ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

For Inclusive Last Level Caches, Adaptive Cache Bypassing

Mr.Rajesh Kumar Pati[1], Mr.Narottam Sahu[2]*
*[1]*Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
rajeshkumar@thenalanda.com, narottam@thenalanda.com*

***Abstract*— The inclusion property, bypassing, and replacement cache hierarchy designs all have a major performance impact. Cache bypassing is an efficient method to improve the performance of the last level cache (LLC), according to recent research on high performance caches. The inclusion property is essentially broken by bypassing, hence this technique cannot be used to improve the widely used inclusive cache hierarchy. The problem of permitting cache bypassing for inclusive caches is addressed in this study. To an LLC, we add a bypass buffer. The tags of bypassed cache lines are kept in this bypass buffer while they skip the LLC. In order to maintain the inclusion property, when a tag is removed from the bypass buffer, the matching cache lines in upper level caches are invalidated. Our main finding is that a bypassed line should have a brief lifetime in upper level caches and is most likely dead when its tag is removed from the bypass buffer, assuming a well-designed bypassing mechanism. In order to keep the inclusion property and benefit from the majority of the performance advantages of bypassing, a minimal bypass buffer is sufficient. The bypass buffer also makes bypassing algorithms easier by giving the use data for lines that were bypassed. We demonstrate that the performance of a top-performing cache bypassing approach, which was first developed for non-inclusive caches, is comparable for inclusive caches that have our bypass buffer. When compared to the original design, the bypass buffer's utilisation data dramatically lowers hardware implementation costs. .**

Keywords: Last level cache; cache bypassing; cache replacement policy; inclusion property

I. INTRODUCTION

With increasing working sets of applications, the performance of last level caches (LLCs) is critical to the overall computer system performance. Cache management contains two key components: (1) a replacement policy, which decides the victim block if a block needs to be replaced, and (2) an allocation policy which decides whether an incoming block should be allocated in the cache. A good cache replacement policy improves cache performance by selecting the least likely to be reused block as the victim and has been studied extensively [1][3][5][9][16][17][20][23]. A good cache allocation policy chooses to bypass a block to upper levels if it is predicted to be less useful than the blocks currently in the cache [7].

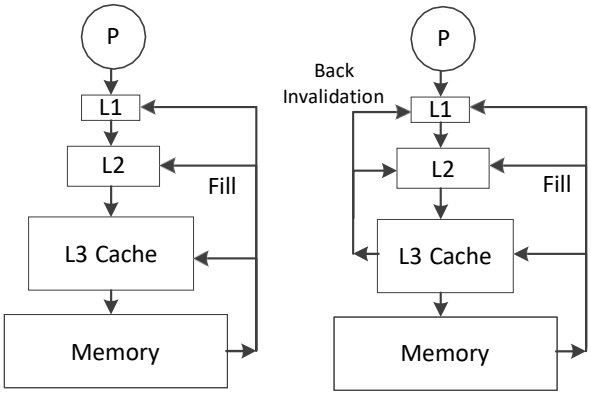
Another key design decision in cache hierarchies is the inclusion property between an LLC and upper level caches. Inclusion simplifies the hardware to support cachecoherence. It enables the LLC to act like a snoop filter because a data block is guaranteed to be absent in upper levels if not found in the LLC. As a result, inclusive caches have been widely used. With inclusive caches, the allocation policy is reduced to allocate all incoming requests by default in an LLC. This is the reason why previous bypassing algorithms [4][5][6][7][12][15][21][22][25] only work with non-inclusive/exclusive LLCs.

Figure 1 shows various flavors of memory hierarchy organization possible with strict/flexible allocation policies combined with inclusive/non-inclusive LLCs. Figure 1a shows a non-inclusive

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

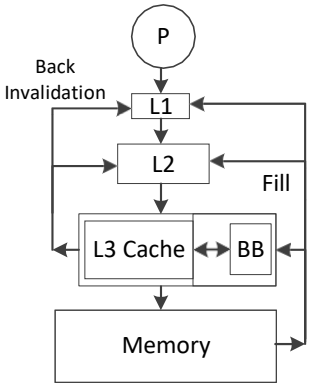
cache where all the incoming cache blocks from memory are allocated in all three levels of caches. The LLC is non-inclusive and therefore the evictions from LLC are silent i.e. they do not try to invalidate the evicted data blocks from upper levels. On the other hand, an inclusive LLC (shown in Figure 1b) will force an eviction of the corresponding data block(s) from L1 and L2 cache when a cache block is evicted from LLC. This event is also referred as *back invalidation*. Applying a selective allocation policy/bypassing is straight forward on a non-inclusive LLC because the selected incoming blocks from memory can be filled into L1 cache and L2 cache only (as shown in Figure 1c) and it does not violate the non- inclusion property. The inclusive LLC is strict about filling each incoming block from memory. This causes the inclusive cache hierarchy to be incapable of using cache bypassing or any selective allocation policy.

In this work, we propose a solution to enabling cache bypassing for inclusive LLCs. We introduce a new structure in an LLC, called a bypass buffer (BB), which keeps bypassed blocks to support the inclusion property (as shown in Figure 1d). Therefore, the last level cache hierarchy consists of an LLC and a bypass buffer. The bypass buffer keeps tags of the data blocks which are predicted to be less important than data present in the LLC. In this manner, the working set present in the LLC is not evicted to make room for less useful data. When a block is evicted from the LLC or BB, it invalidates the data copies present in upper level caches to maintain inclusion property. Our insight is that with a good bypassing algorithm, bypassed blocks should have a short lifetime in upper level caches. Therefore, a small BB is sufficient to ascertain that when a block is evicted from the BB, it is highly likely that its data copies in L1/L2 caches are either dead or already evicted. Furthermore, we show that our proposed BB provides an efficient way to collect the usage information of bypassed blocks, which can be used to simplify and facilitate the design of bypassing algorithms.



(a) Non-inclusive LLC (b) Inclusive LLC

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

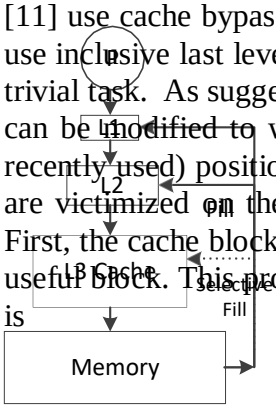


Fourth, we evaluate the performance of inclusive LLC with bypass-buffer in various cache configurations and different scenarios to conclude that bypass buffer can provide robust and effective solution to employing cache bypassing algorithms to inclusive LLCs.

The remainder of the paper is organized as follows. Section II motivates our approach and dissects the lifetime of bypassed blocks to motivate our low cost solution to enabling cache bypassing for inclusive LLCs. Section III details our design of adaptive bypassing for inclusive LLCs. Section IV presents the experimental methodology and Section V discusses the experimental results. We discuss related work in Section VI. Section VII concludes the paper.

II. MOTIVATION

Bypassing has been shown to be high performing by previous research. In particular, two of the three topperformers in the 1st JILP Cache Replacement Competition [11] use cache bypassing. On the other hand, many industrial designs, including Intel Core i7 [26], use inclusive last level caches which makes employing the cache bypassing on these designs a non-trivial task. As suggested in a recent work on cache bypassing algorithm [18], a bypassing algorithm can be modified to work with inclusive caches by inserting the bypassed block at the LRU (least recently used) position. In this manner, the bypassing candidates chosen by the bypassing algorithm are victimized on the next miss to the cache set. There are two main drawbacks to this approach. First, the cache blocks still need to be placed in the cache set thereby replacing onepotentially more useful block. This problem is more likely to manifest itself in a cache where the set associativity is relatively low. Second, this approach is vulnerable to apathological scenario where many consecutive accesses are



(c) Non-inclusive LLC (d) Inclusive LLC with bypass-with bypass buffer for selective bypassing

Figure 1: Memory hierarchy organization for (a) a non- inclusive LLC (b) an inclusive LLC (c) a non-inclusive LLC with bypass (selective fill of L3 cache) (d) an inclusive LLC with a bypass buffer to support cache bypassing

The key contributions of this paper include:

First, we make an important observation on the lifetime of bypassed blocks to motivate our low overhead BB idea.

Second, we show that our proposed BB facilitates the design of bypassing algorithms and it significantly reduces the hardware cost of the DSB algorithm [6], a top performing cache-bypassing

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

algorithm.

Third, we evaluate our proposed solution and show that our bypass-enabled LLC achieves up to 42.0% and an average of 9.4% performance improvement over an inclusive 2MB LLC with the least-recently-used (LRU) replacement policy. Compared to a recently proposed high performing replacement policy, DRRIP [9], our proposed approach outperforms it for both single-core systems, by upto 11.3% and 2.5% on average, and 4-core systems, by up to 14.0% and 1.3% on average.

mapped to a cache set. Due to the prediction of no future reuse, they will compete for the LRU position. As a result, the lifetime of these blocks is short, which causes the victimization of the same data blocks from upper levels. This will degrade performance of inclusive LLCs. These potential performance hazards are inherently present in any cache replacement algorithm and we show in Section V-D that a benchmark (*sphinx*) in our experiments indeed severely suffers from this problem. Therefore, we propose to combine the bypassing algorithm with inclusive caches without converting it to a replacement algorithm. The key reason is that a bypassing algorithm is higher performing than a replacement algorithm since it does not have to insert the data in a cache level if there is no future reuse at that level of cache. Now we present our motivation behind our bypass buffer idea.

We first make an important observation on cache bypassing algorithms. The goal of cache bypassing is to bypass blocks that have fewer reuses than those currently in the cache. Therefore, for a well-designed bypassing algorithm, a block, which is bypassed from the LLC and allocated in the upper levels of caches (i.e., L1/L2 caches),

should not be re-accessed after it is replaced from the L1/L2 caches. Otherwise, such re-accesses would become reuses of the block in the LLC, conflicting with the choice of bypassing. To quantify our observation, we collect the lifetime information of the bypassed blocks, which are chosen with the DSB bypassing algorithm [6], and report the lifetime histogram of selected benchmarks in Figure 2. Here, the lifetime is measured as the number of LLC misses while a cache block was live in the L1 cache (i.e., the number of LLC misses between the time when the bypassed block is allocated in the L1 cache and the time of its last touch before being evicted). From Figure 2, we can see that bypassed blocks quickly become dead in the L1 cache. For example, for benchmark *art*, 75% of its bypassed cache blocks have a short lifetime of between 3 to 4 LLC misses in the L1 cache and 96.4% of its bypassed blocks are dead after 8 LLC misses. On average of all the benchmarks in our study (see Section IV for methodology), 94.3% of the bypassed blocks are dead after 8 LLC misses. We also collected the lifetime information of the bypassed blocks in the L2 cache and it exhibits very similar trends.

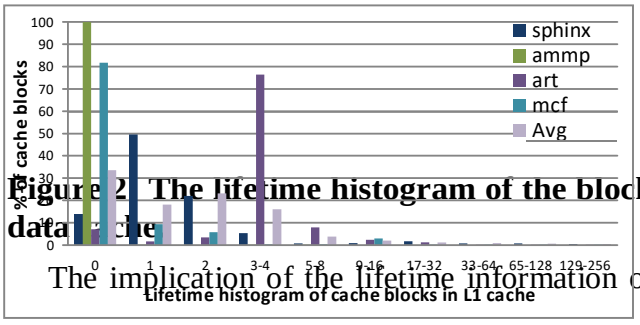


Figure 2. The lifetime histogram of the blocks, which are bypassed from the LLC, in the L1 data cache

The implication of the lifetime information on *inclusive* LLCs is that those blocks, which would

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

have been bypassed otherwise, are essentially useless and are allocated in LLCs only for the inclusion purpose. Note that even marking those blocks as early victims to evict in LLCs may still replace more useful data, thereby not as effective as bypassing. In the next section, we leverage the short lifetime of bypassed blocks to design our low cost solution to enable cache bypassing for inclusive LLCs.

III. ADAPTIVE CACHE BYPASSING FOR INCLUSIVE LLCS

To enable cache bypassing for inclusive LLCs, we propose a bypass buffer (BB). The bypassed blocks are kept in the BB rather than replacing victims in an LLC. When a block is evicted from the BB, it invalidates the copies of the same data in upper level caches to ensure the inclusion property. The lifetime information presented in Section II shows that the bypassed blocks become dead quickly in L1/L2 caches. Therefore, a small BB is sufficient to reap the performance benefit of bypassing while maintaining the inclusion property.

Next, we present our design to incorporate a bypassing algorithm within an inclusive cache hierarchy. We use the winning algorithm from CRC [11], Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing (DSB) [6]. A key feature of DSB is bypassing the LLC adaptively, which is shown as the highest contributing factor to the performance gains. Then, we show how theproposed BB can be leveraged to reduce the hardware costof the DSB algorithm. Our design is based on an inclusive LLC (L3 cache) and a non-inclusive L2 and L1 caches shown in Figure 1d, as used in Intel Core architectures [26].

A. *Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing (DSB)*

In this section we briefly present the DSB algorithm and summarize the key ideas [6].

1. A Segmented LRU (SLRU) replacement algorithm [14], which was originally proposed for cache management for disk systems. Random promotion and aging policies are proposed to enhance performance.
2. An adaptive bypassing policy, which randomly bypasses cache blocks based on a probability. This probability is increased or decreased based on whether bypassing is effective or not. The effectiveness of bypassing is determined by tracking whether a bypassed block is reused before the replacement victim. To do so, each cache set is augmented with an additional tag and a competitor pointer. In the case of a bypass, the additional tag field keeps the tag of a bypassed block and the competitor pointer points to the replacement victim, which would have been evicted without bypassing. If the competitor is accessed before the bypassed one, bypassing is determined as effective. If the bypassed tag is accessed before the competitor, bypassing is determined to be ineffective. DSB algorithm invalidates a *bypass block – competitor pointer* pair when there is a fill at the location pointed by competitor pointer. To assess the impact of bypassing when a no- bypassing decision is made, some newly allocated blocks are randomly selected for 'virtual bypassing'. In this case, the additional tag keeps the tag of the replacement victim and the competitor pointer keeps the position of the newly incoming block. If the replacement victim is re-accessed earlier than the incoming block, it means that bypass is effective.
3. Set sampling, in which a few sample sets maintain auxiliary tag directory (ATD) [20] to exercise two dueling policies and a saturating counter decides which policy is applied to the cache.

B. *Bypass-Buffer Enabled Inclusive DSB*

With a BB, we only need to make the following small changes to support inclusion.

If the bypassing algorithm decides to bypass a requested cache block, it is allocated in the BB instead ofthe LLC and forwarded to upper level caches. If the BB is

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

full, a victim is selected and the data copies of the victim are invalidated in upper level caches.

L2 cache misses are serviced with both the BB and the LLC. A hit in the BB provides the data to the L2 cache and the cache block is de-allocated from the BB and filled in the LLC.

C. *Data-less Bypass Buffers*

To reduce the hardware overhead of a BB, we propose to not include payload data in BB entries. A data-less BB is feasible as the tags are sufficient to maintain the inclusion property. Since the bypassed cache blocks become dead in upper level caches very soon, a hit in the BB should be very rare. Therefore, a data-less BB does not incur performance penalties. In a case when there is a miss in the LLC and hit in BB, it is treated like a miss and the data is brought in from memory. Considering multi-processor design, the BB entries also keep coherence information along with tags similar to the LLC tag store. Assuming a MESI-like coherence protocol, a data-less BB works without any significant modifications. For snoop requests that do not need to respond with data, the data-less BB acts exactly the same as the LLC. In the case for a snoop request asking for data which hits in the BB with the M state, the upper cache levels are searched to find the most recent copy of the data.

D. *Efficient Tracking Using Bypass Buffers*

As discussed in Section III-A, the DSB algorithm needs to track the effectiveness of bypassing and does so by adding additional tags and pointers in each cache set. This incurs relatively high hardware overhead. We propose to leverage the BB to reduce such bookkeeping cost by adding a competitor pointer in each BB entry. Since the number of BB entries is much smaller than the number of sets in the LLC, the overall storage requirement for the DSB algorithm can be significantly reduced.

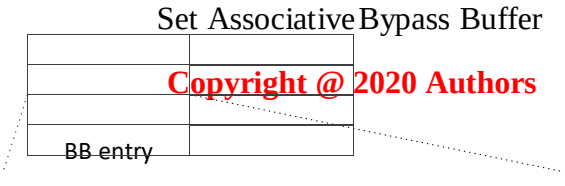
As the tracking information is no longer stored in each cache set, we make the following modifications to the original DSB algorithm:

1. For each bypassed block, its competitor pointer points to the replacement victim, which would have been replaced without bypassing.
2. When a block is chosen to be virtually bypassed (i.e., it is still allocated in the LLC but selected to assess the impact of bypassing), a BB-entry is allocated for the replaced block and its competitor pointer points to the newly allocated block. Since the probability of virtual bypass is low in the DSB algorithm, we do not expect the BB to be flooded with virtual bypasses/victims.
3. When L2 cache misses are serviced with both LLC and BB, depending on whether (virtual) bypassed blocks are accessed earlier than the corresponding replacement victims, the bypassing probability is adjusted accordingly, same as the original DSB algorithm.

To summarize, we present three adaptive bypassing designs for inclusive LLCs: (1) DSB with a BB containing

data (I-DSB-BB-data) (2) DSB with a data-less BB (I-DSB- BB) (3) DSB with a data-less BB, which is augumented for tracking bypass effectiveness (I-DSB-BBtracking). Since the data stored in the BB are very rarely accessed, we mainly focus on I-DSB-BB and I-DSB-BBtracking in the rest of the paper.

The design of the bypass buffer used for I-DSB-BBtracking is shown in Figure 3. It is organized as a set associative strucutre of multiple BB-entries. In each entry, the BB-tag is the block address of the bypassed block. It is different from the tag stored in the cache because the index bits are removed from the cache tags in any cache. To track the effectiveness of bypassing, a virutual bypass bit and a competitor pointer are maintained in each BB-entry.



ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

valid	Virtual Bypass	Competitor pointer	BB-tag

Figure 3: Various fields present in a BB-entry

E. Hardware Overhead of Bypass Buffer

Here, we discuss the hardware storage of bypass buffer for I-DSB-BBtracking. In most of our experiments (if otherwise not mentioned), we use a 64-entry BB which is organized as a 4-way set associative structure. Each entry contains a 54-bit (= 64-bit address – 6-bit block offset – 4- bit index) tag field, a competitor pointer and two status bits as shown in Figure 3. Since the tag field shares the same index bits for the LLC as the competitor (i.e., the bypassed block and the competitor are in the same cache set), the competitor pointer is reduced to a way pointer. For a 16-way LLC, a competitor pointer requires 4 bits. So, the overall hardware storage cost of the BB is 64× (54+4+2) = 3,840 bits.

In comparison, the original DSB algorithm keeps a 16- bit partial tag for bypassed block, a competing way pointer (4 bits for 16-way set associative cache) and 2 status bits. As a 2MB cache with 64-B blocks has 2048 sets, the overall cost is 22x2048 = 44K bits. Therefore, I-DSB-BBtracking incurs 91% less hardware overhead compared to DSB cache bypassing algorithm.

For a 4MB shared LLC in a 4-core system, we use 256 entry bypass buffer. The storage cost of our BB-based design is 256*(52+4+2) = 14.5K bits whereas the original DSB implementation costs 88K bits of storage.

The auxiliary tag directory and randomization hardware as proposed in DSB remain the same in I-DSB-BB and I- DSB-BBtracking and they cost 46.8K bits for a 2MB LLC (93.5K bits for a 4MB LLC) and 51 bits, respectively [6].

IV. EXPERIMENTAL METHODOLOGY

To model the performance impact of our proposed approach, we use an in-house execution-driven simulator. This simulator uses the SimpleScalar [2] frontend while the timing simulator is completely revamped to model a 4-way issue superscalar processor with a 64-entry active list. The memory hierarchy contains a 32kB 4-way set associative L1 data cache with a block size of 64 bytes (1-cycle hit latency), a 32kB 2-way set associative L1 instruction cache (1-cycle hit latency) and an 8-way set associative 256kB L2 cache with a block size of 64 bytes (10-cycle hit latency). We use 16-way set associative 2MB LLC with a block size of 64 bytes (30-cycle hit latency) for our single-core systems. For multi-core systems, we increase the capacity of shared LLC to 4MB. The LLC in our baseline system is inclusive and enforces inclusion on L1 and L2 caches by sending back invalidations for LLC evictions. L1 and L2 caches are kept non-inclusive as mentioned before. The main memory latency is 200 cycles.

We include all the SPEC 2000 and SPEC 2006 benchmarks that we were able to compile and run using the SimpleScalar ISA (PISA), 16 from SPEC 2000 and 7 from SPEC 2006. We use reference input for all the benchmarks and use Simpoint [8] tool to find simulation phases. For each benchmark, we use a representative 100M Simpoint for simulations. We also include 4 additional memory intensive phases and label them as *gap-2*, *gcc-2*, *mcf-2* and *sphinx-2*. Among the 27 benchmark Simpoints, listed in Appendix-1, we only report results for 14 selected programs phases. The selection criterion is that either these phases show performance gains, measured with instructions per cycle (IPC), of more than 3% when the LLC size is increased to 16MB from the baseline size of

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

2MB or they have more than 5 LLC misses per 1K instructions (MPKI).

To evaluate our proposed design in a 4-core system, we generate four multi-programmed workload categories: (a) 4H: all 4 benchmarks with high MPKI; (b) 3H1L: 3 benchmarks with high MPKI and 1 benchmark with low MPKI; (c) 2H2L: 2 benchmarks with high MPKI and 2 benchmarks with low MPKI; and (d) 1H3L: 1 benchmark with high MPKI and 3 benchmarks with low MPKI. We do not include category 4L in this study because of its low memory intensiveness. In each category, eight multi- programmed workloads are generated randomly. The detailed list of all the combinations is in Appendix-2. The performance of multi-programmed workloads is measured using the weighted speedup as proposed in [24].

V. EXPERIMENTAL RESULTS

A. *Effect of bypassing on LLC performance*

We start our experimental analysis with evaluating the LLC miss rates obtained by the baseline system, DSB with a non-inclusive LLC, I-DSB-BB with an inclusive LLC and I- DSB-BBtracking with an inclusive LLC and the results are shown in in Figure 4. I-DSB-BB and I-DSB-BBtrackingboth use a 64 entry bypass buffer which is organized as a 4- way set associative structure. DSB is able to reduce LLC misses for many benchmarks. For some benchmarks such as *equake*, *mcf*, *parser* and *sphinx*, I-DSB-BB has slightly more misses than DSB. It is caused by inclusion victims, i.e. few live L1 and L2 blocks being invalidated due to back invalidations. Between I-DSB-BB and I-DSB-BBtracking, some entries in a 64-entry BB are evicted early, which affects the accuracy of tracking the bypassing effectiveness for I-DSB-BBtracking. Therefore, I-DSB-BBtracking has a slightly higher number of misses than I-DSB-BB. This difference gets smaller as we increase the number of BB entries (see Section V-C on the impact of the BB size). Also, the benchmark *mcf* from SPEC2000 (*mcf-2k*) have a very low number of LLC misses for a 2MB LLC therefore there is no impact of using bypassing for this LLCconfiguration. But we include this benchmark because it shows high MPKI due to thrashing behavior when the LLC capacity is 1MB. (More results in Section V-F).

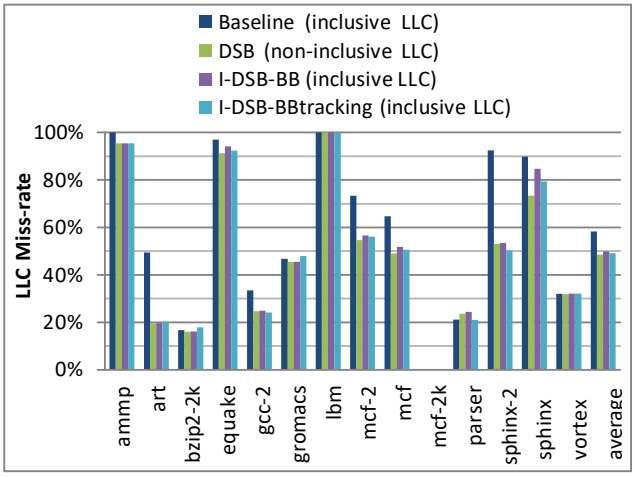


Figure 4: LLC miss rate comparison for different designs

Next, we analyze the fraction of bypassed LLC allocations and fraction of bypass buffer hits (BB-hits) for I- DSB-BBtracking. Figure 5 shows the fraction of LLC allocations which are decided to be bypassed and Figure 6 shows the fraction of bypassed blocks which are recalled by L2 cache and experience a hit in BB. As shown in Figure 5, many benchmarks heavily prefer bypassing of cache blocks. For most of the benchmarks with high fraction of cache bypassing (e.g. *art*, *gcc-2*, *mcf* and

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

sphinx), the bypassing is effective and we observe significant reductions in LLC misses as shown in Figure 4. The exception is the benchmark *equake*, which shows a high fraction of bypasses and yet does not achieve significant reduction in LLC miss rate, meaning that both the bypassed blocks and their competitor LRU blocks have no reuses. The benchmarks *bzip2-2k, gromacs, lbm, mcf-2k, parser* and *vortex* show low amount of bypassing and therefore their LLC miss rates are largely unaffected. The benchmark *ammp* has a repetitive access pattern with very long reuse distances and causes the

tracking information (i.e. *bypass block –competitor pointer* pair) to be cancelled before it can be detected to be effective (as mentioned in III-A). Therefore, the bypassing probability stays low, and this minimal amount of bypassing leads to a small reduction in the LLC miss rate.

A key aspect of our motivation of proposing the Bypass Buffer is that the bypassed blocks are not likely to be reaccessed by upper levels. We also mentioned in the Section III-C that hits in bypass buffer should be very rare and therefore it does not incur any performance penalty if BB-entries are data-less. In Figure 6, we present the fraction of hits in the bypass buffer (called BB-hits) normalized to the number of cache bypasses. There are two key observations that can be made from comparing Figure 5 and Figure 6. First, for most benchmarks with high amount of bypassing, the fraction number of BB-hits is very low. Second, benchmarks such as *bzip2 (SPEC2000)* and *vortex* have relatively higher fraction of BB-hits. A BB-hit indicates an incorrect bypassing decision and results in lower probability of bypass. Therefore the fraction of bypassed blocks is relatively low for these two benchmarks.

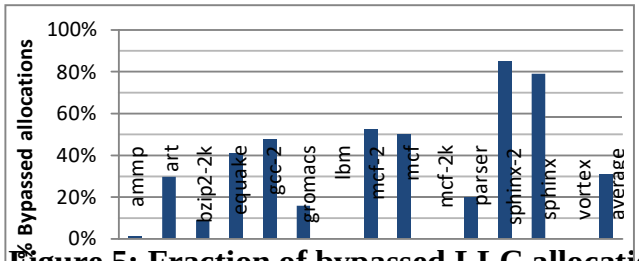


Figure 5: Fraction of bypassed LLC allocations for I-DSB- BBtracking

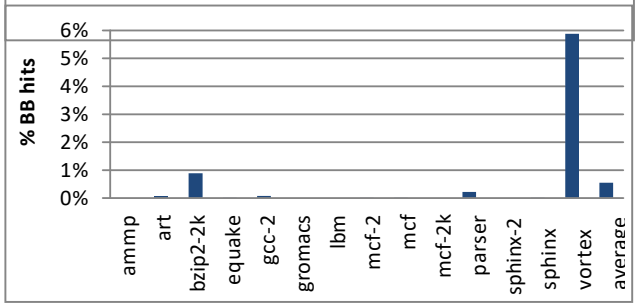


Figure 6: Fraction of bypassed blocks incurring a hit in the bypass buffer for I-DSB-BBtracking

B. Performance improvement of Bypass Buffers

In this experiment, we evaluate the effectiveness of our proposed I-DSB-BB and I-DSB-BBtracking designs. We present the performance improvements, measured in the instruction per cycle (IPC)

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

speedups, as shown in Figure 7. For reference, we also show the IPC improvement of the non-inclusive DSB design. From Figure 7, we can see thatDSB achieves an 11.6% IPC improvement for non-inclusive cache hierarchies on average, using the geometric mean (Gmean), across the high MPKI benchmarks. Both I-DSB- BB and I-DSB-BBtracking enable bypassing for inclusive LLCs. I-DSB-BB achieves an 9.8% performance gain on average while I-DSB-BBtracking has an overall speedup of 9.4%. As discussed in Section III-D, I-DSB-BBtracking uses the BB to keep usage information for both bypassed blocks and the replacement victims chosen to participate in virtual bypass. Compared to I-DSB-BB, some entries in a 64-entry BB are evicted early, which affects the accuracy of tracking the bypassing effectiveness. Therefore, I-DSB- BBtracking has slightly lower performance than I-DSB-BB. When increasing the BB size to 128 entries, the performance gains of I-DSB-BBtracking is improved to 10.0%. Considering its significant savings in hardware cost and relatively minor performance difference to I-DSB-BB,we consider I-DSB-BBtracking as our design of choice.

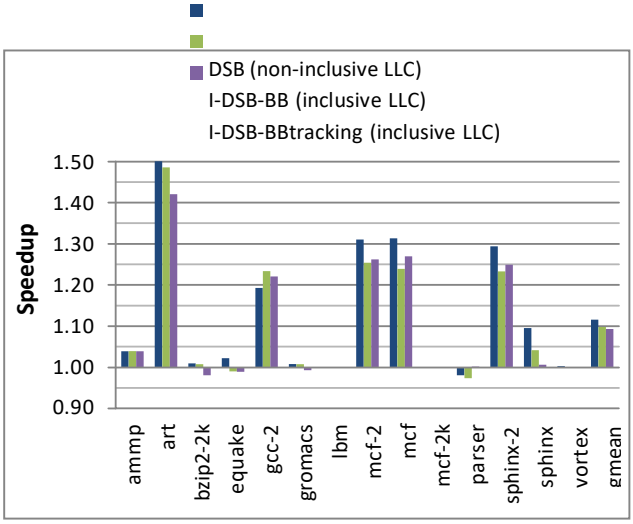


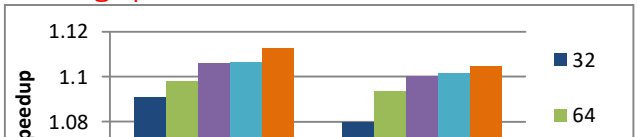
Figure 7: Performance improvements of DSB, I-DSB-BB and I-DSB-BBtracking (w.r.t. the baseline inclusive LLC with the LRU replacement policy)

C. *Effect of the Bypass Buffer size*

In this section, we analyze the performance of I-DSB- BB and I-DSB-BBtracking for different bypass buffer sizes. As mentioned in Section II the number of bypass buffer entries required should be small because of the lifetime of most bypassed cache blocks in upper cache levels is small. Therefore, we chose to experiment with a design with 64 BB-entries. Figure 8 shows the geometric mean of IPC speedup of benchmarks for different sizes of bypass buffer. It can be observed from the results clearly that increasing the size of bypass buffer increases the performance of I- DSB-BB gradually. On the other hand, I-DSB-BBtracking gains performance with increasing size of bypass buffer more rapidly in the beginning but it saturates after 128- entries.

To elaborate, increasing the bypass buffer size for I- DSB-BB allows the tags of bypassed blocks to be stored in BB longer. Therefore, with the increase of size of bypass

buffer the performance increases. I-DSB-BBtracking has two benefits from increasing the size of BB. The first is the same as I-DSB-BB and the second is that the bypass tracking mechanism has more entries and the detection of effectiveness and ineffectiveness of bypassing is done more accurately. This is the reason why I-DSB-BBtracking recovers more performance compared to I-DSB-BB whennumber of BB-entries is increased.



ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

performance compared to non-inclusive case. Multiple consecutive accesses to the same cache set are inserted at the LRU position in the set and evict each other in the case of this benchmark. This phenomenon does not hurt performance in a non-inclusive LLC but it degrades performance when inclusion is enforced via back invalidation. As a result, DRRIP (inclusive LLC) shows 17% and 7% lesser performance for *sphinx* and *sphinx-2* respectively compared to DRRIP (non-inclusive LLC). On the other hand, I-DSB-BBtracking recovers all the performance in the case of *sphinx-2* and ensures no slowdown in the case of *sphinx*. This recovery of performance is enabled by the bypass buffer which lets the bypassed cache blocks stay in the LLC for longer duration as opposed to using DRRIP which evicts the LRU inserted cache blocks on the next miss to the cache set.

Figure 8: Performance of I-DSB-BB and I-DSB-BBtracking for different bypass buffer sizes

D. Comparison to a high performing replacement algorithm, DRRIP

DRRIP [9] is one of the high performing cache replacement algorithms for LLCs. It provides scan-resistance and thrash-resistance via the use of bimodal insertion policies and set dueling [20]. We adopt the source code (distributed by the authors) to incorporate DRRIP in our simulator framework. We compare the performance gains of the DRRIP replacement policy with our proposed I- DSB-BBtracking design and the results are shown in Figure

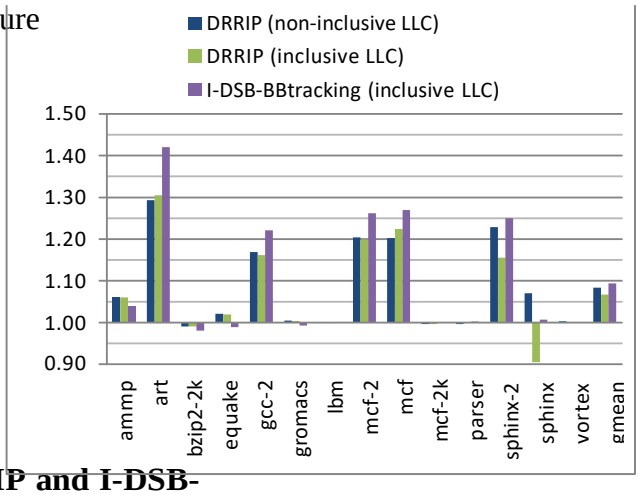


Figure 9: Performance improvements of DRRIP and I-DSB-

9. In this experiment, both DRRIP and our I-DSB- BBtracking are used for an inclusive LLC and the baseline is an inclusive LLC with the LRU replacement policy. For reference, we also include a bar for each benchmark in the figure showing the IPC speedup of DRRIP on a non- inclusive LLC and it is normalized to same baseline system as other two bars (inclusive LLC with the LRU replacement policy).

From Figure 9, we can see that both DRRIP and our proposed I-DSB-BBtracking can support the inclusion property while achieving performance gains over the baseline LLC, an average of 9.4% and 6.7%, respectively. For benchmarks such as *art*, *gcc-2*, *mcf* and *sphinx*, I-DSB- BBtracking outperforms DRRIP significantly although DRRIP already shows good performance. The reason is that DRRIP still needs to allocate a block even if it knows the thrashing behavior. Therefore, in our 16-way set associative LLC, out of 16 ways in a cache set, one way is being thrashed while other ways enjoy hits when being reused. Bypassing eliminates such inefficiency and can fully utilize the 16 ways for data reuse. The other limitation of any replacement algorithm, as discussed in Section II, causes both program phases of the benchmark *sphinx* to degrade

BBtracking (w.r.t. the baseline inclusive LLC with the LRU replacement policy)

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

E. Performance gains of Bypass Buffers in the presence of a stream-buffer

High performance microprocessors employ hardware prefetching mechanisms to hide memory latency. For our high MPKI benchmarks, when we employ a stream buffer [13] with the following configuration: 8 four-entry stream buffers with a PC-based two-way 512-entry stride prediction table, the streaming buffer prefetcher results in a 39% IPC improvement on average. Here, it is interesting to see whether the intelligent LLC management can still benefit in the presence of the stream buffer. As shown in Figure 10, when we use I-DSB-BBtracking algorithm for inclusive cache hierarchy with such a stream buffer, a 9.2% IPC improvement (on average) is observed over the baseline inclusive cache hierarchy with the LRU replacement policy and the same streaming buffer.

In comparison, inclusive DRRIP provides an IPC speedup of 7.3% in such a case. Therefore, we conclude that bypassing for inclusive cache using I-DSB-BBtracking can outperform intelligent replacement policy like DRRIP in presence of stream prefetching as well.

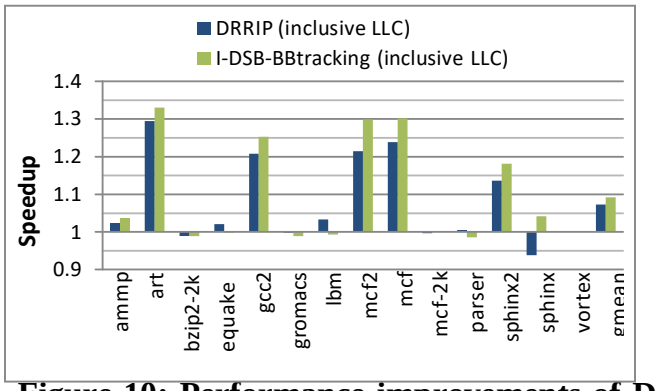


Figure 10: Performance improvements of DRRIP and I-DSB-BBtracking in presence of a stream buffer. The baseline is an inclusive LLC with the LRU replacement and a stream buffer prefetcher.

F. Sensitivity to LLC configurations

We also evaluate DSB, I-DSB-BB, I-DSB-BBtracking and DRRIP for different LLC sizes and set-associativity. Their average performance gains across the high MPKI benchmarks are shown in Figure 11. The baseline LLC for each set of bars is of specified size and uses LRU replacement policy. The same BB size, 64, is used for all these cache configurations. The rationale behind the same size of bypass-buffer being effective for various cache sizes is that the bypass-buffer is sized for the lifetime of bypassed cache blocks in upper cache levels and which does not change significantly with the configuration of LLC. Moreover, the number of cache sets (which increases with increasing cache size and decreasing set associativity) increases the overhead of bypassing for DSB and I-DSB-BB while I-DSB-BBtracking has a fixed overhead.

From Figure 11, we can see that the adaptive bypassing schemes achieve high performance for all the cache configurations that we studied. As we move towards a smaller capacity of LLC, the performance gains of bypassing (DSB) as well as replacement (here DRRIP) are reduced. This is due to the thrashing behavior of some of the benchmarks which causes the baselines of 1MB and 2MB to be similar performing. On the other hand, DSB and DRRIP both provide thrash resistance and 2MB LLC can fit bigger portion of the working set compared to 1MB LLC and enjoys significantly more hits. It should also be noted that, with bigger LLC capacity of 4MB the

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

difference in performance between DSB (non-inclusive LLC) and DRRIP (inclusive LLC) grows comparatively small. This is due to two reasons. Firstly, the negative effect of enforcing inclusion decreases when LLC capacity is bigger. Secondly, DSB does not evict data from LLC while DRRIP has to insert data in LLC which hurts more performance for smaller cache associativity/capacity and vice-versa. Overall, we can see that I-DSB-BBtracking consistently outperforms DRRIP across a variety of LLC configurations we studied.

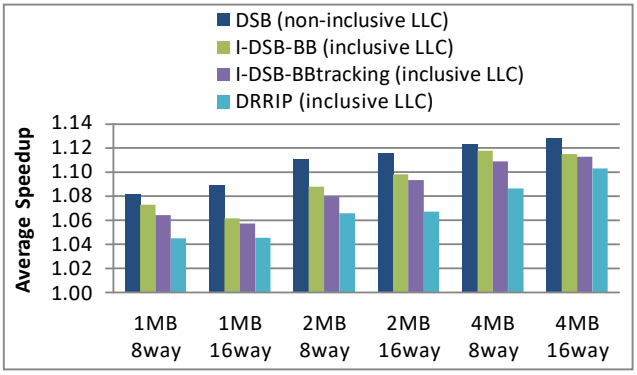


Figure 11: Performance improvements of DSB, I-DSB-BB and I-DSB-BBtracking for different cache configurations. (the baselines are inclusive LLCs with the corresponding configurations)

G. *Bypass Buffers for Shared Last Level Caches*

In the next experiment, we focus on the effectiveness of the BB for the shared LLCs in multi-core systems. For a4MB LLC shared among 4 cores, we employ a 256-entry BB and measure the performance of 32 multi-programmed workloads as described in Section IV. Although the original DSB bypassing algorithm is inherently thread unaware, it outperforms thread-aware DRRIP (TA-DRRIP) [9] in our

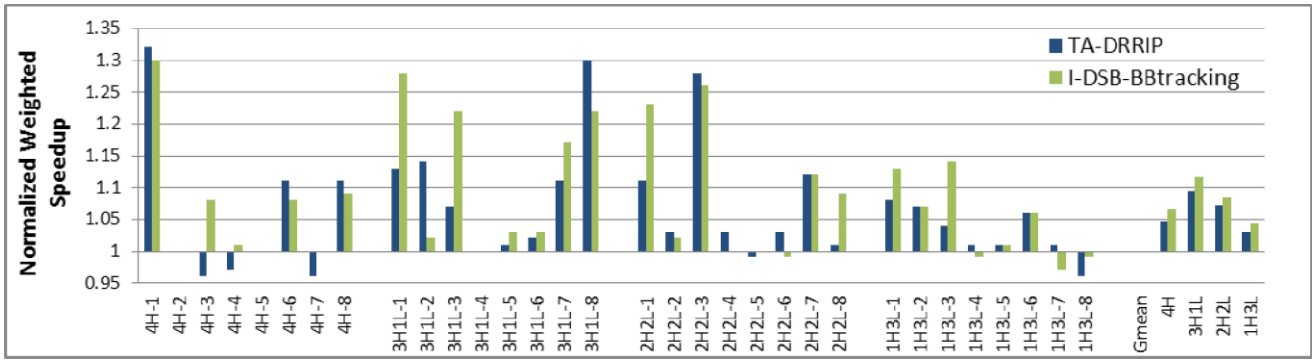


Figure 12: Performance improvements of I-DSB-BBtracking and TA-DRRIP for a 4MB inclusive LLC for multi-programmed workloads (w.r.t. the baseline with the LRU replacement policy)

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

experiments. We did not observe any significant improvement from making DSB algorithm thread aware therefore we conclude that a simpler (thread unaware) design is a more cost effective approach. We also compared the performance of a shared BB and a private BB in our design. A thread private bypass buffer limits the amount of bypassed blocks a thread can store in the bypass buffer. Moreover, if there are a few threads in a multi-programmed workload that do not prefer bypassing, many BB-entries may be underutilized. In our experiments, the shared bypass buffer design performs superior to a private bypass buffer design and therefore it is the choice in our design of bypass buffer for shared LLC.

Our simulations results comparing the performance of TA-DRRIP and I-DSB-BBtracking are reported in Figure 12, which shows that our proposed I-DSB-BBtracking algorithm provides an average speedup of 6.6%, 11.6%, 8.4%, 4.3% for the category 4H, 3H1L, 2H2L and 1H3L respectively. The workloads in these categories have one or more programs, which have high MPKI and they compete heavily for the shared LLC. The bypassing algorithm selectively bypasses the cache blocks and therefore provides the cache capacity to data blocks with smaller reuse distances.

As shown in Figure 12, TA-DRRIP also improves the performance with an average of 4.8%, 9.4%, 7.2% and 2.9% for categories 4H, 3H1L 2H2L and 1H3L respectively. Due to strict allocation policy, it always has to allocate a data block in cache. Moreover, multiple insertions from different threads at lower LRU stack positions in a cache set result in victimizing the blocks prematurely and a loss of performance due to inclusion. This is responsible for relatively lower average performance of TA-DRRIP compared to I-DSB-BBtracking algorithm in all the categories. Therefore we conclude that bypass buffer can make cache bypassing effective for inclusive shared LLCs as well.

H. Energy Consumption

In this section, we compare the energy consumption of I- DSB-BBtracking scheme with the baseline. We use McPAT tool [19] to obtain the static and dynamic power consumption results. The power model is based on the 45nm technology and 3.4 GHz frequency. Figure 13 shows the energy consumption for each benchmark when the 100M simulation phase is executed on the baseline system as well as when the I-DSB-BBtracking mechanism is deployed. From Figure 13, we can see that for most benchmarks DSB- BBtracking reduces overall energy consumption by up to 25.6% and 6.2% on average. Most of the energy savings are a result of reduced execution time which translates into significant savings in static energy consumption. For the remaining benchmarks, on which DSB-BBtracking has little performance impact, the energy consumption impact is nearly negligible.

Figure 13: Comparing energy consumption of I-DSB- BBtracking with the baseline

I. Memory Bandwidth Reduction

As shown in Section V-A, I-DSB-BBtracking reduces LLC misses. This in turn reduces the off-chip memory traffic and the execution time. As the reduction in LLC miss-rate is typically higher in proportion than the reduction in execution time, average memory bandwidth is reduced. As shown in Figure 14, we observe average memory bandwidth of 3.3 GB/s in the baseline for benchmark *art* while I-DSB-BBtracking reduces it to 1.9 GB/s. For benchmarks such as *bzip2*, *equake*, *gromacs* and *mcf*, we observe slight increase in the bandwidth. *Bzip2* and *gromacs* have slightly more misses compared to the baseline and therefore the bandwidth is increased. On the other hand, *equake* and *mcf* have reduced number of LLC misses and yet the bandwidth increases due to faster execution of the program. Overall, the aggregate memory bandwidth is reduced by 13% on average across the benchmarks.

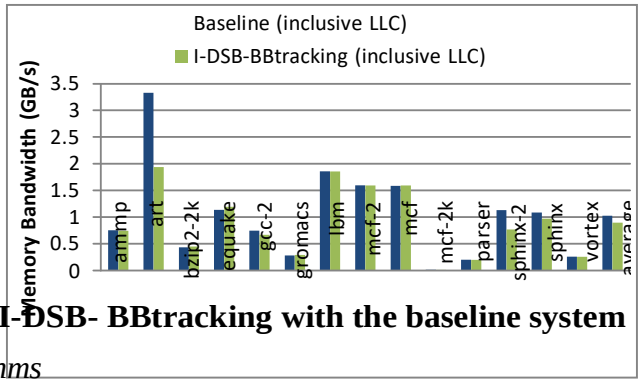


Figure 14: Comparing memory bandwidth of I-DSB- BBtracking with the baseline system

J. Additional Benefits of Cache Bypassing Algorithms

Many proposals for high performance cache management focus on replacement algorithms. A common theme is to alter the insertion policy whereas the allocation policy is strict and each incoming block is allocated space in an LLC. The key contribution of a bypassing algorithm is tocombine a placement / allocation policy with the replacement policy. Bypassing has some interesting benefits for upcoming memory technologies. For example, phase- change memory (PCM) provides high integration density while suffers from limited write endurance. In a design using PCM technology for LLCs, we can utilize cache management algorithms like DSB to reduce the number offill operations to the LLC. This would not only increase the life of such PCM based LLC structure but also increase the overall performance (as shown in previous sections). In Section V-A, Figure 5 shows the fraction (in percentage) of bypassed LLC allocations when the I-DSB-BBtracking algorithm is used. From this figure, we can see that for certain benchmarks, up to 80% of the LLC allocations can be bypassed while enjoying good performance gains.

VI. RELATED WORK

There is a plethora of research work on designing high performing cache replacement algorithms. Recently proposed replacement algorithms [3][9][16][20][23] for LLCs focus on providing thrash-resistance and scan- resistance. There is also a significant amount of work on cache bypassing algorithms [4][5][6][7][12][15][18][21] [22][25]. The results of 1st JILP Cache Replacement Competition indicated that bypassing algorithm like DSB[6] can be an effective method to improve the LLC performance for a wide variety of workloads. Recent work by Li et al.
[18] uses a similar mechanism to decide the effectiveness of bypassing. The key difference is they do

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

not use probabilistic bypassing and instead maintain a signature based history per program counter to make bypassing decisions. A major disadvantage of using signature based

VII. CONCLUSIONS

In this work, we focus on the inherent limitation of inclusive caches in utilizing cache bypassing. We propose and evaluate a novel solution, called a bypass buffer (BB), to overcome this limitation. The bypassed data blocks skipthe LLC and their tags are stored in the BB. When a tag is replaced from the BB, it invalidates the upper cache levelsto maintain the inclusion property. We show that for a well design bypassing algorithm a relatively small BB is sufficient to reap most of the performance gains of bypassing. Our proposed BB also enables us to significantly reduce the storage hardware cost of bypassing algorithms as it readily provides the usage information of bypassed cache blocks. Our experimental results show that our proposed design achieves high performance and outperforms a recently proposed high performing replacement algorithm, DRRIP, in both single core and 4-core systems. Our evaluation of our proposed design on different cache configurations and in presence of a stream prefetcher shows that it provides a cost-effective design for inclusive LLCs.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments to improve our paper. This research is supported by an NSF grant CCF-1216569, an NSF CAREER award CCF-0968667 and a research fund from Intel Corporation.

REFERENCES

[1] L. A. Belady. “A Study of Replacement Algorithms for aVirtual-storage Computer.” In *IBM Systems Journal*, 5(2):

approach is to have the program counter being sent to LLC from each core with the cache access. This can be an expensive requirement to fulfill in a real design. Therefore, DSB is our choice of bypassing algorithm in this study. However, it should be noted that a bypass buffer can help

[2]

[3]
78-101, 1966.
D. Burger and T. M. Austin. “The Simplescalar Tool Set Version 2.0.” Technical Report, *Computer Science Department, University of Wisconsin-Madison, 1997.*
M. Chaudhuri. “Pseudo-LIFO: The foundation of a new family of replacement policies for LLCs.” In *MICRO 2009.*
employ any complex bypassing algorithms on inclusiveLLCs.

Gaur et al. [7] proposed cache bypassing algorithms specific to exclusive cache hierarchies. The key contribution

[4] C. H. Chi and H. Dietz, “Improving cache performance by selective cache bypass.” In Proceedings of the Twenty- Second Annual Hawaii International Conference on System Sciences, 1989. Vol. I: Architecture Track, 1989, pp 277 -

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

of this work is to tackle the problem of no reuse information being available for cache blocks (a cache hit de-allocates the cache block and the reuse information is lost). The work presents both insertion and bypassing algorithms designed for exclusive caches.

Jaleel et al. [9] pointed out a key performance problem in inclusive cache hierarchies due to invalidatio victims. The

[5]6]
285 vol.1.
H. Dybdahl and P. Stenstro¨m, “Enhancing Last-Level Cache Performance by Block Bypassing and Early Miss Determination.” In *ACSAC06, 2006.*
H. Gao and C. Wilkerson. “A dueling segmented LRU replacement algorithm with adaptive bypassing.” In *Proceedings of the 1st JILP Workshop on Computer Architecture Competitions,* 2010
non-inclusive cache performance is achieved in this work by
[7] Jayesh Gaur et al. “Bypass and Insertion Algorithms for

making the last-level cache replacement policy aware of the temporal locality in upper levels of cache. This work can be used in conjunction with our scheme to further enhance performance for inclusive caches.

[8]

[9]
Exclusive Last-level Caches.” In *ISCA 2011.*
G. Hamerly et al. “SimPoint 3.0: Faster and More Flexible Program Analysis.” In *MoBS 2005.*
A. Jaleel et al. “Achieving Non-Inclusive Cache Performance with Inclusive Caches -- Temporal Locality Aware (TLA) Cache Management Policies.” In *MICRO- 2010.*

[10] A. Jaleel et al. “High performance cache replacement using re-reference interval prediction (RRIP).” In *ISCA 2010.*
[11] 1st JILP Workshop on computer architecture competitions (Cache Replacement Championship). http: //jilp.org/jwac-1/
[12] Teresa L. Johnson et al.“Run-Time Cache Bypassing.” *IEEE Trans. Comput.* 48, 12 (December 1999), 1338-1354.
[13] N. P. Jouppi. “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers.” In *ISCA 1990.*
[14] R. Karedla et al. “Caching Strategies to Improve Disk System Performance.” *Computer, vol. 27, no. 3, pp. 38-46, Mar. 1994.*
[15] M. Karlsson and E. Hagersten. “Timestamp-based Selective Cache Allocation.” High Performance Memory Systems, edited by H. Hadimiouglu, D. Kaeli, J. Kuskin, A. Nanda, and J. Torrellas, Springer-Verlag, 2003.
[16] S. Khan and D. A. Jimenez. “Insertion Policy Selection Using Decision Tree Analysis.” In *ICCD, 2010.*
[17] M. Kharbutli and Y. Solihin. “Counter-based Cache Replacement and Bypassing Algorithms.” In *IEEE Trans. on Computers, 57(4): 433-447, April 2008.*
[18] L. Li et al. “Optimal Bypass Monitor for High Performance Last-level Caches.” In *PACT* 2012.
[19] Sheng Li et al. “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures.” In *MICRO-42, 2009.*

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

[20] M. K. Qureshi et al. “Adaptive insertion policies for high performance caching.” In *ISCA, 2007*.
[21] J.A. Rivers and E.S. Davidson, “Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design.” In *ICPP, 1996.*
[22] G. Tyson et al. “A Modified Approach to Data Cache Management.” In *Proceedings of MICRO-28, 1995.*
[23] C.-J. Wu et al. “SHiP: Signature-based hit predictor for high performance caching.” In *MICRO, 2011.*
[24] A. Snavely and D. Tullsen. “Symbiotic job scheduling for a simultaneous multithreading processor.” In *ASPLOS, 2000.*
[25] L. Xiang et al. “Less reused filter: improving: L2 cache performance via filtering less reused lines.” In *ICS, 2009*.
[26] Intel. Intel Core i7 Processor. http://www.intel.com/ products/processor/corei7/specifications.htm

Appendix-1: Simulation points (starting instruction number) used for benchmarks from SPEC2000 and SPEC2006

Benchmark	**Starting Point**	**Benchmark**	**Starting Point**
ammp	1,400,000,000	mcf-2k	16,900,000,000
art	2,800,000,000	mesa	117,600,000,000
bzip2	44,800,000,000	milc	20,700,000,000
bzip2-2k	18,800,000,000	parser	42300,000,000
equake	36,700,000,000	perl	1,800,000,000
gap-2	4,200,000,000	sjeng	91,400,000,000
gap	6,900,000,000	sphinx-2	3,200,000,000
gcc-2	3,800,000,000	sphinx	83,900,000,000
gcc	6,100,000,000	swim	50,000,000,000
gromacs	62,900,000,000	twolf	800,000,000
gzip	13,800,000,000	vortex	70,000,000,000
lbm	66,100,000,000	vpr	15,200,000,000
mcf-2	2,000,000,000	wupwise	10,900,000,000
mcf	46,300,000,000		

Appendix-2: The list of multiprogrammed workloads

	4H	**3H1L**
1	sphinx, gcc-2, milc, mcf-2k	ammp, art, mcf-2k, twolf
2	equake, lbm, gap-2, sphinx2	ammp, art, lbm, bzip2-2k
3	ammp, equake, gcc-2, lbm	equake, mcf-2k, sphinx, gap
4	lbm, art, gap-2, swim	lbm, milc, gap-2, gcc
5	sphinx, gap-2, sphinx2, swim	sphinx, gap-2, gcc-2, gzip

ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

6	sphinx2, gcc-2, swim, milc	gcc-2, sphinx-2, swim, mesa
7	lbm, sphinx, milc, swim	swim, mcf-2k, equake, parser
8	gap-2, gcc-2, sphinx2, milc	art, mcf-2k, milc, gromacs
	2H2L	**1H3L**
1	art, ammp, bzip2-2k, gap	art, wupwise, gcc, bzip2-2k
2	equake, mcf-2k, gzip, perl	equake, wupwise, vpr, twolf
3	mcf-2k, equake, vpr, vortex	art, bzip2-2k, gcc, gap
4	ammp, swim, twolf, wupwise	equake, gromacs, sjeng, perl
5	lbm, milc, bzip2-2k, sjeng	mcf-2k, mesa, parser, gap
6	sphinx, gap-2, gromacs, parser	sphinx-2, twolf, gzip, gcc
7	gcc-2, sphinx-2, gcc, vpr	gap-2, parser, sjeng, gzip
8	swim, art, vortex, gzip	lbm, bzip2-2k, gromacs, perl