Reducing Insertion of Low Reuse Blocks in SHiP++

Jonathan Pollard

ABSTRACT

SHiP++ is a top performing last level cache (LLC) replacement policy that was created during the 2017 second cache replacement championship by improving upon the Signature-Based Hit Predictor (SHiP) replacement policy. This paper discusses ways to improve the performance of SHiP++ by reducing the insertion of blocks with low reuse into the LLC. First, adding bypassing is discussed and all the different ways to improve bypassing to get optimal performance when implemented in the SHiP++ policy. Second, adding more priority to evicting writeback blocks with typically low reuse can show benefits across some benchmarks. Lastly, combining bypassing and priority writeback eviction can lead to significantly improving the SHiP++ policy.

Each modification to SHiP++ that was implemented was evaluated against 28 benchmarks from the 2017 second cache replacement championship, for a 2MB LLC with configuration 1 a single core with no prefetching, and configuration 2 a single core with L1/L2 prefetchers. The modifications of adding bypassing and priority writeback eviction to SHiP++ improves performance over LRU compared to SHiP++ by 19.2% and 19.8% for configuration 1 and configuration 2 respectively. This comes with only adding a total of 29 Bytes extra of overhead compared to the original SHiP++ implementation.

1. INTRODUCTION

Cache Replacement Policies are extremely important in improving the overall design of a computer. This is due to the fact that with an efficient policy more useful data blocks will stay in cache longer. This will allow for lower usage of off-chip memory which in turn will reduce the number of long-latency memory accesses, and also increases energy efficiency since more data will be accessed from the on-chip cache. There are not many ways to improve all of this at once which is why improving cache replacement policies is such a unique and important research area. LRU, the standard replacement policy that every new policy typically gets compared to, works extremely well with programs that have high temporal locality. For this reason, it works so well with lower-level caches such as L1 due to the high temporal locality in the blocks that are at the lower levels. However, in LLC LRU is not as effective due to most of the temporal locality being filtered out by this point in the memory hierarchy. This is why replacement policies such as SHiP++ were created to intelligently pick the insertion position of each block placed into LLC in order to get the best performance from this cache level.

2. BACKGROUND

SHiP++ inserts blocks into cache using a modified version of Re-Reference Interval Prediction (RRIP) policy. RRIP uses a 2-bit counter for each cache line called Re-Reference Interval Prediction Value

(RRPV), which determines which block to be evicted next. When there is a cache miss the first way in the set that is found to have RRPV equal to three is the victim, and if no such block, then every RRPV in the set is incremented and a value of three is searched for again until a victim is found. On each hit the RRPV is set to zero for that cache line, and for misses the inserted cache line is either set to RRPV equals two for high priority install or RRPV equals to three for low priority install.

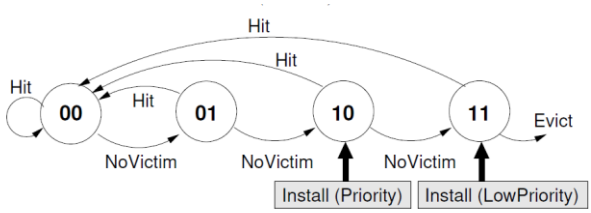

Figure 1: Re-Reference Interval Prediction

Insertion position for RRIP can either be statically set or dynamically chosen at runtime. SHiP dynamically tracks where each cache line should be inserted using the Signature History Counter Table (SHCT). The SHCT uses the past behavior of the PC to accurately determine the potential reuse of the cache line being brought in. Each entry of the SHCT has a counter called SHCTR that holds a value from 0-7, where 0 means low reuse probability and 7 means high reuse probability. Each entry is also indexed by a 14-bit signature that uses the lower 14-bit of the PC. When a memory block is inserted into cache the signature of the PC is stored for the SHCT as well as a reused bit labeled R to denote if the block was used while in cache. 64 sample sets are used to reduce the total overhead for storing the signature and reuse bits. When a sample set block has a hit the SHCTR for that block gets incremented saturating at 7 and the reuse bit gets set to one, and if the block is evicted without being used which means reuse bit is zero then SHCTR will be decremented saturating at 0. New cache blocks get installed at high priority if the SHCT entry has a SHCTR entry greater than zero and gets installed at low priority if equal to zero.

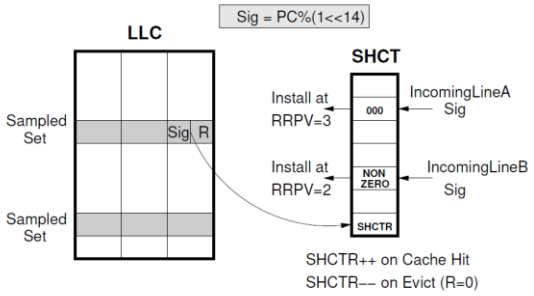

Figure 2: Signature-Based Hit Prediction

SHiP++ improved upon SHiP by improving its insertion policy and by accounting for prefetching vs. demand access in the SHCT training. SHiP++ changes insertion so that if the SHCTR entry is equal to 7 a new block will be inserted with RRPV equal to 0, since there is strong evidence of reuse this allows for the block to stay in cache longer. Another enhancement was balancing training on cache hits and cache evictions by only incrementing SHCTR on the first hit to a cache block, and decrementing or incrementing on eviction depending on if R was 0 or 1 respectively. Since writeback requests provide no indication of reuse SHiP++ always keeps writeback blocks in cache with an RRPV equal to 3 regardless of whether it has a hit or not. In order to improve learning for demand accesses vs. prefetch accesses the SHCT is now separate for the two different requests. This is done by using the lower 13-

bit of the PC for the 13 MSB's of the 14-bit index and using the LSB to denote 1 if it is a prefetch and 0 if it is a demand access. The last Improvement that SHiP++ implemented was changing the RRPV update policy for prefetched requests vs. demand requests, in order to do so 1 bit was added to each cache line called is_prefetch to denote whether the current cache line was brought in by the prefetcher. Two changes were made to prefetch update policy, the first change is that if a demand request hits on a prefetched block RRPV gets set to 3 and is_prefetch gets downgraded to 0, and by doing so any subsequent requests to the block would now update the RRPV to 0. The second change is that if a prefetch request hits a block that was prefetched then the RRPV will remain the same and nothing will be changed. With all of these Improvements SHiP++ is a policy that works significantly better than SHiP for both prefetch and no prefetch LLC configurations.

3. IMPROVING SHiP++

3.1 BYPASSING

SHiP++ already greatly improves SHiP by improving the SHCT training and improving the update and insertion policies to allow for blocks with greater reuse characteristics to stay in cache longer. However, there are still further actions that can be taken to this policy to enhance performance and allow for higher reuse blocks to have a longer duration in cache. One such change is to add bypassing to the SHiP++ policy. This allows for cache blocks that are getting brought in by a PC with poor reuse characteristics shown by the SHCT to not even get inserted into the LLC at all, as long as the block is not a writeback block that must be inserted for proper functionality of a writeback cache. This will be implemented by updating the eviction policy so that whenever looking for a victim if the SHCT entry for the block being entered has an SHCTR value of 0 then it will be bypassed and not be entered into the cache at all.

Figure 3: SHiP++ with bypassing

The results across 28 benchmarks can be seen in figure 3 which compares SHiP++ colored blue to SHiP++ with Bypassing (SHiP++B) colored orange. Overall, adding bypassing alone can lead to some promising results across many benchmarks and on average made the performance with no prefetching better while making performance with prefetching worse.

Bypassing can be very effective, but it is not always useful in all programs and can even be useful in some parts of a benchmark and hurt performance in other parts. In order for bypassing to only be used when helpful it can be dynamically tracked

at run time. This can be done by having a flag called doBypass that gets set to true when doing bypassing. Then every X number of instructions the IPC is stored for the current method being used which is either no bypassing or bypassing, and then for a shorter number of instructions the other method not currently being used will be run and then the IPC over those instructions will be calculated. If the short test of changing which method was being used yields a higher IPC, then it will be used until the next check otherwise the method will be switched back. For the first implementation using SHiP++ with Dynamic Bypassing (SHiP++DB) doBypass was initially set to true, it was dynamically checked every 10,000 instructions, and the short test length for the method not being used was 500 instructions.

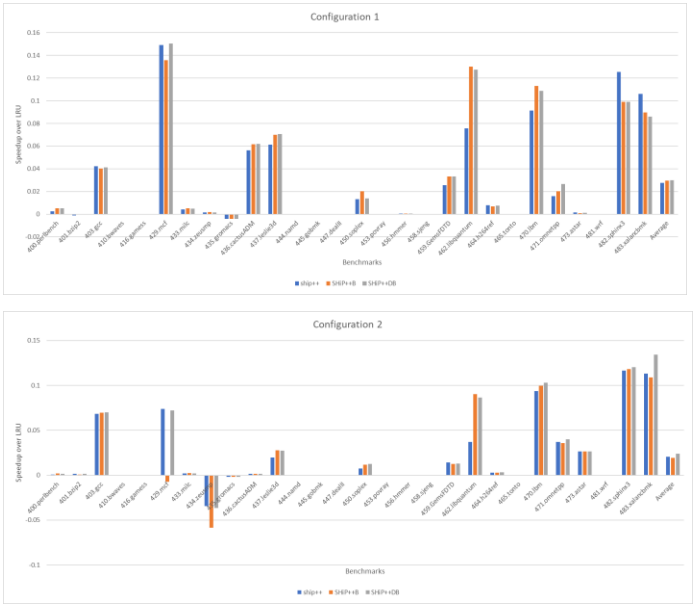

Figure 4: SHiP++ with dynamic bypassing

Figure 4 shows the performance of adding dynamic tracking to bypassing. It can be seen that there is still a similar performance increase without prefetching but now instead of a performance drop there is significant performance improvement for prefetching as well.

In order to further improve SHiP++ while bypassing it is important that SHCT stays accurate and up to date in its learning. That is why the bypassing policy was updated to now no longer use bypassing on ship sample sets. Since these sets are the only ones updating the SHCT it allows for accurate and up to date re-learning of the PC reuse characteristics without interference from bypassing. In order to see the full effect of this change the policy was originally implemented and ran without bypassing being dynamically tracked. Figure 5 shows the performance of SHiP++ with No Sample set Bypassing (SHiP++NSB), it can be seen that there are again very positive benchmarks, and average performance is very similar to that of the dynamic bypassing.

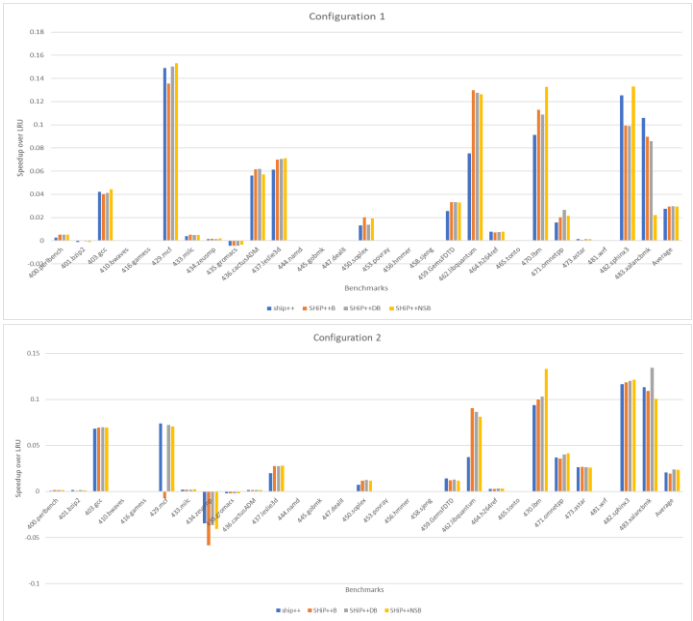

Figure 5: SHiP++ with no sample set bypassing

The final bypassing policy implemented for improving SHiP++ would

be built by combining the idea of dynamic bypassing and no sample set bypassing. The dynamic checking was also altered to work better with no sample set bypassing. The new dynamic checking runs a test every 1 million instructions, and now will run the current method for 10,000 instructions and collect the IPC over those 10k instructions. Then the other method will be given a 10k instruction warm up and then run for 10k instructions and the IPC will be computed, and if this IPC is higher than the original method being used then this will be used until the next test. The performance of this implementation is greatly improved for no prefetching and prefetching compared to SHiP++. However, the performance for prefetching is slightly worse than some of the previous bypassing implementations discussed. Figure 6 shows the performance of SHiP++ with No Sample set Dynamic Bypassing (SHiP++NSDB).

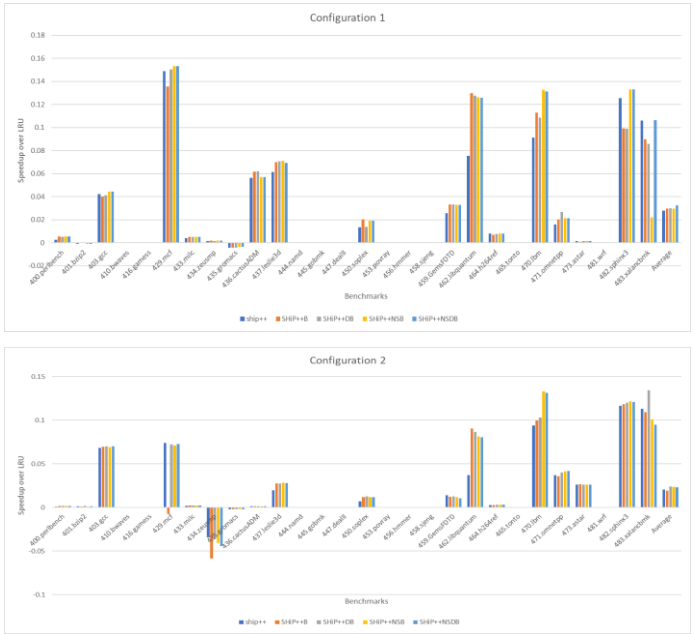

Figure 6: SHiP++ with no sample set dynamic bypassing

3.2 WRITEBACK EVICTION

SHiP++ accounts for writebacks as a side effect of a writeback cache hierarchy by always setting its RRPV value to 3 regardless of if it is a hit or insertion of a writeback block. This is great for allowing this type of block with typically low reuse characteristics to get evicted as quickly as possible. However, this could be taken a step further by updating the eviction policy to look for a writeback block with RRPV equal to 3 before looking at non-writeback blocks in the set. This will lead to even faster eviction of writeback blocks and can potentially improve performance even more. This is the reasoning behind adding priority writeback eviction to the SHiP++ policy. The actual implementation of this requires no extra overhead since the dirty bit can be used to see if each block is of type writeback.

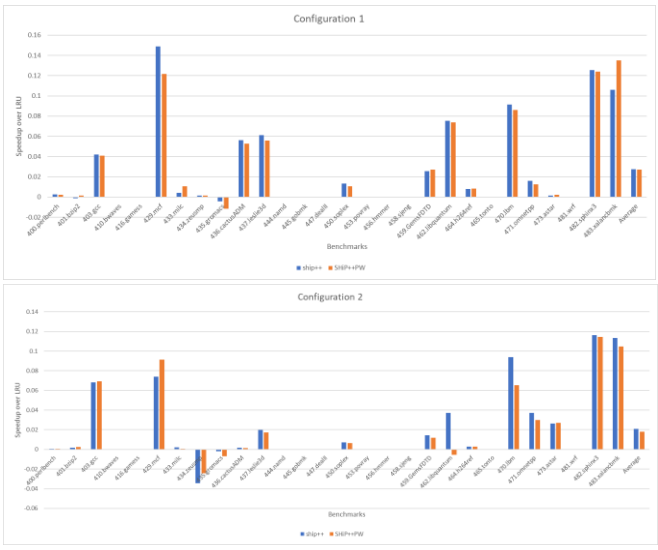

Figure 7: SHiP++ with priority writeback eviction

Figure 7 shows the results of SHiP++ with Priority Writeback eviction (SHiP++PW). The performance of SHiP++PW has some promising benchmark results but overall, the averages are worse for both configurations.

To improve priority writeback eviction the policy was updated to now no longer use this type of eviction prioritization on the sample sets. The reasoning is similar to why this was done for bypassing in order to try and keep the training of the SHCT as it was before. By doing this many of the bottlenecks introduced seemed to get better, however all of the improved benchmarks no longer existed. The results of SHiP++ with No Sample set Priority Writeback eviction (SHiP++NSPW) can be seen in Figure 8 and it can be seen that performance for both configurations are still worse than regular SHiP++.

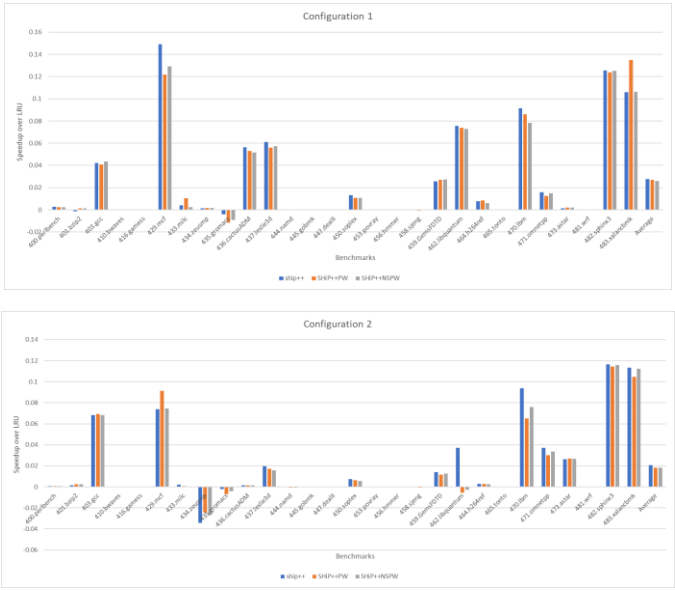

Figure 8: SHiP++ with no sample set priority writeback eviction

To try and further reduce the bottlenecks that priority writeback eviction introduces while still getting the improved results of some benchmarks, an approach that would combine SHiP++PW and SHiP++NSPW was taken. This new approach would now switch between using priority writeback eviction on sample sets and not using it and would switch at a rate of every 1,000 instructions. This would allow for normal re-learning behavior of the SHCT half the time, and SHiP++ with Re-learning Priority Writeback eviction (SHiP++RPW) ended up being able to bridge the gap between the two previous implementations. Figure 9 shows that there are still some great performing benchmarks while a reduction in bottlenecks, and this implementation even performs slightly better than SHiP++ with no prefetching even though it still significantly falls behind when prefetching is added.

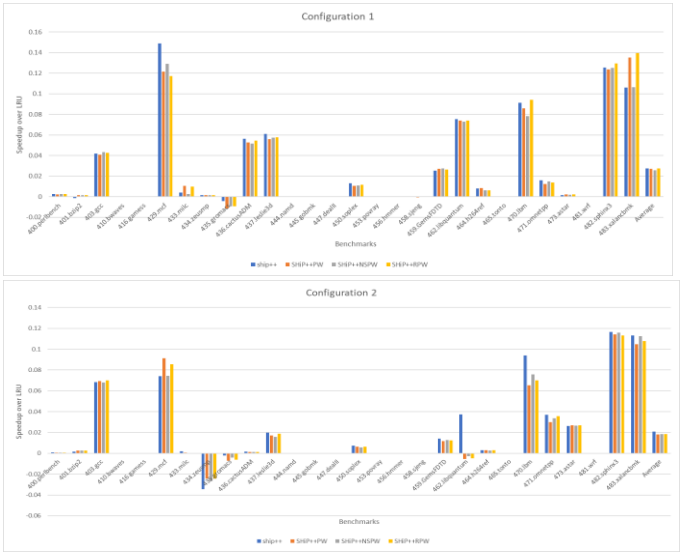

Figure 9: SHiP++ with re-learning priority writeback eviction

3.3 SHiP++BPW

Even though adding priority writeback eviction to SHiP++ alone was not successful, with it added to the SHiP++ enhanced with bypassing it is much more successful since most low re-use non-writeback blocks will be filtered out. That is why each of the three priority writeback eviction implementations were combined with the final bypassing implementation SHiP++NSDB.

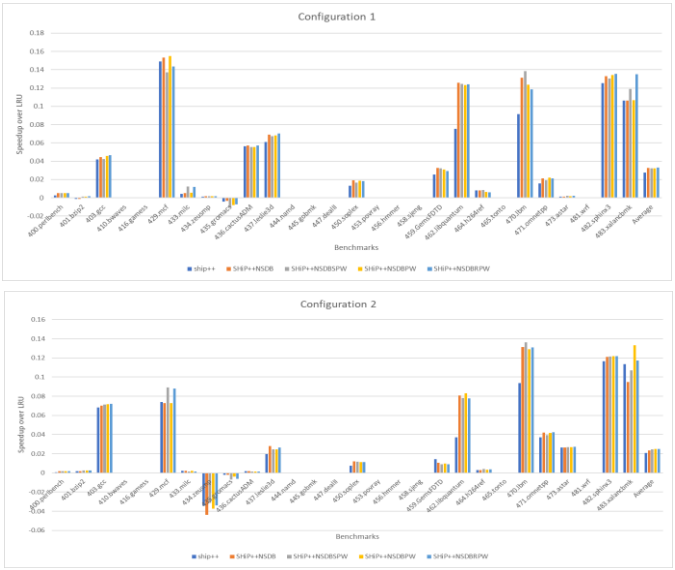

Figure 10: SHiP++ with bypassing and priority writeback eviction combined

Figure 10 shows that each priority writeback eviction implementation combined with SHiP++NSDB improves performance with prefetching significantly and performs similarly without prefetching. The best implementation however was SHiP++ with No Sample set Dynamic Bypassing Re-learning Priority Writeback eviction (SHiP++NSDBRPW). For the following sections of this paper this final improved SHiP++ LLC replacement policy will be referred to as SHiP++ with Bypassing and Priority Writeback eviction (SHiP++BPW).

4. STORAGE OVERHEAD

The SHiP++BPW uses 2-bit for every cache line for its RRPV value which is 8KB total. For prefetch configurations there is 1-bit added per line to denote whether the currently stored block was brought in by the prefetcher, which amounts to 4KB total. SHCT is a 16k entry table with 3-bits per entry to store the reuse probability, this takes up 6KB of storage. For each of the 64 sample sets there are 15 bits of overhead for each set, 14 bits for the stored PC signature, and 1 reuse bit, this amounts to 1.875KB of total storage. There is also a 64-entry sample set ID table that has 2B per entry to store which sets are sample sets. This is all the same as SHiP++, what was added is 27B for dynamic bypassing to track and test if bypassing is working. Lastly, 2B of overhead is used for relearning of the priority writeback eviction. Also as mentioned earlier in the paper priority writeback eviction does not add extra overhead for tracking which blocks are of type writeback, the dirty bit that is already stored can be used to check this.

Table 1: SHiP++BPW Storage Requirements

	Config 1	Config 2
RRPV (2-bit/line)	8KB	8KB
is_prefetched (1-bit/line)	0	4KB
SHCT (16K-entry/core)	6KB	6KB
Sampled sets (64 sets) sig + reuse (+core_id)	1.875KB	1.875KB
SampleSet ID (64 entries)	128B	128B
Bypassing	27B	27B
Priority Writeback	2B	2B
Total Storage Overhead	16.028KB	20.028KB

5. RESULTS

The 2017 second cache replacement championship framework was used for testing SHiP++BPW against SHiP++. This framework uses a 2MB LLC with 2048 sets that have 16 ways each. Two configurations were used for testing, configuration 1 a single core with no prefetching, and configuration 2 a single core with L1/L2 prefetchers. Figure 11 shows the results of SHiP++BPW improvement over SHiP++ for both configurations. The x-axis states which benchmark, and the y-axis is the speedup

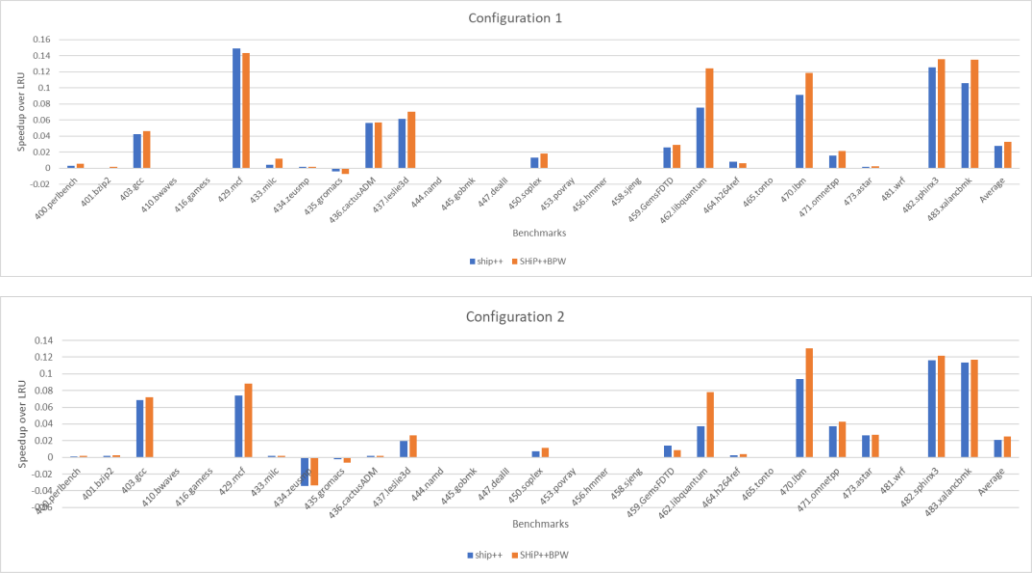

Figure 11 Performance of SHiP++BPW

over LRU (ex. if y-axis number is 0.15, it's 15% speedup). SHiP++BPW improves SHiP++ in almost every Benchmark and there are only one or two benchmarks of the 28 for both configurations in which SHiP++BPW performs slightly worse. Overall, on average across the 28 benchmarks for configuration 1 SHiP++ has a 2.76% speedup over LRU, and SHiP++BPW has a 3.29% speedup over LRU which is significantly improved. For configuration 2 SHiP++ has a 2.07% speedup over LRU, and SHiP++BPW has a 2.48% speedup over LRU which once again is significant improvement.

6. FUTURE WORK

SHiP++BPW with the current implementation calculates IPC for the dynamic bypass checking. This means that in a physical cache configuration there would need to be two floating point division calculations in order to compute these values. This type of operation is time consuming and will require many cycles after the test interval has completed to actually be able to compare and evaluate the results. Since this dynamic check is only done every 1 million instructions it should not affect performance much at all, however if this number was to ever be drastically lowered issues could occur. A potential solution is to simply check and compare the numbers of cycles without using the instructions and whichever had less cycles over the test windows is the better method to be run. Another potential fix would be to use the number of misses during the test window and whichever has less misses be used.

7. CONCLUSION

SHiP++BPW is able to improve SHiP++ by lowering the number of low reuse blocks that get inserted into LLC by adding bypassing to the policy. By doing this more useful blocks are able to stay in cache longer which in turn improves the overall performance. Also, by changing the eviction policy once bypassing is added to prioritize kicking writeback blocks out of cache allows for even further reduction of low reuse blocks unnecessarily filling up the cache. Overall, SHiP++BPW was able to improve the speedup over LRU by 19.2% for configuration 1 and by 19.8% for configuration 2 compared to SHiP++.

8. REFERENCES

[1] SHiP++: Enhancing Signature-Based Hit Predictor for Improved Cache Performance, V. Young, C. Chou, A. Jaleel, M. Qureshi, 2nd Cache Replacement Championship, 2017.