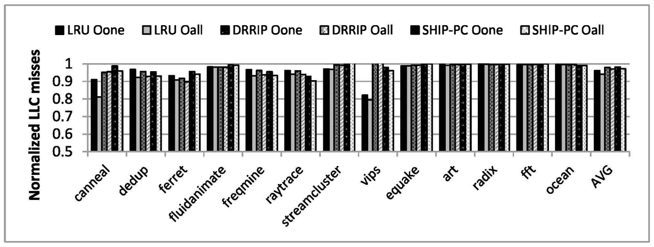


(a) 4MB LLC

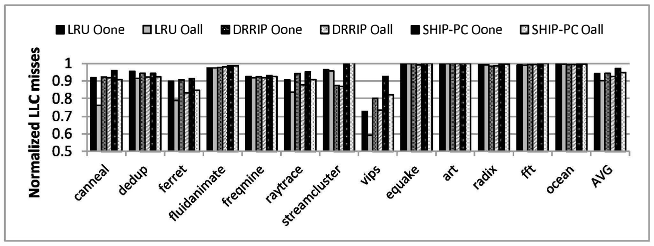


(b) 8MB LLC

Fig. 6: Average number of distinct sharers per LLC fill for various replacement policies in 4 MB and 8 MB LLCs.



(a) 4MB LLC



(b) 8MB LLC

Fig. 7: Number of LLC misses experienced by the sharing-aware oracles normalized to the corresponding baseline policies for 4 MB and 8 MB LLCs.

to Belady's policy, the LRU, SRRIP, DRRIP, SHiP-PC, and SA-Partition policies prematurely evict several shared blocks that, if retained longer, could have enjoyed accesses from more sharers. This, in turn, could have saved several LLC misses to the shared blocks. There is a large gap between the sharing-awareness of the existing LLC replacement policies and the optimal level of sharing-awareness. The biggest difference between the optimal policy and the other policies is observed in canneal, dedup, ferret, and vips. For an 8 MB LLC, the optimal number of distinct sharers per LLC fill is at least two in these applications. These are also the applications that exhibit high fractions of shared fills in the optimal policy (canneal: 40%, dedup: 38%, ferret: 50%, vips: 51% for 8 MB LLC), as shown in Figure 5. On the other hand, the SPLASH-2 applications (fft, ocean, radix) have mostly private blocks.

VI. Sharing-aware Replacement Policies: A Generic Design

The data presented in the last section show that several recently proposed LLC management policies fall significantly short of the optimal policy in terms of sharing-awareness. However, these data do not offer any direction as to how one can introduce sharing-awareness in an existing policy and how much performance benefit can come from introducing sharing-awareness. This section discusses a general approach to designing a sharing-aware policy on top of an existing baseline and evaluates two oracle policies to explore the performance potential that can be uncovered by introducing sharing-awareness in an existing policy.

Consider a baseline LLC management policy P. On top of P, we design two oracles O_{one} and O_{all}, which have differing degrees of sharing-awareness. The input to the oracles is a description of P and the usual LLC access trace with some additional annotations that we discuss below. To generate the annotations, we execute the LLC access trace in the presence of Belady's algorithm. Each eviction from the LLC by Belady's algorithm is marked in the access trace. Since a cache block may have to be filled multiple times into the LLC, each such fill is defined to start a new *optimal lifetime* of the cache block in the LLC and the life lasts until it gets evicted from the LLC.

The oracles digest the annotated LLC access trace while simulating the policy P. On every LLC miss that P suffers from, the oracles consult the annotations to look ahead into the future and determine the number of distinct cores that access the block until the end of its current optimal LLC lifetime. If the number of such cores is more than one, the block is marked as a shared block in the LLC and its number of sharers is also recorded in the extended tag. In O_{one}, a cache block remains marked as shared in the LLC until it has seen the first sharing access (an access from a core different from the one that filled the block) from any of the expected sharers. In O_{all}, a cache block remains marked as shared in the LLC until it has seen at least one access from each of its sharers. Only cache blocks that are not marked as shared are considered by policy P when choosing a replacement victim from an LLC set. Therefore, the oracles augment P with the optimal sharing information and protect the cache blocks that become shared in the future, thereby increasing data sharing in the LLC.

Figure 7 shows the number of LLC misses of the two oracles working with baseline LRU, DRRIP, and SHiP-PC normalized to each of the baseline policies. For an 8 MB LLC with the LRU policy, the O_{one} oracle saves up to 27% LLC misses (vips) and 6% LLC misses on average compared to the baseline LRU policy. These figures for the O_{all} oracle are 41% (maximum) and 10% (average), respectively. Some of the top gainers of O_{all} include canneal (24%), ferret (21%), raytrace (16%), and vips (41%). With the DRRIP policy on an 8 MB LLC, the O_{one} oracle saves up to 20% LLC misses (vips) and 6% LLC misses on average compared to the baseline DRRIP policy. These figures for the O_{all} oracle are 24% (maximum) and 8% (average), respectively. Relative to the baseline SHiP-PC policy, the O_{one} and O_{all} oracles save 3% and 5% LLC misses on average for the 8 MB LLC. The biggest gainer for O_{one} on SHiP-PC is ferret (9% LLC miss saving) and for O_{all} on SHiP-PC it is vips (18% LLC miss saving). Overall, the O_{all} oracle is more effective than the O_{one} oracle in applications that have more intense sharing (see Figures 5 and 6). The trends are similar for a 4 MB LLC, but the oracles are less effective due to the smaller-capacity LLC, as expected.

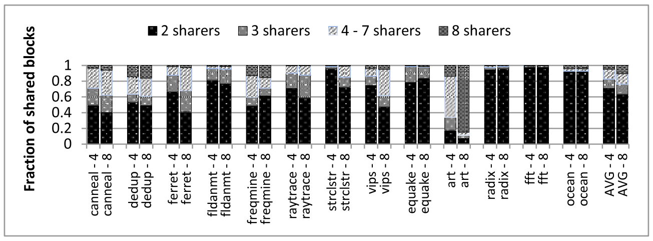


Fig. 8: Distribution of the shared fills to the LLC in Belady's optimal replacement policy categorized based on the number of sharers for 4 MB and 8 MB LLCs.

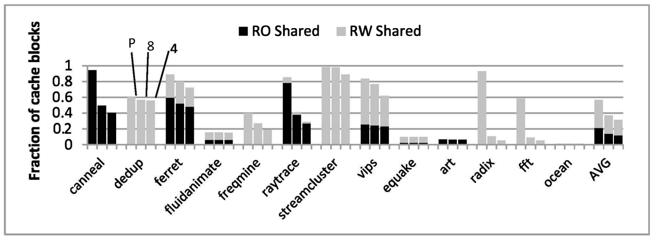


Fig. 9: Fraction of cache blocks shared at program level (P) and during at least one LLC lifetime for 4 MB and 8 MB LLCs with Belady's optimal replacement policy.

To further understand the differing effectiveness of the two oracles, Figure 8 shows the distribution of the shared LLC fills experienced by Belady's optimal policy in terms of the number of distinct sharers. For each application, we show the data for 4 MB (left bar) and 8 MB (right bar) LLCs. On average, 71% and 63% of the shared fills observe only two sharers for 4 MB and 8 MB LLCs, respectively. These fills can be covered by the O_{one} oracle. Further, the O_{one} oracle also satisfies at least one cross-thread LLC reuse of the shared fills having more than two sharers. The applications that gain most from the O_{all} oracle have high fractions of shared fills observing more than two sharers. These are canneal, dedup, ferret, and vips. Although art has a high fraction of shared fills with more than two sharers, it does not gain much from the O_{all} oracle because this application has a small overall volume of shared fills (4% of all fills in 8 MB LLC), as shown in Figure 5.

These oracles offer important insight into how sharing-awareness can be introduced in an LLC management policy. The oracles, as designed, need assistance from Belady's optimal policy. Realistic implementations of the oracles need two pieces of information at the time of filling a block into the LLC. They need to predict if the block being filled is likely to be shared during its optimal lifetime in the LLC. If the block is inferred shared, the sharing-aware policies need to have an estimate of the number of distinct sharers for this block. In the next section, we explore the feasibility of implementing a predictor that infers, at the time of filling a block, if the block is going to be shared. If we can design such a predictor with high enough accuracy, we can easily implement the O_{one} oracle, which would retain the inferred shared blocks until they see their first sharing access. Mispredictions are costly because predicting an actually private block as shared can occupy cache space for an unnecessarily long time and may degrade performance. One way to handle such mispredictions is to have a time-out mechanism that would unmark a predicted shared block if it fails to see a sharing access within the time-out period.

VII. Challenges in Realizing Sharing-aware Replacement Policies

In this section, we analyze how data is shared and utilized in the applications, and discuss the implications of these characteristics on the design of sharing-aware replacement policies. In these characterizations, we use Belady's optimal policy for LLC replacement so that our conclusions can highlight the true nature of the characteristics and are free of any implementation artifacts.

A. *Data Sharing in Multi-threaded Applications*

The amount and nature of data sharing in the LLC in multi-threaded applications are dependent not only on the application characteristics, but also on the capacity of the LLC and the LLC management policy. If the LLC cannot accommodate the entire shared working set of an application, it can hurt cross-thread sharing observed in the LLC. We refer to the amount of sharing observed in an application for a given LLC configuration as its LLC lifetime sharing. A cache block is said to be shared during its LLC lifetime if it is accessed by more than one core while it resides in the LLC. A cache block can have multiple lifetimes depending on how many times it is filled into the LLC. A shared cache block always refers to a cache block that is shared during at least one of its LLC lifetimes. While the LLC lifetime sharing of an application is dependent on the LLC configuration, the maximum possible sharing in an application occurs when there is no constraint on the LLC capacity and it can accommodate the entire shared working set. We refer to this theoretical limit on sharing that occurs with an infinite LLC as *program-level* sharing. A cache block is shared at the program level if it is accessed by more than one core, even across different LLC lifetimes, over the course of execution of the entire application. Program-level sharing is purely an application characteristic and is unaffected by the LLC configuration.

Figure 9 compares the fraction of memory blocks that are shared during at least one LLC lifetime running Belady's optimal policy with program-level sharing. For each bar, each block is classified as read-only shared or read-write shared. On average, although 57% of the memory blocks are shared at the program level in these applications, only 37% of them are shared in an 8 MB LLC and 31% in a 4 MB LLC even with the optimal replacement policy. These results show that the inter-core sharing distances of the shared memory blocks in these applications are large and only a fraction of these can be captured by the optimal policy. The sharing distances that are beyond the LLC reach lead to premature eviction from the LLC before all the accesses from the sharing cores take place. In fact, these blocks appear to be private even to the optimal replacement policy due to LLC capacity constraints. The largest differences between program-level sharing and LLC lifetime sharing are exhibited by canneal, raytrace, radix, and fft. We examine ferret and canneal in greater detail.

Ferret uses a database that is shared among all the threads in the program and is queried throughout the course of execution of the application. Since the threads do not coordinate their queries, accesses by different threads can be widely spread apart. As a result, while an entry from the database can be accessed by different cores thereby experiencing program-level sharing, the cache block holding the entry may never get shared during any LLC lifetime.

Canneal is another application where the uses of the shared data in the LLC by different threads are widely spread apart leading to a large fraction of the cache blocks being shared at the program-level, but relatively few blocks shared during an LLC lifetime. Canneal uses simulated annealing to optimize the routing cost for a chip design. The program has a large shared data structure that is accessed by all the threads in the program.

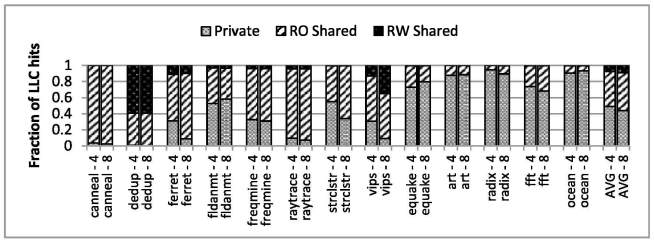


Fig. 10: Distribution of the LLC hits to different categories of cache blocks for 4 MB and 8 MB LLCs under Belady's optimal replacement policy.

Listing 1 shows the main loop of the program which is parallelized. Each thread randomly chooses two different elements, `a` and `b`, from `_netlist` and performs some computations in the function `calculate_delta_routing_cost` using the two elements as inputs. If an element is accessed by two different threads while cached in the LLC, it becomes shared during an LLC lifetime. Since the elements are chosen at random, accesses by different threads can happen at irregular intervals. The random nature of the accesses also results in some elements from `_netlist` being shared frequently while others never get shared. It also results in a highly irregular sharing pattern where a cache block is shared during some LLC lifetimes but not during others.

```
netlist_elem *a, *b;
long a_id, b_id;
Rng rng; //store of randomness
....
for (i = 0; i < _moves_per_thread_temp; i++){
      //get new element b different from a
      a = b;
      a_id = b_id;
      b = _netlist->get_random_element(&b_id, a_id, &rng);
      routing_cost_t delta_cost =
            calculate_delta_routing_cost(a,b);
      ....
}
```

Listing 1: Canneal main loop

Given the sharing behavior of the multi-threaded applications, a simple heuristic to decide if a block will be shared during its residency in the LLC, like the one used by SA-Partition, will be ineffective. Recall that SA-Partition identifies a cache block as shared if it has been shared in the past or it is filled by a core that is different from the core which filled the block during its previous LLC lifetime. However, if the accesses by the two cores are spread wide apart, such a cache block will never be shared during an LLC lifetime. Essentially, SA-Partition tries to capture program-level sharing, which may significantly depart from the optimal sharing behavior for a particular LLC configuration, as shown in Figure 9.

Returning to our discussion on Figure 9, we find that the applications show varying degrees of read-only and read-write sharing. While canneal, raytrace, and art have mostly read-only shared data, dedup, freqmine, streamcluster, equake, radix, fft, and ocean have mostly read-write shared data. Both types of sharing are experienced by ferret, fluidanimate, and vips.

A recent proposal (CSHARP) [31] argues that offering extra protection to the dirty shared blocks can improve the LLC performance. To understand the benefits of such a technique, Figure 10 explores the composition of the LLC hits observed by Belady's optimal policy. A shared fill that enjoys only read hits in the LLC is classified as a read-only shared fill. A read-write shared fill is defined similarly. All LLC hits to a private-reuse fill are counted as private. All LLC hits to a read-only (read-write) shared fill are counted as read-only shared (read-write shared). For each application, the left bar shows the data for a 4 MB LLC and the right bar for an 8 MB LLC. In general, most hits to the shared blocks are contributed by the read-only shared blocks except in dedup and, to some extent, in vips. On average, in an 8 MB LLC, 47% and 9% of the LLC hits with Belady's optimal policy come from the read-only and read-write shared blocks, respectively. These figures for a 4 MB LLC are 44% and 7%, respectively. In summary, the read-write shared blocks are not a major source of the shared hits in the LLC even for the optimal LLC replacement policy and hence, a policy like CSHARP may not be effective for this set of applications. Instead, biasing the sharing-awareness of a policy toward the read-only shared blocks may be beneficial, but separating such blocks from the others at the time the LLC fill takes place is not easy. We explore the general problem of identifying the shared fills next.

B. *Predictability of Sharing in Multi-threaded Applications*

In Section VI we concluded that a realistic implementation of a sharing-aware replacement policy would require a highly accurate predictor that infers, at the time a block is filled into the LLC, whether the block is likely to be shared during its residency in the LLC. In this section, we explore the feasibility of designing such a predictor. Recall that each fill into the LLC uniquely defines one LLC lifetime of a memory block and the life lasts until the block is evicted from the LLC. To predict the nature (shared or private) of each life of a memory block, we first represent the behavior of the block as a binary string of two symbols, namely, *P* for private and *S* for shared. This allows us to refer to the sharing behavior of a memory block as a *sharing history*, similar to the branch history that a branch instruction possesses. A memory block that is always private or always shared has a unary history string, while a block that is shared in only a subset of its lifetimes presents a more challenging task of predicting the nature of its next lifetime given its binary history string. In the following, we focus only on the shared blocks (a block that is shared in at least one of its LLC lifetimes) and explore the feasibility of designing a predictor that predicts the nature (private or shared) of the next LLC lifetime of the block, given a *history window* of the last *w* LLC lifetimes of the block. Following the branch prediction terminology, this can be termed a local history-based predictor.

We start our analysis by exploring how regular the sharing history of a shared block is. While discussing the main loop of Canneal, we have already pointed out that a shared cache block in this application may not be shared in each of its LLC lifetimes. Figure 11 shows the distribution of the shared cache blocks based on the fraction of the LLC lifetimes during which they are shared. For each application, the left bar is for a 4 MB LLC, while the right bar is for an 8 MB LLC, both running Belady's optimal policy. Each bar shows the fraction of shared blocks that are shared in less than 50%, 50%-90%, and more than 90% of each of the blocks' LLC lifetimes. For example, a shared block that is shared in forty LLC lifetimes out of its total of hundred lifetimes would be included in the first of the three categories. On average, for an 8 MB LLC, about 60% of the shared blocks are shared in more than 50% of their LLC lifetimes and only one-third of the shared blocks are shared in more than 90% of their LLC lifetimes. For a 4 MB LLC, these figures are 44% and 14%, respectively. It is clear that a shared block is only sparsely shared across its lifetimes and the sparseness only increases as the LLC capacity decreases, which is an expected behavior. Further, there is significant variability across the applications. On an 8 MB LLC, fluidanimate, equake,

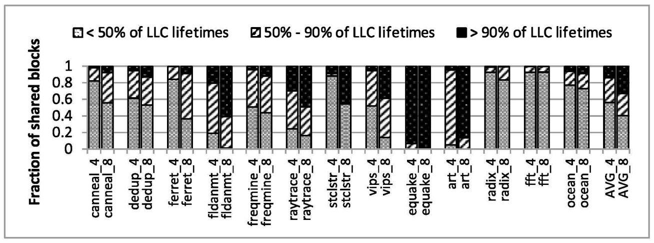


Fig. 11: Distribution of the shared blocks based on LLC lifetime sharing for 4 MB and 8 MB LLCs with Belady's optimal replacement policy.

and art exhibit very dense sharing with at least 60% shared blocks being shared in more than 90% of their LLC lifetimes. On the other hand, canneal, dedup, streamcluster, radix, fft, and ocean show very sparse sharing with at least 50% shared blocks being shared in less than 50% of their LLC lifetimes. From these data we conclude that the sharing history of the shared blocks in most of the applications is expected to be irregular and sparse. These results further emphasize that a simple heuristic to predict the nature (private or shared) of a fill, like the one used in SA-Partition, will be ineffective for these applications. Recall that in SA-Partition, once a cache block is identified as shared during a particular LLC lifetime, all subsequent LLC lifetimes of that block are predicted to be shared. But the results in Figure 11 show that such a heuristic is highly inaccurate. In the rest of the analysis, we focus only on the PARSEC applications, as the remaining applications have low volumes of LLC lifetime sharing (Figure 9) and almost no improvement with the oracles (Figure 7).

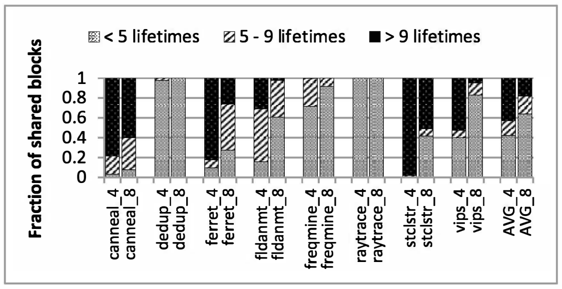
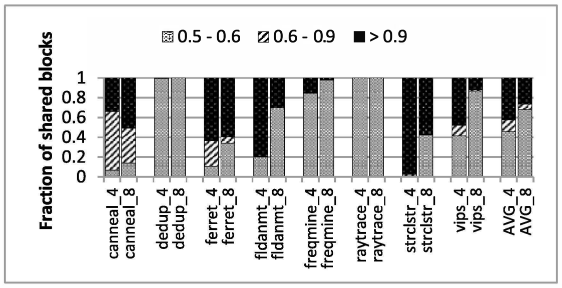


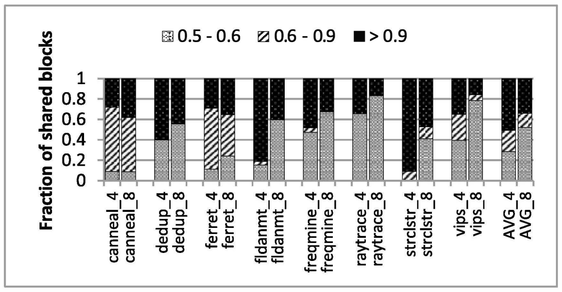
Fig. 12: Distribution of the shared cache blocks based on the number of LLC lifetimes for 4 MB and 8 MB LLCs with Belady's optimal replacement policy.

Figure 12 further quantifies the distribution of the number of LLC lifetimes across the shared blocks. On average, for an 8 MB LLC, 64% of the shared blocks experience less than five LLC lifetimes, while only 18% of the shared block experience more than nine LLC lifetimes. As expected, for a smaller LLC, the shared blocks experience larger numbers of LLC lifetimes with 43% of the shared blocks having more than nine LLC lifetimes. The distribution of the number of LLC lifetimes is directly correlated to the shared data working set size of an application and canneal, ferret, and streamcluster show a higher fraction of shared blocks experiencing larger numbers of LLC lifetimes on an 8 MB LLC. On a 4 MB LLC, canneal, ferret, fluidanimate, streamcluster, and vips experience larger numbers of LLC lifetimes for most of the shared blocks. On the other hand, dedup, freqmine, and raytrace have relatively small shared working sets and show larger fraction of smaller LLC lifetime counts.

Our history-based sharing behavior predictor uses a history of length w bits. We first collect the entire sharing history H_A of each



(a) History window size four bits



(b) History window size two bits

Fig. 13: Distribution of the shared addresses based on the sharing predictability index with history window sizes of four and two bits for 4MB and 8MB LLC

shared block address A under Belady's optimal policy. For each shared block address A, we move a sliding window of length w over the entire history H_A. For each history pattern h of length w bits encountered in the process (subset of the possible 2^w patterns), we record the number of times the block's next LLC lifetime is private and the number of times the block's next lifetime is shared. Let these counts be p_h and s_h, respectively. Thus, given a history pattern h of length w bits, the probability that the block's next LLC lifetime is shared is $s_h/(p_h + s_h)$. The pattern h is a good indicator of the sharing behavior of the next LLC lifetime if the aforementioned probability is either close to one (shared lifetime) or close to zero (private lifetime). We define the predictability index of a shared block at address A for w-bit history as

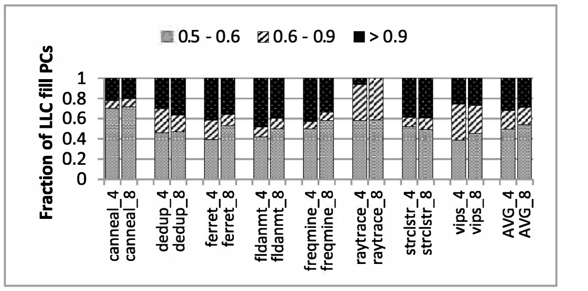
$$P_A(w) = \frac{1}{N_w} \sum_h \frac{\max(p_h, s_h)}{p_h + s_h}, \qquad (1)$$

where the sum is over all w-bit history patterns h captured by the sliding window and N_w is the number of such distinct patterns. P_A lies between 0.5 and one and indicates how accurately we can predict the sharing behavior of a shared block at address A, given the recently seen w LLC lifetimes of the block. If P_A is close to one, such a predictor can predict with high accuracy the nature of the current LLC lifetime of the block when it is filled into the LLC. On the other hand, a value close to 0.5 indicates a poor prediction accuracy. To be able to cover most of the shared cache blocks from most of the applications, we use history lengths less than five (see Figure 12). In particular, we explore history lengths of four and two.

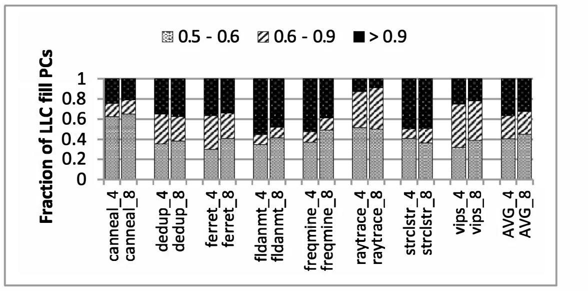
Figure 13 shows the distribution of the shared blocks based on the computed predictability. For each application, the left bar shows the results for a 4 MB LLC and the right bar for an 8 MB LLC. On average, for a four-bit history on an 8 MB LLC, only 26% of the shared blocks show a predictability value of more than 0.9. On a 4 MB LLC, this figure improves to 42%. This is expected because most shared blocks on a 4 MB LLC spend more of their LLC lifetimes in private mode leading to a lower-entropy history and better predictability compared to an 8 MB LLC. The predictability, in general, improves with a two-bit history (34% and 51% of the shared blocks have more than 0.9 predictability on 8 MB and 4 MB LLCs). A short and more recent two-bit history offers higher accuracy in dedup, fluidanimate, freqmine, and raytrace. However, for the applications with intense sharing such as canneal, ferret, streamcluster, and vips, a longer history helps more by pushing more shared blocks into the upper 0.9 predictability group for a four-bit history. These are some of the applications that show large improvements with the oracles (Figure 7). In general, we find that none of the applications (except fluidanimate and streamcluster on a 4 MB LLC) enjoys a high sharing predictability.

Instead of designing a predictor that learns the sharing pattern for each individual LLC block, it is possible to learn this pattern for each program counter (PC) of the memory access instructions that trigger LLC fills. Such a predictor, on encountering a fill from a particular PC, would predict the nature of the current LLC lifetime of the block being filled based on the sharing history exhibited by the blocks already filled by this PC. We next conduct the same predictability study for each PC that brings at least one block into the LLC. Each such PC triggers a sequence of fills into the LLC over the entire execution of the application. Each such fill leads to a private or shared LLC lifetime of the block being filled. Thus we can attach a sharing history with each PC in the same way as we attach a sharing history with a shared block. Therefore, we can define a predictability index P_{PC} for each fill PC in the same way as shown in Formula (1). Figure 14 shows the distribution of the fill PCs based on their predictability index. The trends are very similar to the address-based predictability study. On average, for a history length of four bits on an 8 MB LLC, 29% of the fill PCs show a predictability of more than 0.9, while for a two-bit history, this figure improves to 32%. For dedup and freqmine, the fill PC-based predictability is much better than the address-based predictability (compare the percentages in the upper 0.9 category), while for the other applications, the address-based predictability is higher. Overall, the introduction of the fill PC does not help improve the sharing predictability for this set of applications. In general, a particular fill PC brings a large number of memory blocks into the LLC. If all these blocks do not exhibit similar LLC lifetime sharing history, the sharing history irregularity of each of these blocks only adds up and makes the mixed sharing behavior of all LLC blocks filled by a particular PC even more unpredictable.

We evaluate the effectiveness of address and PC-based sharing behavior prediction by augmenting the DRRIP and SHiP-PC policies with a sharing behavior predictor. The predictor identifies if an LLC fill brings a cache block that will be shared during its current LLC lifetime. Once identified, such a cache block is inserted at the highest priority in the LLC (RRPV of 0 for the DRRIP and SHiP-PC policies). The sharing behavior predictor is implemented as a 16K-entry table which is indexed using a 14-bit PC or address hash similar to SHiP-PC. Each entry in the table maintains a 2-bit saturating counter, which records the sharing behavior history. On every LLC eviction, the history counter is incremented if the evicted cache block is shared, else it is decremented. On an LLC fill, the predictor table is consulted with the fill PC or fill address. If the indexed 2-bit counter has a value of three, the fill is predicted to be shared and inserted in the LLC with the highest priority. Otherwise the underlying replacement policy (DRRIP or SHiP-PC) decides the insertion priority. Our evaluations show negligible improvement with such a sharing behavior predictor for both the



(a) History window size four bits



(b) History window size two bits

Fig. 14: Distribution of the LLC fill PCs based on the sharing predictability index for 4 MB and 8 MB LLCs with Belady's optimal replacement policy.

DRRIP and SHiP-PC policies and we attribute this to the low predictability of the sharing patterns.

Overall, our analysis indicates that the sharing behavior of multi-threaded applications does not correlate well with the sharing history of the shared block addresses or LLC fill PCs. A sharing-aware policy must explore beyond these commonly used techniques to predict the sharing behavior of the LLC blocks.

VIII. Summary

In this paper we investigate the need for sharing-aware LLC replacement policies and their impact on the LLC performance of multi-threaded applications. We show that the shared LLC blocks contribute more to the LLC hits than the private LLC blocks in multi-threaded applications. We show that the LLC replacement policies significantly affect cross-thread data sharing in the LLC and that introducing sharing-awareness can significantly improve the performance of a range of LLC replacement policies. We present a thorough analysis of how data is shared and utilized in the LLC in multi-threaded applications and highlight the implications of these characteristics on the design of sharing-aware LLC replacement policies. We propose a generic approach to incorporate sharing-awareness in the existing LLC replacement policies. At the heart of this generic design is a sharing predictor that, on an LLC fill, predicts if the currently filled block is likely to be shared during its residency in the LLC. Based on the characteristics of the multi-threaded applications, we explore two designs of such a sharing predictor and compute the predictability limits of these designs as a function of the history length. Based on these studies we conclude that the address-based and fill PC-based sharing predictors do not offer adequate levels of accuracy and there remains a need for better architectural and high-level program semantics features for designing such predictors with high accuracy.

IX. Acknowledgements

We would like to thank Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney and Antonia Zhai for their feedback during this work. This work is supported in part by National Science Foundation grants CCF-0916583 and CPS-0931931.

References

[1] V. Aslot et al. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 1–10, July 2001.

[2] N. Barrow-Williams, C. Fensch, and S. W. Moore. A Communication Characterisation of SPLASH-2 and PARSEC. In *Proceedings of the International Symposium on Workload Characterization*, pages 86–97, October 2009.

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

[4] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC Performance on Contemporary CMPs. In *Proceedings of the International Symposium on Workload Characterization*, pages 98–107, October 2009.

[5] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques*, pages 72–81, October 2008.

[6] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of the International Symposium on Workload Characterization*, pages 47–56, September 2008.

[7] M. Chaudhuri et al. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*, pages 293–304, September 2012.

[8] 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.

[9] Y. Chen et al. Efficient Shared Cache Management through Sharing-aware Replacement and Streaming-aware Insertion Policy. In *Proceedings of the 23rd International Symposium on Parallel and Distributed Processing*, May 2009.

[10] N. Duong et al. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 45th International Symposium on Microarchitecture*, pages 389–400, December 2012.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] A. Jaleel, M. Mattina, and B. Jacob. Last-level Cache (LLC) Performance of Data Mining Workloads on a CMP - A Case Study of Parallel Bioinformatics Workloads. In *Proceedings of the 12th International Symposium on High-performance Computer Architecture*, pages 88–98, February 2006.

[16] M. T. Kandemir et al. Optimizing Shared Cache Behavior of Chip Multiprocessors. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 505–516, December 2009.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] W. Li et al. Understanding the Memory Performance of Data-Mining Workloads on Small, Medium, and Large-Scale CMPs Using Hardware-Software Co-simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 35–43, April 2007.

[24] J. Lin et al. Understanding the Memory Behavior of Emerging Multi-core Workloads. In *Proceedings of the Eighth International Symposium on Parallel and Distributed Computing*, pages 153–160, June 2009.

[25] 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.

[26] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic Shared Cache Management (PriSM). In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 428–439, June 2012.

[27] 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.

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

[29] V. Mekkat et al. Performance Characterization of Data Mining Benchmarks. In *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*, March 2010.

[30] R. Natarajan et al. Effectiveness Of Compiler-Directed Prefetching on Data Mining Benchmarks. In *Journal of Circuits, Systems and Computers*, **2**(21), April 2012.

[31] B. Panda and S. Balachandran. CSHARP: Coherence and Sharing Aware Cache Replacement Policies for Parallel Applications. In *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing*, pages 252–259, October 2012.

[32] 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.

[33] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 423–432, December 2006.

[34] K. Rajan and R. Govindarajan. Emulating Optimal Replacement with a Shepherd Cache. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 445–454, December 2007.

[35] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 57–68, June 2011.

[36] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 187–198, December 2010.

[37] R. Ubal et al. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*, pages 335–344, September 2012.

[38] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[39] 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.

[40] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 174–183, June 2009.