International Journal of Engineering Sciences Paradigms and Researches (Volume 47, Issue: Special Issue of January 2018)
ISSN (Online): 2319-6564 and Website: www.ijesonline.com

The Effect of Policies for Cache Inclusion on Cache Management Methods

Mr. Gopal Behera[1], Mrs.Pragyan Paramita Panda[2]*
*[1]*Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
gopalbehera@thenalanda.com, pragyanparamita@thenalanda.com*

ABSTRACT

Caches were created to lessen the significant difference in performance between the processor and memory. To further enhance performance, a variety of last-level cache (LLC) management approaches have been developed. A first finding is that non-inclusive caches have been used to create and test the majority of strategies in the literature. Another finding is that a lot of contemporary processors have either inclusive or exclusive practises in their LLCs. Due to their greater effective capacity, exclusive caches are becoming more and more popular as the number of cores rises. Thus, unique multi-core caches are anticipated in the future. These two observations highlight a discrepancy between the evaluation's cache inclusion policy assumptions and the reality of current and possibly future processor implementations. The effectiveness of current cache management techniques for a variety of inclusivity options is quantified in this paper, and it also discusses the question of how sensitive they are to inclusion policies. Finally, a concept LLC design is suggested that includes the features that are most appropriate for exclusive caches. The findings demonstrate that due to their close interaction, modern prefetchers are essential for evaluating replacement strategies, and inclusive caches necessitate a less aggressive prefetching strategy to prevent excessive back-invalidation. The findings also identify the characteristics needed for an exclusive cache's replenishment policy. LLCs must receive reuse data from lower cache levels along with the data block because they lack recency information and it is difficult to determine the memory access that allocated the data. Moreover, the proposed LLC features include keeping global reuse information structures detached from data blocks to prevent losing that information when the data is evicted on hit.

CCS CONCEPTS

Computer systems organization → Parallel architectures;

KEYWORDS

Inclusion policies, cache hierarchy, cache management, cache replacement policies, hardware prefetching

INTRODUCTION

Main memory technology has not kept pace with the rapid improvement of CMOS-based processors, giving rise to the so-called *Memory Wall* [38]. Modern multi-core processors rely on large on-chip caches to mitigate the disparity in access latency to main memory. The last-level cache (LLC), *i.e.*, the level closer to memory and further away from cores, is typically shared among all cores in the chip. When shared, it stores blocks from all cores and is typically sized at about 1-2 megabytes (MB) per core. Increasing the number of cores requires a larger cache to maintain high core performance.

Efficient LLCs should keep data that is being used and going to be used soon. Many cache management techniques have been investigated to bring the data before its use (prefetching) and to keep only the useful data (replacement policies).

There are several ways to manage how data is allocated in the multiple levels of cache depending on whether a lower level (closer to memory) includes data resident in higher levels (closer to cores).

In *inclusive* caches, a data block present in a cache must also be present in all of its corresponding lower levels. The result of this policy is a lower effective cache capacity due to the data replication across cache levels, and the potential performance and energy impact of inclusiveness-induced invalidations.

Non-inclusive caches attempt to reduce the limitations of inclusive accesses by not enforcing inclusivity in higher cache levels. When a data block is accessed, it is still allocated in all cache levels. However, an eviction on a cache does not trigger back invalidations to higher levels. There is still data replication, but there are not inclusiveness-induced invalidations affecting performance and energy.

Exclusive caches go one step further by enforcing that a data block present in a cache cannot be also present in the corresponding higher-level cache. Only data evicted from higher levels is present in the exclusive cache, effectively making the last-level cache into a *victim* cache [23]. Thus, there is no data replication and there cannot be inclusiveness-induced invalidations.

Industry

Modern processors use different types of inclusion policies in each cache level. The most common is to either use an inclusive or an

exclusive policy in the LLC and an inclusive or non-inclusive in the lower levels. Below are a few examples of real processors with the information on the inclusion policy they use. None of which has a non-inclusive LLC.

AMD processors generally use exclusive LLCs and Intel processors, inclusive. The AMD Athlon (Thurderbird architecture) had an exclusive L2 (LLC), while its rival at the time, the Pentium 4 (from Willamette) [14] had inclusive L2 (LLC). Currently, the latest AMD Zen architecture has a (mostly) exclusive L3. Current Intel processors like Sandy Bridge, Ivy Bridge and Skylake have an inclusive L3 and a non-inclusive L2 [15]. The Intel Knights Landing has an L2 (LLC) that is inclusive of the L1D and non-inclusive of L1I [16]. However, Intel recently announced the Skylake-X which has a mostly exclusive LLC (which they call non-inclusive) [1].

The ARM Cortex-A9 can have an (optional) L2 cache (LLC). The core has support to be attached to exclusive L2 caches as long as that is properly configured both in the core and L2 controller sides [2].

The processors in the IBM POWER series had mostly L3 (LLC) exclusive caches. The POWER5 has an L3 exclusive and an L2 inclusive of both L1D and L1I [11]. The POWER6 has an L3 exclusive cache and the POWER7 has an L3 mostly exclusive cache [13]. The IBM zEC12 has an inclusive L3 (on die) and an inclusive L4 (off-die, on-package) [12].

Motivation

Research in replacement policies (RPs) has been focused on non-inclusive and inclusive caches. Table 1 shows a list of relevant RPs and the evaluated inclusion policy. There has been a lower effort in exclusive caches.

	Inclusive	Non-Incl.	Exclusive
RRIP [21]	X		
SDBP [25]		X	
SHiP [36]		X	
GIPPR [22]		X	
MDPP [33]		X	
EAF [28]		X	
Perceptron [34]		X	
KPCR [26]		X	
Hawkeye [18]		X	
Bypass and Insertion [10]			X
CHAR [5]	X		X
ExDRRIP [20]			X

Table 1: State-of-the-art RPs and their inclusion policy.

The number of cores in multicore chips has been increasing in recent years. A rule of thumb for the size of the LLC is 1-2MB per core. High performance cores typically include 256-512KB of private cache. With an inclusive cache, that could be a large amount of capacity wasted on maintaining inclusivity in the LLC. Exclusive caches are a promising option to increase the effective capacity without increasing chip area.

Most RPs in the literature have been designed and evaluated in inclusive and non-inclusive caches. However, that does not mean that they would also work well on exclusive caches since they have different properties. For example, in an exclusive LLC, a block is evicted on hit while many RPs are based on the reuse of a block.

Inclusive policies in RP design are often overlooked or not even mentioned. The motivation of this paper is to determine whether there is a need for designing different cache management techniques depending on the inclusion policy, with a focus on the exclusive caches. For this, we investigate how the existing RPs perform in different inclusion policies.

Contributions

The contributions of this paper are:

- A comprehensive evaluation of multiple cache configurations including multiple RPs and prefetchers (PFs) for all three inclusion policies: inclusive, non-inclusive and exclusive caches.
- A discussion on the results targeting to understand the gaps in the design of cache management to improve performance in the presence of a given inclusion policy.

2 BACKGROUND

One solution to mitigate the processor and memory performance gap was introducing several levels of cache into the memory hierarchy to bring data closer to the processor. To make them more efficient, there has been extensive research on improving memory management, mostly PFs and RPs.

Prefetchers

Prefetching is a technique used to hide memory latency by bringing data that will potentially be needed by the processor to a closer level of the memory hierarchy. There is, however, the risk of polluting the cache. The PF needs to be accurate (bring the data that is going to be requested) and timely (the prefetched data arrived to the cache made the accesses hit). We describe below the PFs used in our experiments.

Stride-based. Code and data structures (e.g. arrays, matrices), are stored sequentially in memory and accessed consecutively. These PFs exploit this spatial locality by prefetching: the next block (stride of 1) like the Next-line [31]; the block plus an offset (stride of "offset" for timeliness) like the Instruction pointer-based stride [3]; the block plus a dynamic offset (dynamic stride) like Best Offset [27].

DRAM-Aware Access Map Pattern Matching. Ishii *et al* [17] proposed a PF to exploit locality in DRAM called DRAM-aware access map pattern matching (DAAMPM). Before the time the prefetch is going to be used, they suggest waiting and reordering prefetch requests to optimize row activation. The prefetches are reordered so that all blocks that need to access the same row are done together. They also claim that many RPs are unaware about which blocks were from a prefetch or by demand. They added a prefetch bit to avoid promoting a prefetch hit.

Kill the Program Counter. There has been little work on studying the interaction between cache RPs and PFs, and their effect in each of the cache levels [17, 29, 37]. Those studies show that the benefit of RPs can be small or negative when combined with a PF. Kim *et al.* proposed a holistic approach to speculatively manage all cache

levels with coordinated PF and RP [26] called kill the program counter (KPC). This approach aims to improve performance and reduce the overall hardware budget necessary for both PF and RP. The PF component (KPCP), is a PF that decides in which level of the hierarchy to prefetch each specific block. They use a signature table to store a compressed history of past L1 misses. The history is used as a signature to index a pattern table to predict the next block. The predicted block plus the previous history generates another signature which is again used. This technique has an initial training phase that sets a confidence value that is increased as PFs are useful. Later, PFs are only triggered if confidence on the prediction is high. The RP component (KPCR), is a low-overhead RP that uses two global hysteresis to predict dead blocks by tracking global reuse behavior. One hysteresis is for demands and, the other, for prefetches.

Replacement Policies

RPs improve the management of cache contents to evict first the blocks that are not likely to be used again. RPs are used at allocation time when a cache set is full: the algorithm chooses which block to evict from the cache to place the new one. They could also hurt performance if they mispredict.

Bélády proposed an optimal cache replacement algorithm assuming knowledge on the future [4]. As a processor does not have such knowledge, there has been plenty of work in cache replacement algorithms. We explain below a few, focusing on the ones we experimented with.

Aging-based. Many RPs are based on the block's reuse, by aging the blocks over time and evicting the older ones [21, 22, 30, 33, 34]. Jaleel *et al.* proposed an exclusive version of RRIP where they keep a bit in L2 to know if the block came from LLC (previous hit) or from memory [20].

Signature-based. Other RPs make use of the program counter as a signature to predict dead blocks [25, 36].

Evicted Address Filter. Seshadri *et al.* proposed evicted address filter (EAF) claiming to address previous efforts deficiencies in preventing both cache pollution and thrashing at the same time [28].

Bypass and Insertion Policies for Exclusive caches. Gaur *et al.* proposed to use the number of trips to LLC (LLC hits) and their use count in the L2 (L2 hits) cache [10]. Blocks with either a high use or trip count will usually be predicted alive. They later extended this work to also accommodate inclusive caches by using the same prediction to hint the LLC with a dead block prediction [5]. When the use or trip count number is low the LLC will mark the block as the next victim. The same hint could be used to bypass in both cases.

Inclusion Policies

A cache level is related to the previous/higher level (if it exists) depending on which data blocks each level contains. A particular cache level can contain exactly all, exactly none or some of the data blocks of the higher level. An *inclusive* cache contains all data blocks of the higher level. An *exclusive* cache does not contain any of the data blocks of the higher level. A *non-inclusive* cache can contain some blocks from the higher level but not necessarily all or none. Inclusion policy trades off ease of implementation of cache coherence with capacity. The choice of inclusion policy for a cache hierarchy is a design decision with many inputs beyond the scope of this work, however as we demonstrate this choice has a large impact on the effectiveness of cache management techniques.

Inclusive. An inclusive cache level contains all data blocks from higher levels plus some other blocks. That is, a data block is replicated in both cache levels.

In a 2-level cache hierarchy, the data block will be placed in both cache levels on an L2 miss. If the block is evicted from the L1 and, later, a request comes (L1 miss), the data may still be in the L2, thus avoiding accessing main memory. On an L1 eviction, only write backs of dirty blocks are required. If the block is clean, there is no need to copy it back to the L2 because it is already there as per the inclusion policy. A potential problem is on an L2 eviction: to preserve inclusivity, if the block was present in L1, it must be evicted too.

In a multi-core system, an inclusive policy simplifies the coherence protocol implementation. A cache wanting to invalidate copies of a block in other caches has to notify the LLC because it has the information of all blocks in all higher level caches. Therefore, it avoids coherence message broadcasts, thus reducing complexity and energy consumption. Also, coherence information (state and caches having a copy) can be encoded with the cache block so the information is available when accessing it, thus cutting latency of potentially having to access separate structures, such as a directory.

One disadvantage is the effective cache size due to data duplication. The effective size of the cache hierarchy is the size of the LLC. For example, in a 2-level cache hierarchy, the effective size is the one from L2 because it contains all contents from L1. The L1 cache only keeps data closer but does not contribute with additional capacity.

Another disadvantage is back invalidations. An eviction from the LLC can generate an invalidation in L1 but, if it was present in L1, the block may be in use. This can be a problem if the RP is not aware of the usage of a block in the higher caches. Jaleel *et al.* confirmed the limited performance of an inclusive cache comes from back invalidations because the LLC RP is unaware of the core presence and recency [19].

A related problem is that an inclusive cache has less flexibility to improve cache management due to the impossibility of bypassing the LLC to maintain inclusivity.

Non-Inclusive. A non-inclusive cache level may or may not contain blocks from higher levels. The data is replicated when there is a miss in a cache level, and the block is allocated in that cache level and all higher ones. For example, in an L1 miss where the block is in none of the caches, the block will be allocated in L2 and L1. The difference with an inclusive cache is that the inclusivity is not enforced. That means, when a block is evicted from a lower level, it does not generate back invalidations to the higher levels. This simplifies the implementation of this type of caches.

The effective cache size is between the size of L2 and the sum of both. The cache hierarchy usage in a case with non-inclusive cache changes depending on the application and RP. Conflict misses will be reduced in the intermediate or LLC. The blocks that are

referenced frequently stay in L1, therefore L2 has space for other blocks.

One disadvantage of non-inclusive caches is coherence. A non-inclusive LLC that needs to evict a block will have to broadcast the invalidation to the higher level caches, because that information is not present in the LLC, unless a separate directory is implemented and then it must access the directory and pay its extra latency. However, there has been work to separate the cache coherence structures (i.e. directory) from the data blocks of the cache. Zhao *et al.* proposed a non-inclusive cache with an inclusive directory to keep the positive features of both inclusive and non-inclusive policies [39].

Exclusive. An exclusive cache does not include any repli-cated block from higher levels, thus increasing the total amount of data blocks that can fit in the whole cache hierarchy.

The exclusive inclusion policy is similar to a victim cache [23]. In a three-level cache hierarchy, the LLC would be the victim cache of a two-level cache hierarchy. Victim caches contain the evicted blocks from the higher levels aiming to reduce conflict misses. This was originally introduced as a fully associative small cache to reduce conflict misses from direct-mapped caches.

However, it incurs higher complexity. In the example of two cache levels (L1 and L2), when a block that is in L1 (and not in L2) is evicted, it will be allocated in L2. When the block is accessed again, it will be invalidated in L2 and allocated in L1. This generates more work to do on an L2 hit. Also, it makes impossible to use the recency of a block to choose which block to replace when the L2 cache is full, as it only contains data that was evicted from L1 and not accessed again since that eviction.

Jouppi and Wilton identified the benefits of exclusive caching and evaluated them [24]. They found that the extra space of not duplicating the data in the two levels of cache and a higher associativity in the LLC was indeed beneficial.

Ten years later, Zheng *et al.* evaluated the performance of exclusive caches with respect to inclusive [40]. They found that exclusive caching is beneficial for most of the benchmarks they tried (SPEC 2000), but especially for smaller lower-level caches. They suggest that exclusive caches are more suitable for server applications and embedded systems.

The main benefits when using exclusive caches are:

- Less conflict misses like with a higher associativity: two memory references that are mapped to the same set can reside one in each level instead of only one.
- Higher hit rate thanks to a higher effective space by avoiding the blocks duplication in different levels. This is especially relevant in caches with more than 3 levels of cache or with large higher level caches.
- Avoids premature evictions from the higher levels by not requiring back-invalidations (compared to inclusive).

The main drawbacks and limitations of exclusive caches are:

- Less design flexibility because the block size of the exclusive cache has to be equal as the other levels.
- More control complexity and power consumption due to the higher data movement of blocks.
- More complex cache coherence protocols and more area required in symmetric multiprocessing (SMP).

One fundamental difference of exclusive caches is that clean lines are moved to the LLC when evicted from the L2. In inclusive and non-inclusive caches, only dirty lines are moved. This significantly increases the traffic generated by L2 evictions and affects data reusability in the LLC. In non-inclusive, the LLC may have already evicted a line before, when it is evicted from the L2. If the line is clean, the L2 would just invalidate/replace the line with no effects in the LLC. In the exclusive cache, conversely, the L2 copies the clean evicted data to the LLC, which allocates it, potentially replacing other data which may be useful in the near future.

Some exclusive cache designs allocate block in an *inclusive* way, i.e., allocate in all levels, for specific types of data, for example for data shared between multiple threads. In this case the reasoning is that if other threads may access the data, having it allocated in the last-level cache avoid the three-way trip to read it from a remote private cache. For this type of exclusive behavior with exception, we use the term non-exclusive. This types of caches are sometimes referred as non-inclusive. In this paper, non-inclusive cache only refers to the definition provided in Section 2.3.2.

3 METHODOLOGY

Benchmarks

We used all the traces from the SPECspeed CPU2017 benchmark suite [32], single-threaded and both integer and floating point. We also used three single threaded server workload traces from Cloud-Suite [9] (data_caching, sat_solver and graph_analytics using the default inputs specified in the paper), and a trace from a machine learning workload "mlpack_cf" [7]. In total, 24 benchmarks.

Multi-core simulations execute a single benchmark per core. We used 20 mixes of SPECspeed CPU2017 benchmarks, a memory intensive selection from many runs, excluding mixes that were too similar.

All traces warm-up until all finish 200 million instructions. For the timing modeling phase, all cores run until all have run at least one billion instructions, also known as *last* [35]. Some cores will run more than one billion instructions but only the first billion will count towards the statistics. This methodology is used so that all cores are running at the same time during all the execution. This is important to model the effects of a shared LLC in a more realistic environment.

Simulator

We used the ChampSim simulator, which is an extended version of the simulator used in both 2^{nd} Data and Cache Replacement Championships [6, 8]. This simulator models a simple multi-core out-of-order. The configuration we used is described in Table 2. The baseline consists of a 3-level cache hierarchy with all levels following a non-inclusive policy. All caches use copy back with write allocate write policies.

Configurations

We evaluated multiple combinations of PFs, RPs and inclusion poli-cies for both single-threaded and multiprogrammed workloads. In

Parameter	Configuration
L1 I-cache (private)	64KB, 64B blocks, 8-way, 8 MSHRs, 1 cycle latency, 64 read/write/prefetch queue size
L1 D-cache (private)	64KB, 64B blocks, 8-way, 8 MSHRs, 4 cycles latency, 64 read/write/prefetch queue size
L2 unified cache (private)	512KB, 64B blocks, 8-way 16 MSHRs, 8 cycles latency, 32 read/write/prefetch queue size non-inclusive
L3 unified cache (shared)	2MB/core, 64B blocks, 16-way 32 MSHRs, 20 cycles latency, 16/core read/write/prefetch queue size non-inclusive
I-TLB (private)	64 entries, 4-way 8 MSHR, 1 cycle latency
D-TLB (private)	64 entries, 4-way 8 MSHR, 1 cycle latency
L2 TLB (shared)	1536 entries, 12-way 16 MSHR, 8 cycle latency
Frequency	4GHz
Page size	4KB
Fetch, decode, retire	4 wide
Execution	6 wide
Load Queue	2 wide
Store Queue	1 wide
DRAM	2 channels (1 DIMM per channel), 8 banks (64MB per bank), 8 ranks (512MB per rank), 4GB per DIMM
DRAM channel width	8
DRAM I/O frequency	800MHz
Branch Predictor	Perceptron
Reorder Buffer size	256
Pipeline depth	5

Table 2: Simulator configuration.

these experiments, we vary the inclusion policy of the LLC and keep the private L2 cache non-inclusive of the L1 cache, except in the case of the inclusive LLC, in which both the L2 and the L3 are inclusive.

There are six evaluated L2 PFs: no PF, ip stride, next line, BOP, DAAMPM and KPCP. There are two evaluated L1 PFs: no PF and next line.

There are five evaluated RPs: LRU, EAF, KPCR, SHiP and DRRIP. These PFs, replacement and inclusion policies are explained in detail in Section 2.

3.4 Performance Measurement

The baseline configuration used as reference in our evaluation is: non-inclusive cache inclusion policy, next-line L1 PF, no L2 PF, and the LRU RP.

For the single-core, we show the speedup of a configuration over the baseline running a particular benchmark by extracting the instructions per cycle (IPC).

For the multi-core simulations, we show the weighted speedup normalized to the baseline configuration. We compute the IPC of each of the cores in the multi-core execution and divide it by their single-core IPC (same cache size as in multi-core). We sum all the IPC to get a weighted speedup and then we normalize it to the baseline configuration. Equation 1 shows the speedup calculation used.

$$Speedup_i = \frac{\sum IPC\ IPC_i}{IPC_{baseline}} \quad (1)$$

4 RESULTS

In this section we present our results. We analyze single- and multi-core experiments of all cache configurations previously described in Section 3.3.

Single-Core Results

Figure 1 shows the geometric mean speedup of multiple cache configurations across all SPECspeed CPU2017 benchmarks. The baseline in this figure is: next_line L1 PF, no L2 PF, LRU RP and non-inclusive. Similarly, Figure 2 shows the same results across three CloudSuite benchmarks, and Figure 3 with MLpack benchmark (machine learning).

The inclusion policy clearly affects the impact of the multiple PFs and RPs. For example, LRU is the least effective RP in combination with any of the evaluated PFs when the cache is inclusive or non-inclusive. In contrast, *LRU is the best RP when the cache is exclusive* regardless of the PF.

Across the board, exclusive caches are inferior to inclusive and non-inclusive except for the case of LRU. When LRU is the RP of choice, exclusive caches are as performant or even marginally better than inclusive and non-inclusive. The exclusive cache results show a high ratio of copy-backs and write-backs in the LLC and an increase of the number of LLC evicts. These evictions may not be effective, given that they are not based on data reuse information but rather on the time when they were evicted from the L2.

Figure 3 shows a larger difference between exclusive and inclusive/non-inclusive, where the exclusive performs better than the baseline compared to the other benchmarks but very low compared to the other inclusion policies.

Prefetcher Impact. The next_line PF and no PF configurations show the same performance for the same RP. This is because the L1 has a next_line PF in these experiments which already intro-duce the same kind of accesses into the L2 that, on miss, generate the same pattern a next_line L2 PF would.

KPCP is the PF with the largest difference in performance for the multiple inclusion policies. For inclusive and non-inclusive caches, it achieves similar speedups to the other PFs. However, those same replacement policies in combination with KPCP actually worsen performance for exclusive caches compared to the baseline.

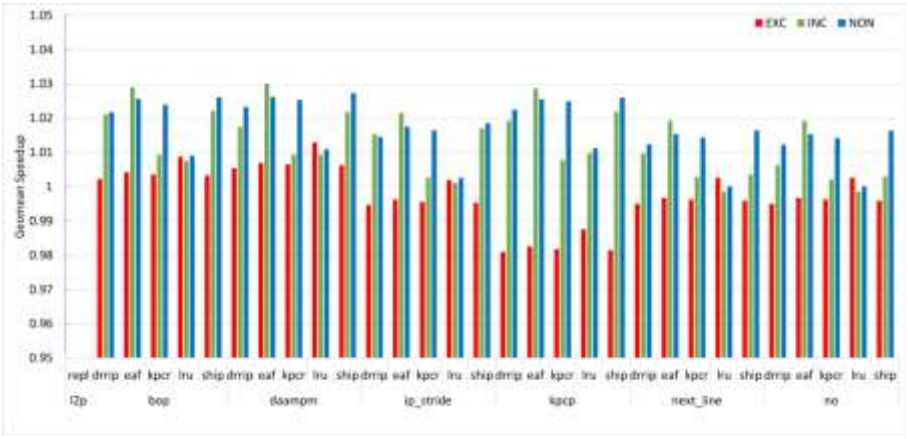

Figure 1: Geomean speedups of SPECspeed CPU2017 benchmarks to compare different configurations of L2 PFs, RPs and cache inclusions. The configurations compared are: L2 PF, RP and cache inclusion, always with a next_line L1 PF. The Y-axis shows the speedup over the baseline configuration: no PFs, LRU RP and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L2 PF (l2p) and RP (repl).

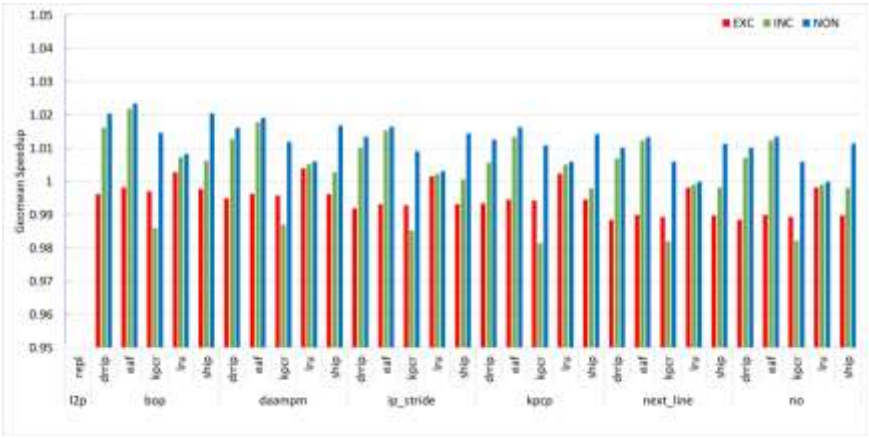

Figure 2: Geomean speedups of CloudSuite benchmarks to compare different configurations of L2 PFs, RPs and cache inclusions. The configurations compared are: L2 PF, RP and cache inclusion, always with a next_line L1 PF. The Y-axis shows the speedup over the baseline configuration: no PFs, LRU RP and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L2 PF (l2p) and RP (repl).

DAAMPM is the best PF among all configurations for SPEC. For the exclusive cache, it is the one that achieves the best results regardless of the RP.

Replacement Policy Impact. There is no clear winner among RPs, but there are a few patterns that indicate that the LRU RP is among the best options to use in general for an exclusive cache independently of the PFs on a single core. That is not the case in the machine learning benchmark where LRU performs like the other RPs.

SHiP, similarly to others, does not work well for exclusive caches. SHiP uses a predictor on placement and updates it using the program counter (PC). PC-based policies such as SHiP rely on the correlation between the PC of a memory access instruction and the reuse behavior of the block accessed by that instruction. However, only evicted blocks are placed into an exclusive LLC. *The PC of the memory access instruction that triggers a block eviction has very little correlation with the reuse of that block.* Thus, PC-based policies only work well with inclusive and non-inclusive caches.

The KPCR RP makes the inclusive cache perform worse than non-inclusive. Other RPs have a closer behavior for non-/inclusive policies.

Furthermore, exclusive caches do not show much sensitivity to the RP. In contrast, inclusive and non-inclusive caches are more variable. The speedup percentage differences are within 5% because several benchmarks are insensitive to cache behavior while others see a larger impact. The benchmarks that show more sensitivity are: gcc, lbm, mcf, cactuBSSN, fotonik3d, satSolver and MLpack.

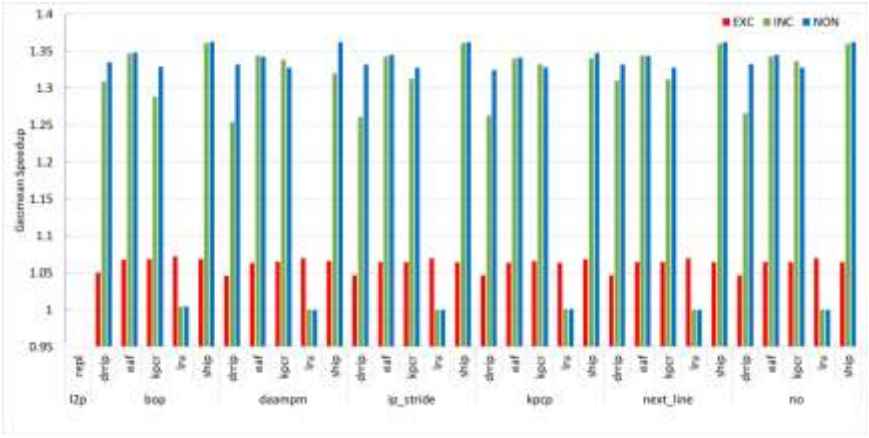

Figure 3: Geomean speedups of the Machine Learning "mlpack" benchmark to compare different configurations of L2 PFs, RPs and cache inclusions. The configurations compared are: L2 PF, RP and cache inclusion, always with a next_line L1 PF. The Y-axis shows the speedup over the baseline configuration: no PFs, LRU RP and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L2 PF (l2p) and RP (repl).

Multi-Core Results

Figure 4 shows the weighted speedup of all cache configurations in multi-core. The baseline in the figure is: next_line L1 PF, no L2 PF, LRU RP and all non-inclusive caches. This shows the speedup of the multi-core experiments compared to the single-threaded case. Similarly, Figure 5 shows the same results for CloudSuite (three mixes), and Figure 6 for MLpack (four instances).

In multi-core, the performance difference in an exclusive cache is more variable across the RPs. LRU does not work as efficiently in multi-thread as it did in single-threaded runs; it is actually the worst one, as expected.

Figure 5 shows the only one case where the exclusive is as good or better than the other inclusion policies, even though it is a small difference. And, although LRU is not the worse RP, it is still better than in non-/inclusive.

Figure 6 shows that the difference between exclusive and inclusive/non-inclusive is smaller than in one core. Overall, in all multi-core plots the exclusive cache is more competitive than for single-core. This is encouraging since one of the reasons to use exclusive caches is the increase of the number of cores per chip.

Prefetcher Impact. Figure 4 shows that ip_stride performs generally better across inclusion policies.

The PF seems to be determining for exclusive and non-inclusive LLCs, as different RPs using the same PF do not show large variations. The inclusive LLC, however, is more sensitive to the RP. This makes sense because replacement in an inclusive LLC may cause invalidations of useful data in upper-level caches and, therefore, getting replacements right is as important or more than precise prefetching.

Replacement Policy Impact. Figure 4 shows that the best RPs for exclusive caches are EAF, SHiP and KPCR. These RPs are different from the others in this study because they store information about accesses to a block even after having evicted the block from the LLC. In an exclusive cache, a block will be evicted on hit and any bookeeping information stored with the block is lost. These policies, however, maintain reuse information in a separate hardware structure which is kept even on a block miss or eviction. For example, the EAF RP stores the evicted memory addresses and, on that address miss, it inserts at the MRU position to protect it. SHiP has a table of counters indexed by a partial tag. The counters are updated on hit and eviction. Both things happen on the cache either the block is removed or not in the LLC. Even though the program counter has no correlation with the block, SHiP is still able to get the general trend on dead blocks in multi-core. As demonstrated with KPCR, even a global up/down counter can be a good first-cut dead-block predictor. SHiP can be seen as roughly tracking the tendency of blocks to be dead within a given program phase despite the lack of correlation between specific PCs and block accesses. Similarly, KPCR also updates a global counter on hit and miss.

These techniques employ these separate hardware structures to reduce the area and power impact of the RP. Collaterally, these mechanisms help exclusive caches because blocks are invalidated on hit and therefore lose reuse information that is stored with the block. By keeping these separate hardware structures, RPs in an exclusive cache can prioritize blocks that have been evicted or accessed recently even if those blocks are not in the cache.

SHiP and KPCR are the best RPs for inclusive caches. Some benchmarks in the multiprogram mixes are cache resident in private caches. With an inclusive cache, the data of those threads will eventually be evicted given that their blocks are not promoted in the LLC by hits in the private caches. If that thread's data is inserted in the LLC with low priority it will be evicted faster. This happens with DRRIP, which inserts with a maximum age of two. SHiP and KPCR use a similar strategy but outperform DRRIP across inclusive configurations. This is because their improvements may mitigate the described effects for inclusive caches. SHiP can insert into a lower age position (higher priority) when it predicts the block is live and help keep the private-cache-resident data in the LLC for

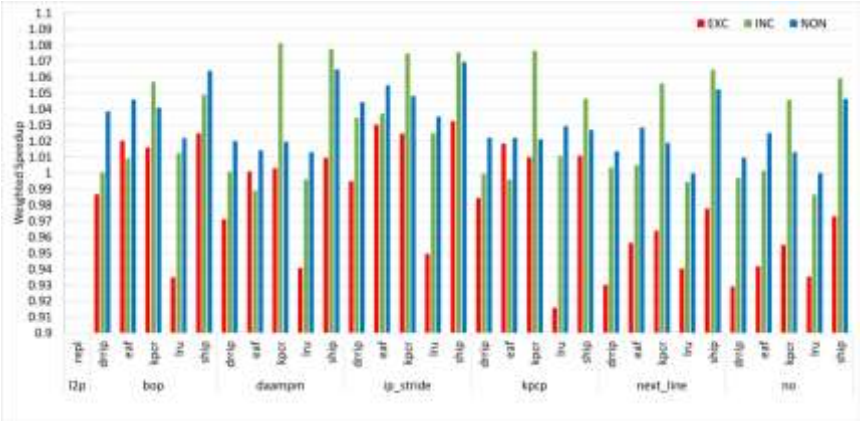

Figure 4: Weighted speedups of 4-core multiprogrammed SPECspeed CPU2017 benchmark mixes to compare different configurations of PFs, RPs and cache inclusions. The configurations compared are: L2 PF, RP and cache inclusion. The Y-axis shows the speedup over the baseline configuration: no PFs, LRU RP and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L2 PF (l2p) and RP (repl).

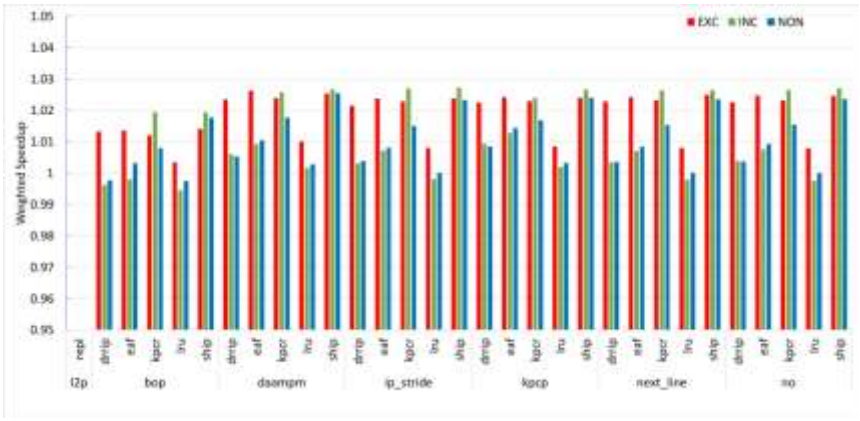

Figure 5: Geomean speedups of CloudSuite benchmarks to compare different configurations of L2 PFs, RPs and cache inclusions. The configurations compared are: L2 PF, RP and cache inclusion, always with a next_line L1 PF. The Y-axis shows the speedup over the baseline configuration: no PFs, LRU RP and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L2 PF (l2p) and RP (repl).

longer. An aggressive PF of another thread can also interfere with those cache resident threads by allocating prefetched blocks in the inclusive LLC and evict useful data. KPCR keeps a separate hysteresis counter for prefetches and demand misses. If prefetches are identified as dead, they would be inserted with an age of three (the lowest priority value), therefore not displacing useful data.

L1 Prefetching

We ran the same configurations as in previous sections but without the L1 PF. Contrary to expectations, most configurations without L1 PF perform better. Figure 7 shows that the speedups without L1 PF are generally higher than those with a next_line L1 PF. We observed the same behavior on the other benchmarks and in multi-core.

The only configuration that performs worse without L1 PF is when there is no L2 PF either. In all other cases it appears that L1 PFs interfere with the L2 prefetches and worsen performance. We generated the same results for multi-core without L1 PF and it shows the same performance improvement over next_line L1 PF. *There is a need to design L1 and L2 PFs so they cooperate instead of interfere.*

Case Studies on Inclusion Impact

We looked at the statistics of the most sensitive benchmarks to inclusion and cache management techniques. We observed that some benchmarks such as satSolver and cactuBSSN did not work well for inclusive caches for both SHiP and KPCR. satSolver shows a slowdown of 0.92 compared to either exclusive and non-inclusive, and cactuBSSN a 0.77 slowdown.

In satSolver, the L2 evictions due to the inclusivity for KPCR are between 34 and 27 million (M) for most PF combinations, while for

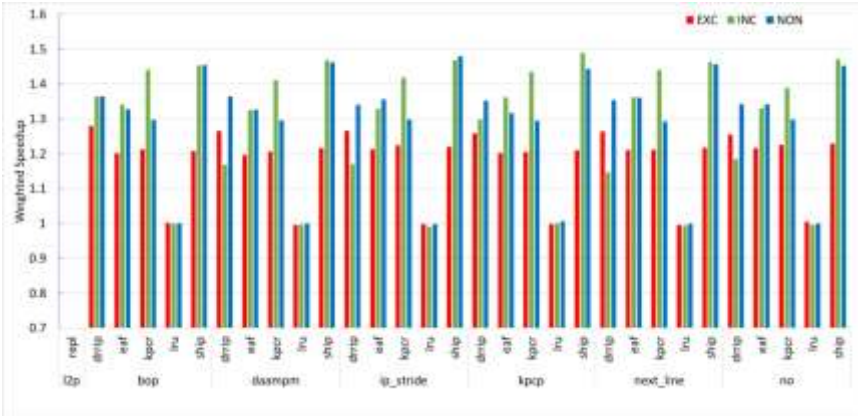

Figure 6: Geomean speedups of the Machine Learning "mlpack" benchmark to compare different configurations of L2 PFs, RPs and cache inclusions. The configurations compared are: L2 PF, RP and cache inclusion, always with a next_line L1 PF. The Y-axis shows the speedup over the baseline configuration: no PFs, LRU RP and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L2 PF (l2p) and RP (repl).

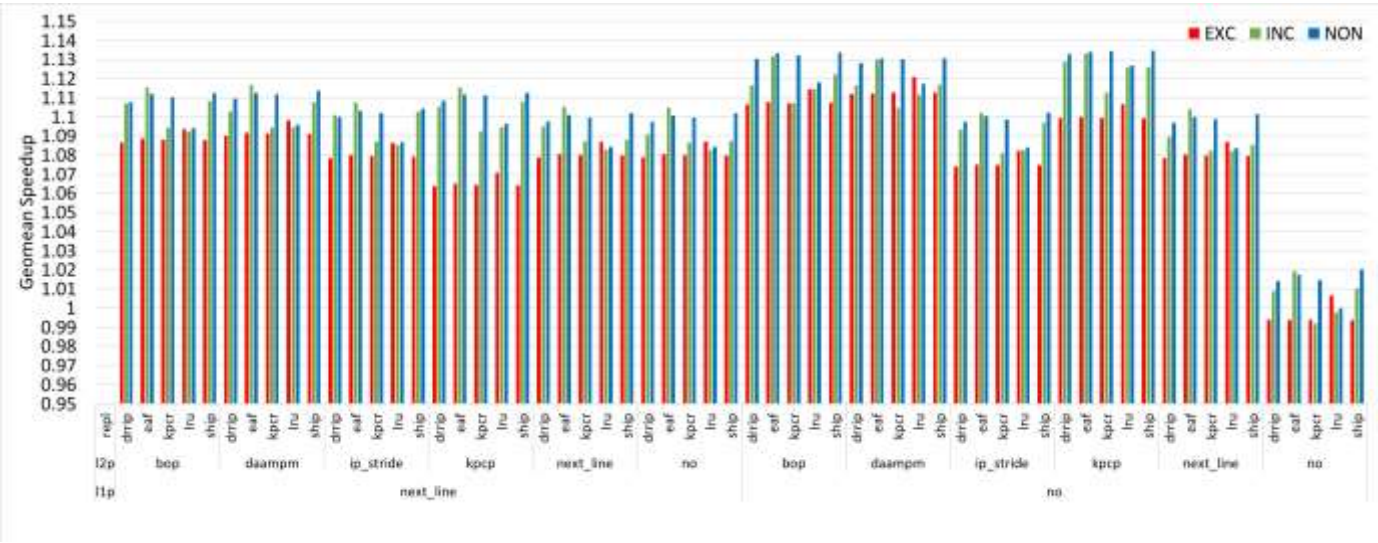

Figure 7: Geomean speedups to compare different configurations of PFs, RPs and cache inclusions including with and without L1 PF. The configurations compared are: L1 PF, L2 PF, RP and cache inclusion. The Y-axis shows the speedup over the baseline configuration: no PFs, LRU RP and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: RP (repl), L2 PF (l2p) and L1 PF (l1p).

the rest of the RPs it is under 1M. In SHiP, the difference is also clear but smaller than KPCR being between 24-20M versus 1M. The L1 evictions due to inclusivity show the same trend, with KPCR and SHiP clearly higher than the rest. Our hypothesis is that these RPs are so effective that they predict a block dead in LLC so fast that the higher level caches might still be using those blocks. For some cases this works well when the block is indeed dead.

In cactuBSSN, the L2 evictions due to the inclusivity for SHiP are between 12M and 10M while others are below 1M. In this case, KPCR is around 2.5-1.5M, a smaller difference than SHiP. In this case, the inclusion policy degrades performance (0.7 slowdown). The ones that show more difference in evictions due to inclusivity and performance are the ones with a combination of PFs of: no PF and next_line (e.g. no L1 PF + next_line L2 PF, next_line or none in both, or another order). Our hypothesis is that a not so smart (or lack of) PF is generating back invalidations plus interfering with the SHiP replacement policy, which is not core aware and can generate more back invalidations).

Cache inclusion techniques should be designed taking into account the inclusion policy. This section indicates that the PF should also be designed accordingly not to interfere.

Summary

The single-core simulations showed BOP and DAAMPM PFs worked generally better for any inclusion policy. The best RPs for inclusive and non-inclusive are EAF and SHiP, in contrast to exclusive with LRU.

The multi-core simulations showed, for non-inclusive, that the best PF depends on the cache configuration, but in general, ip_stride works well. The RP that works better in all cases is SHiP. That is very clear in non-inclusive caches. However, in inclusive caches KPCR also works well, and in exclusive both SHiP and KPCR work well in addition to EAF.

5 CONCLUSIONS

Cache management techniques have been mostly designed and evaluated in the context of non-inclusive LLCs. However, many modern processors implement their LLC with either an inclusive or exclusive policy. In this paper we explored the design space of cache management techniques in different cache configurations, with a focus on the cache inclusion policy. We implemented an inclusive and an exclusive policy on top of a simulator that had a non-inclusive policy. We evaluated different prefetchers (PFs), replacement policies (RPs) and number of cores for each inclusion policy.

The results demonstrate that a cache management technique that performs well in one inclusion policy does not necessarily work well for another inclusion policy. Different inclusion policies have a different behavior which affects how different cache management techniques impact performance. Inclusive caches are more sensitive to the RP in use while non-inclusive and exclusive caches are largely influenced by PF choice. This makes sense as replacements may trigger invalidations of potentially-useful data.

RPs that keep global reuse information in separate structures perform better for exclusive caches. Exclusive caches invalidate cache blocks on hit which renders reuse information contained along cache blocks irrelevant. The use of the program counter is also useless because there is no correlation between the program counter and the block that is placed in the LLC on L2 evictions.

Exclusive caches have a larger capacity than inclusive and non-inclusive. However, our results show that exclusive caches perform worse in most cases. This shows a need for PFs and RPs tailored for exclusive caches to unleash its performance potential. The multi-core results are encouraging since they highlight the higher effective capacity.

Based on all this data, we propose some features that an exclusive LLC design should include. First, to overcome the lack of recency information, forward such information together with the data block on eviction, such as ExRRIP and Bypass and Insertion RPs. Second, to solve the information loss on a hit when the information lives with the block, an approach like EAF helps by keeping a separate structure that tracks the evicted addresses, so that information outlives the block and helps prioritizing blocks on future replacements.

REFERENCES

[1] AnandTech. 2017. Skylake-X's New L3 Cache Architecture.https://www.anandtech.com/show/11464/intel-announces-skylakex-\ bringing-18core-hcc-silicon-to-consumers-for-1999/3. (2017). Accessed:2018-04-03.
[2] Arm. 2012. Cortex-A9 Technical Reference Manual. http://infocenter.arm.com/ help/topic/com.arm.doc.ddi0388i/DDI0388I_cortex_a9_r4p1_trm.pdf. (2012). Ac- cessed: 2017-04-30.
[3] Jean-Loup Baer and Tien-Fu Chen. 1991. An Effective On-chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC'91)*. ACM, New York, NY, USA, 176–186.
[4] Laszlo A. Bélády. 1966. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal* 5, 2 (1966), 78–101.
[5] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. 2012. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. IEEE, 293–304.
[6] CRC 2017. The 2nd Cache Replacement Championship. http://crc2.ece.tamu.edu. (2017). Accessed: 2017-04-30.
[7] Ryan R Curtin, James R Cline, Neil P Slagle, William B March, Parikshit Ram, Nishant A Mehta, and Alexander G Gray. 2013. MLPACK: A Scalable C++ Machine Learning Library. *Journal of Machine Learning Research* 14 (2013), 801–805.
[8] DPC 2015. The 2nd Data Prefetching Championship. http://comparch-conf. gatech.edu/dpc2. (2015). Accessed: 2017-04-22.
[9] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: a Study of Emerging Scale-out Workloads on Modern Hardware. In *ACM SIGPLAN Notices*, Vol. 47. 37–48.
[10] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. 2011. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *ACM SIGARCH Computer Architecture News*, Vol. 39. 81–92.
[11] IBM. 2012. IBM POWER5 Architecture. https://www.ibm.com/developerworks/ community/wikis/home?lang=en#!/wiki/Power%20Systems/page/POWER5% 20Architecture. (2012). Accessed: 2017-04-30.
[12] IBM. 2013. IBM zEC12. https://www.hotchips.org/wp-content/ uploads/hc_archives/hc25/HC25.20-Processors1-epub/HC25.26. 220-zEC12-Processor-Sonnelitter-IBM-v5.pdf. (2013). Accessed: 2017-04-30.
[13] IBM. 2016. IBM POWER7 Architecture. https://www.cs.rice.edu/~johnmc/ comp522/lecture-notes/COMP522-2016-Lecture7-Power7.pdf. (2016). Accessed: 2017-04-30.
[14] Intel. 2000. Intel Pentium 4 Willamette. http://ark.intel.com/products/27426/ Intel-Pentium-4-Processor-1_70-GHz-256K-Cache-400-MHz-FSB. (2000). Accessed: 2017-04-30.
[15] Intel. 2016. Intel 64 and IA-32 Architectures Optimization Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/ 64-ia-32-architectures-optimization-manual.pdf. (2016). Accessed: 2017-04-30.
[16] Intel. 2016. Knights Landing Architecture. http://pages.cs.wisc.edu/~david/ courses/cs758/Fall2016/handouts/restricted/Knights-landing.pdf. (2016). Accessed: 2017-04-30.
[17] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2012. Unified Memory Optimizing Architecture: Memory Subsystem Control with a Unified Predictor. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. 267–278.
[18] Akanksha Jain and Calvin Lin. 2016. Back to the future: leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 78–89.
[19] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C Steely Jr, and Joel Emer. 2010. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*. 151–162.
[20] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C Steely, and Joel Emer. 2015. High Performing Cache Hierarchies for Server Workloads: Relaxing Inclusion to Capture the Latency Benefits of Exclusive caches. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. 343–353.
[21] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. 60–71.

[22] Daniel A Jiménez. 2013. Insertion and Promotion for Tree-based PseudoLRU Last-Level Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. 284–296.

[23] Norman P Jouppi. 1990. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. 364–373.

[24] Norman P Jouppi and Steven JE Wilton. 1994. Tradeoffs in two-level on-chip caching. In *Proceedings the 21st Annual International Symposium on Computer Architecture (ISCA'94)*. 34–45.

[25] Samira M Khan, Yingying Tian, and Daniel A Jiménez. 2010. Dead Block Replacement and Bypass with a Sampling Predictor. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*. 175–186.

[26] Jinchun Kim, Elvira Teran, Paul V Gratz, Daniel A Jiménez, Seth H Pugsley, and Chris Wilkerson. 2017. Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XXII)*. 737–749.

[27] Pierre Michaud. 2016. Best-offset Hardware Prefetching. In *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture (HPCA-22)*. 469–480.

[28] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd C Mowry. 2012. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 355–366.

[29] Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2015. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Transactions on Architecture and Code Optimization* 11, 4 (2015), 51.

[30] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. 1999. EELRU: Simple and Effective Adaptive Page Replacement. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 27. 122–133.

[31] Alan Jay Smith. 1978. Sequential Program Prefetching in Memory Hierarchies. *Computer* 11, 12 (1978), 7–21.

[32] SPEC. 2017. ISPEC CPU2017 benchmark suite. https://www.spec.org/cpu2017. (2017). Accessed: 2017-11-22.

[33] Elvira Teran, Yingying Tian, Zhe Wang, Daniel A Jiménez, et al. 2016. Minimal Disturbance Placement and Promotion. In *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture (HPCA-22)*. 201–211.

[34] Elvira Teran, Zhe Wang, and Daniel A Jiménez. 2016. Perceptron Learning for Reuse Prediction. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. 1–12.

[35] Javier Vera, Francisco J Cazorla, Alex Pajuelo, Oliverio J Santana, Enrique Fernandez, and Mateo Valero. 2007. Fame: Fairly measuring multithreaded architectures. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. 305–316.

[36] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. 2011. SHiP: Signature-based Hit Predictor for High Performance Caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. 430–441.

[37] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. 2011. PACMan: prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. 442–453.

[38] William A Wulf and Sally A McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (1995), 20–24.

[39] Li Zhao, Ravi Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. 2010. NCID: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. In *Proceedings of the 7th ACM international conference on Computing Frontiers (CF'10)*. 121–130.

[40] Ying Zheng, Brian T. Davis, and Matthew Jordan. 2004. Performance Evaluation of Exclusive Cache Hierarchies. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04)*. 89–96.