CaRP : Cache Replacement Policy for Linked Data Structure

15-740 Project Report, Spring 2018
Tushar Goyal (tgoyal1), Rini Patel (rinip), Aditi Sinha (aditis2)

ABSTRACT

Linked data structures are characterised by sequence of linked data records organized by value and one or more references (pointers) from one record to another. The irregular memory access pattern lacks spatial locality and don't work well for LRU. We compare and analyze the existing cache replacement policies (DIP, DRRIP and Hawkeye) for dynamically constructed linked data structures. We suggest a program counter based linked data structure access detection and victim cache placement for such lines to avoid pollution of primary workload cache behavior.

PROBLEM STATEMENT/MOTIVATION

Use of Linked Data Structures (LDS) such as linked list, tree, hash table etc. is a common thing in database, graphics and graph applications. The data structures are generally of the form of data and associated one or more pointers which define the next data item address to be used for processing. The usability of LDS comes from their flexibility, not their performance. LDS access, often referred to as pointer-chasing, entails chains of data dependent loads that serialize address generation and memory access. In traversing an LDS, these loads often form the program's critical path. Consequently, when they miss in the cache, they can severely limit parallelism and degrade performance.

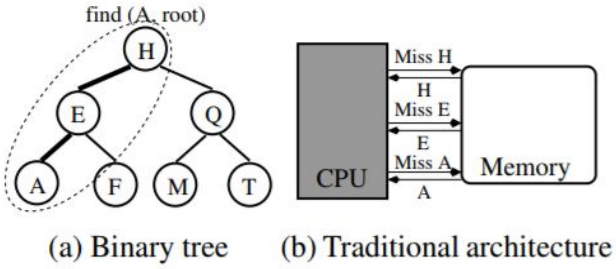

Figure 1. Pointer chasing in a traditional architecture[1]

The memory accesses generated due to pointer chasing are prone to memory latency and frequent cache and TLB misses due to highly irregular memory access. Caches only exploit spatial and temporal locality in a set of address referenced in a program. Linked data structures provide flexibility of creating and deleting nodes dynamically, and hence avoid the overhead of bothering about the size of the data structure in advance. However, due to dynamic construction, they are difficult to cache as the spatial locality between the nodes is highly dependent on the

data layout. Such accesses don't benefit much from prefetching due to low accuracy in pre-fetching and poor locality of data being accessed.

RELATED WORK

We found several efforts for implementation of hardware/software based prefetchers for LDS in academia. Some of the literatures talk about prefetching techniques to improve the performance of LDS. Roth et. al [2] suggest exploitation of the dependency between the consecutive accesses of LDS in a program and feed this information to the prefetcher to hide the memory latency. Such approaches depend highly on the amount of work in a program that can be interleaved with this latency. Luk and Mowry [3] proposed and evaluated a greedy compiler algorithm for scheduling software prefetching for linked data structures. It uses type information to identify recurrent pointer accesses, including those accessed via arrays, and may have advantages in tailoring a prefetch schedule to a particular traversal. Another popular paper by Roth and Sohi [4] talks about instrumenting the code itself to fill the special queue or array by a mechanism called jump pointers, which is then fed to software/hardware based prefetchers to get the next node of LDS, effectively reducing the memory access latency.

There is a fair amount of research that has been done to improve cache efficiency with adaptive cache replacement policies or predicting future accesses to evict the line (Hawkeye), but there is no fair comparison for their performance for LDS available. IMPICA paper by Hsieh et. al[1] talk about doing the pointer chasing inside the 3D stacked PIM core and only getting the final search result back to the program, this way it saves latency incurred by many intermediate accesses to memory back and forth from processor. That said, we couldn't find anything that discusses a cache replacement technique for linked data structures that could help in performing better for such structures, and thought about giving a fair try at developing the cache replacement policy without the overhead of prefetcher or PIM mechanism for the pointer chasing.

So the goal of the project is twofold:

1. Compare and analyze the existing cache replacement policies namely DIP [5], DRRIP[6] and Hawkeye[7] for dynamically constructed linked data structures and compare with Belady's OPT [8].
2. Try to overcome the shortcomings of existing policies with a better cache policy to accommodate pointer chasing.

OUR APPROACH

Our approach (we call it CaRP) is based on the observation that compiler generated assembly will have a limited range of instructions in which the accesses to linked data structure with high miss based loads are happening (Fig 2). We try to track these instruction addresses and try to place them in a dedicated victim cache to improve cache miss rate.

Pointer Chase Detection

If we consider a program which accesses a linked data structure with a piece of code for some form of traversal or access. The program is most likely to be written in iterative loop or recursion where the different records of data structure say linked list or tree would be referenced by traversing the reference to next data record. The compilation of such a code would result in assembly basic block which would have load instruction with some form of offset from a fixed memory location or from the address contained in a register. For a highly irregular memory access with little or no locality, would cause high number of misses on the particular instruction.

We consider the above property of data access in program execution as our invariant of pointer chase based memory access and try to exploit the behavior for placing such accesses in a victim cache. Consider an code snippet below for inorder traversal of a binary tree and its assembly code. The accesses to next node and data of a node in binary tree is translated in 'mov' instruction to load the data from some memory location pointed by register into a register. For a highly irregular memory access of node pointers, these 'mov' instructions will generate high number of cache misses. These 'mov' instructions can be traced by the program counter (PC) based on some heuristic for which this behaviour occurs and hence, can be used to have a special actions for such PC occurrences or insert those in victim cache in our case.

```
/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    int val = node->data;

    /* now recur on right child */
```

```
                                    printInorder(node*):
4006a655                             push rbp
4006a748 89 e5                       mov rbp,rsp
4006aa48 83 ec 20                    sub rsp,0x20
4006ae48 89 7d e8                    mov QWORD PTR [rbp-0x18],rdi
4006b248 83 7d e8 00                 cmp QWORD PTR [rbp-0x18],0x0
4006b774 2b                          je 4006e4 <printInorder(node*)+0x3e>
4006b948 8b 45 e8                    mov rax,QWORD PTR [rbp-0x18]
4006bd48 8b 40 08                    mov rax,QWORD PTR [rax+0x8]
4006c148 89 c7                       mov rdi,rax
4006c4e8 dd ff ff ff                 call 4006a6 <printInorder(node*)>
4006c948 8b 45 e8                    mov rax,QWORD PTR [rbp-0x18]
4006cd8b 00                          mov eax,DWORD PTR [rax]
4006cf89 45 fc                       mov DWORD PTR [rbp-0x4],eax
```

Figure 2. Example code and its assembly version for pointer chasing

Replacement Policy for Pointer Chasing

We augment the existing cache policies such as LRU, DRRIP, Hawkeye for pointer chasing type of workloads using the variant discussed above by adding a special LRU based fully associative victim cache.

For checking whether a particular data has a hit in the cache, we check the main cache (L2) tag array followed by victim cache array and declare cache hit if found in either one. For cache miss,

we call the predicate function $f_{predicate}$ in cache replacement policy with PC and line address of memory request as an arguments. The replacement policy decides whether to insert the given line in victim cache or the actual cache. The augmented cache replacement policy keeps track of the consecutive misses encountered for last unique 1024 instructions executed. If the number of consecutive misses on particular PC are greater than threshold T_H, then the particular miss is handled by the victim cache, or by the main cache otherwise.

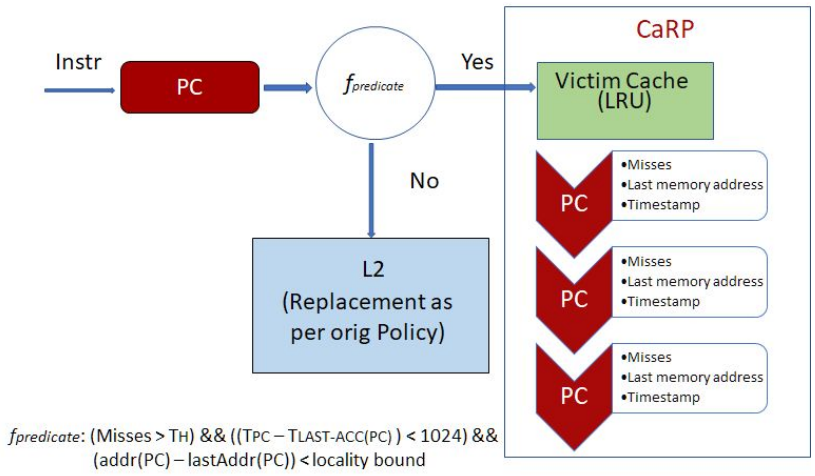

Figure 3. PC based tracking mechanism deployed in CaRP. The predicate function will decide whether to place next line in victim cache or in L2.

To ensure that the replacement policy doesn't capture the cold misses for the beginning of execution, we make sure that all the lines in the particular set to which the memory request maps to contains no unused lines.

We also make sure that for a given PC, if memory accesses are within some range and display some sort of locality, we put the next accesses to the same PC for which the miss is encountered in main cache rather than in victim cache. Since out victim cache array is very small (64 lines), this helps us saving the thrashing that might have happened if we had placed this sort of accesses in victim cache itself.

For deciding the value of threshold $T_{H,}$ we use three different approaches :

1. Static T_H value manually fitted to all benchmarks (A1).
2. Self adjusting workload based threshold where the threshold are adjusted based on the the number of misses encountered are greater or less than the threshold value capped at min and max values (A2).
3. T_H values always set at (misses_encountered - 1) where misses_encountered is the number of consecutive misses for the line last inserted in victim cache (A3).

METHODOLOGY

Simulation

We are using ZSim[3] simulator running on Ubuntu 12.04.2 (Precise Pangolin) to simulate the cache policies. ZSim uses Pin for dynamic binary instrumentation. Our baseline processor is single simple in-order core with three level cache hierarchy. Our cache hierarchy is roughly similar to that of Intel Core i7. The L1 instruction and data cache are 4-way 32 KB each. L2 and L3 cache are unified 8-way 256KB and 16-way 2MB respectively. We model 5, 12 and 35 cycle latency for L1, L2 and L3 cache respectively. All the caches use 64B line size. We target our evaluation and experimentation for cache policies on L2 cache with L1 cache replacement policy always kept LRU, since LRU for L1 is efficient and less expensive in terms of overhead and implementation. L3 cache is always kept LRU as well.

We first use trace generation mode of ZSim, where we generate trace files for accesses from L1 to L2 with maximum simulation time set to 12 minutes. This allows us to capture enough execution for the benchmarks being evaluated such that the L2 cache miss rates come to a steady state for majority of execution time (Fig 4). The traces captured in previous step are replayed on L2 cache with ZSim in trace driven mode with cache policy changed on L2 and hence, allowing us to compare the performance of different cache policies on different benchmarks. Trace driven execution becomes useful in getting theoretical minimum miss rates possible using Belady's MIN (OPT) where we require the knowledge of future to do optimal cache replacement.

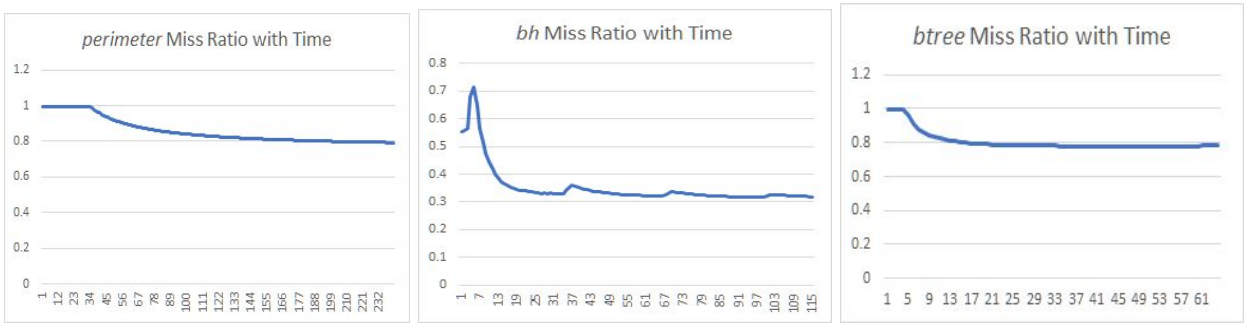

Figure 4. Showing the miss ratio for perimeter, bh, and btree benchmarks reaching to steady state after initial cache warm up

Benchmark

We use Olden benchmark [9] to evaluate performance for LDS. The Olden benchmarks are a collection of programs that includes small and medium sized scientific codes (bh and em3d), process simulations (power), graph optimization routines (tsp), graphics utilities (perimeter and voronoi) and a toy tree benchmark (treeadd). Additionally, we use btree and llu benchmarks from the IMPICA paper.

Benchmark	Data Organisation	Parameters
BH	Heterogenous OcTree	100000 bodies, 32 nodes
Em3D	Single Linked Lists	2000 H 500 E Nodes
Perimeter	Quad-Tree	11 levels
Power	N-way Tree, single-linekd Lists	N/A
TreeAdd	Binary-Tree	25 levels
Tsp	Balanced binary-tree	2 Million Nodes
Voronoi	Balanced binary-tree	1 Milllion Nodes
Simple Tree	Binary Tree	100 levels
B-Tree	B+ Tree	3 Million Nodes

Table 1: Description of benchmarks used for the project [10]

Results

We implemented and evaluated cache policies i.e. MIN, DIP, DRRIP, Hawkeye on ZSim. We first describe the comparison between various cache policies and then compare the performance of our approach by implementing it with LRU, DRRIP and Hawkeye.

Comparison Between Existing Cache Policies

Fig 5 shows the performance of different cache policies with respect to OPT. Majority of benchmarks perform better in terms of miss ratio on DRRIP. We see that LRU on an average performs worse by 20.97% maximum wrt OPT on power benchmark. This difference is drastically reduced to approx 1.77% by use of DRRIP. The benchmarks on which DRRIP performs worse than LRU are graphics utilities benchmarks i.e. voronoi and perimeter by 4.3% and 1.7% respectively. DIP policy shows behavior similar to LRU on multiple benchmarks, but helps in reducing miss rate on benchmarks such as bh (5.8% improvement), em3d (4.8% improvement) and tsp (3% improvement). We found the behavior of Hawkeye to be quite erratic and it sometimes performs very well on few of the benchmarks or goes to very bad performance. For example, Hawkeye exhibits 81.8% miss rate on perimeter, which is worst of all policies including LRU. On the other hand, it performs best with power benchmark with miss rate of 75.16%. However, it would be better to use DRRIP which gives miss rate of 75.43% on power, with much regular and better improvements on other workloads.

Overall, the performance of DRRIP was found to be consistent and very well on multiple benchmarks. The difference between DRRIP and OPT remains at average of 5.08% except the performance on voronoi as discussed above. The results give us a pretty tight bounds of improvement for cache policies with respect to OPT.

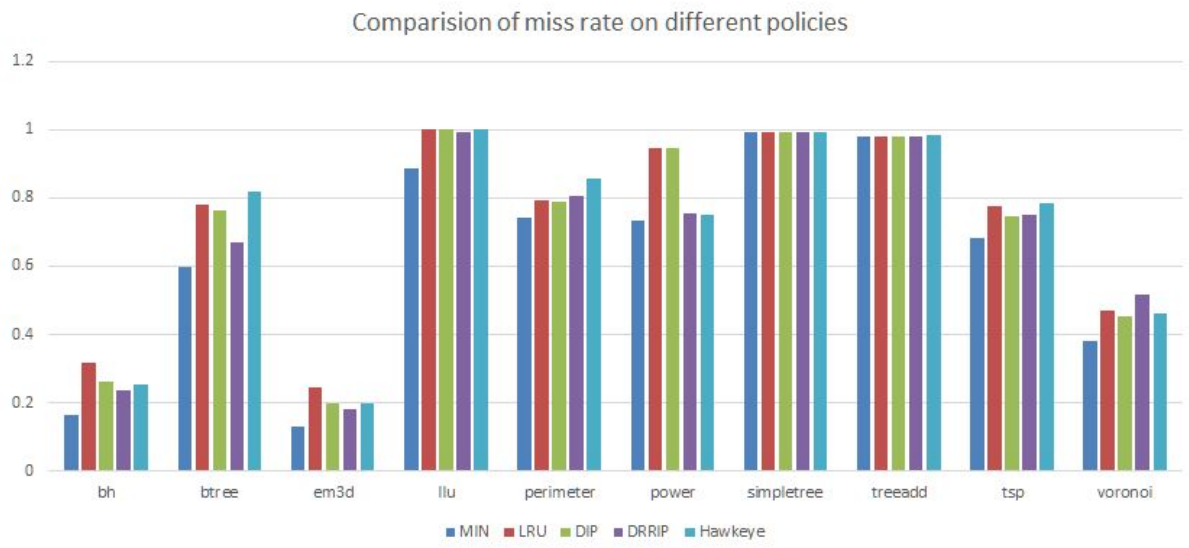

Figure 5: Comparing miss rate with different policies for linked-data structure driven workload.

Comparison of CaRP with Existing Policies

We implemented a new strategy by extending the target cache policy (such as LRU, DRRIP, Hawkeye) with PC based predictor which tracks the consecutive misses on a particular PC with the last memory access address and timestamp on the particular instruction. If the newer memory address maps to conflicting memory address, and the number of consecutive misses on corresponding PC are greater than a threshold, then the particular miss is handled by putting new access in 64 line wide fully associative victim cache. The value of threshold is decided using previously discussed 3 approaches. We use the threshold value of 4 as static threshold value.

Figure 6, 7 and 8 below show the comparison of LRU, DRRIP and Hawkeye respectively with the three approaches for adjusting threshold. In the second approach we are dynamically adjusting the threshold value .i.e if threshold is less than the recorded consecutive misses on given PC, it is increased by 1 and decreased by 1 in the same manner with threshold value capped with (MIN, MAX) = (3,25) with initial threshold value of 5. We found that static threshold value and dynamic threshold value perform moreover the same across multiple policies. But, we see some slight improvements in perimeter, voronoi, em3d and llu.

With respect to original cache policies, we found btree and power to be only benchmark performing significantly better by 4% and 9% respectively. We see very less improvements(less than 1%) on bh, em3d and llu. Interestingly, the power benchmark tends to approximate the performance equivalent to MIN with the help victim cache (less than 0.5% difference to MIN). In our intermediate results during poster session, we were getting majority of workloads to be performing very bad with victim cache, which we solved using intuition of using the last accessed memory address on PC to improve the overall behavior.

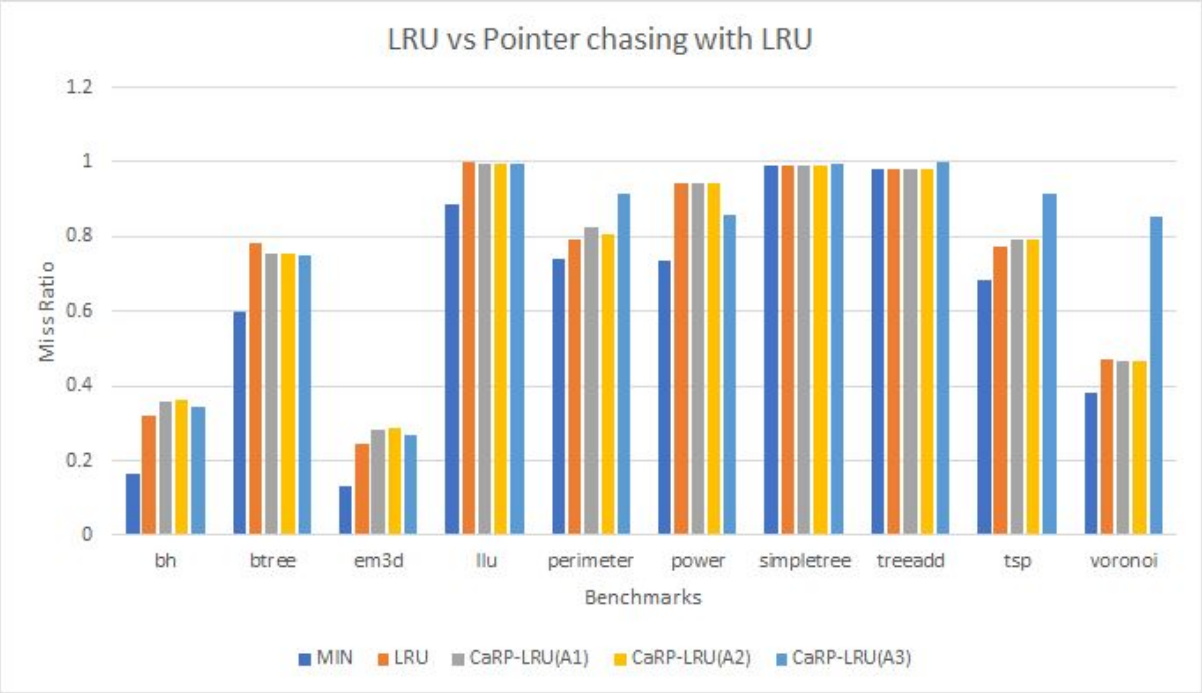

Figure 6: Comparing miss rate with LRU and MIN against the first approach(static value across all benchmarks) for T_H manipulation.

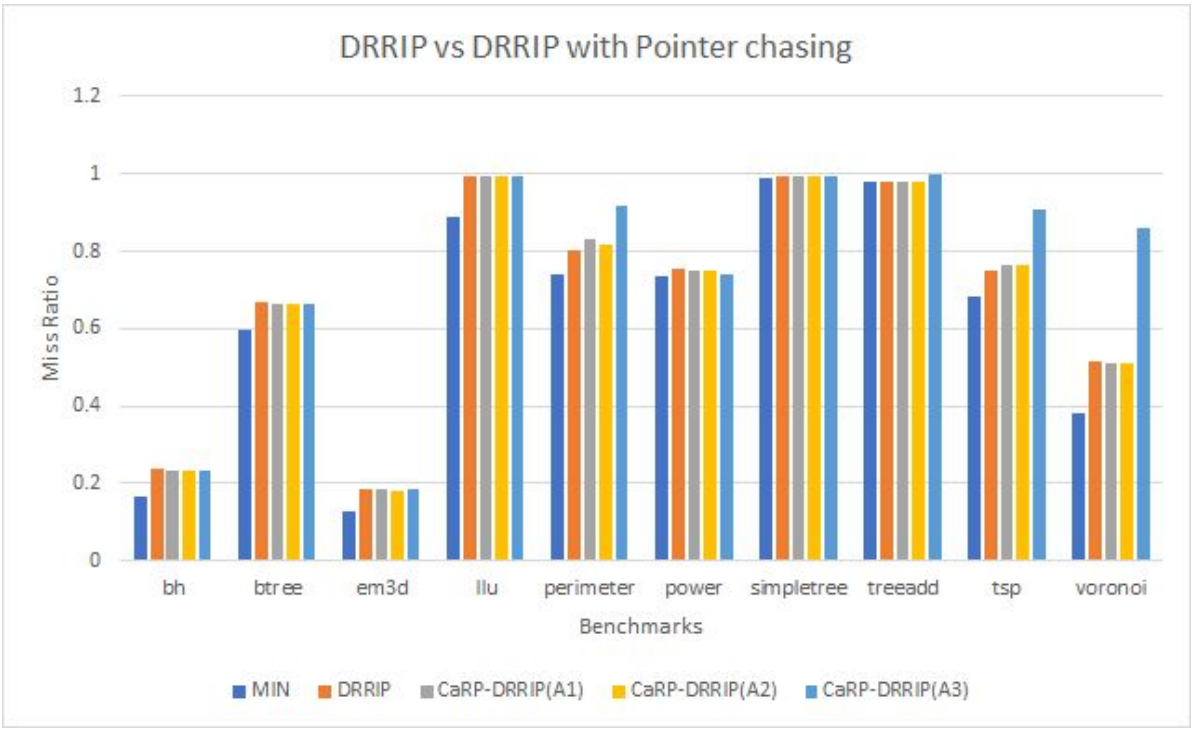

Figure 7: Comparing miss rate with DRRIP and MIN against the second approach(self-adjusting value across all benchmarks) for T_H manipulation.

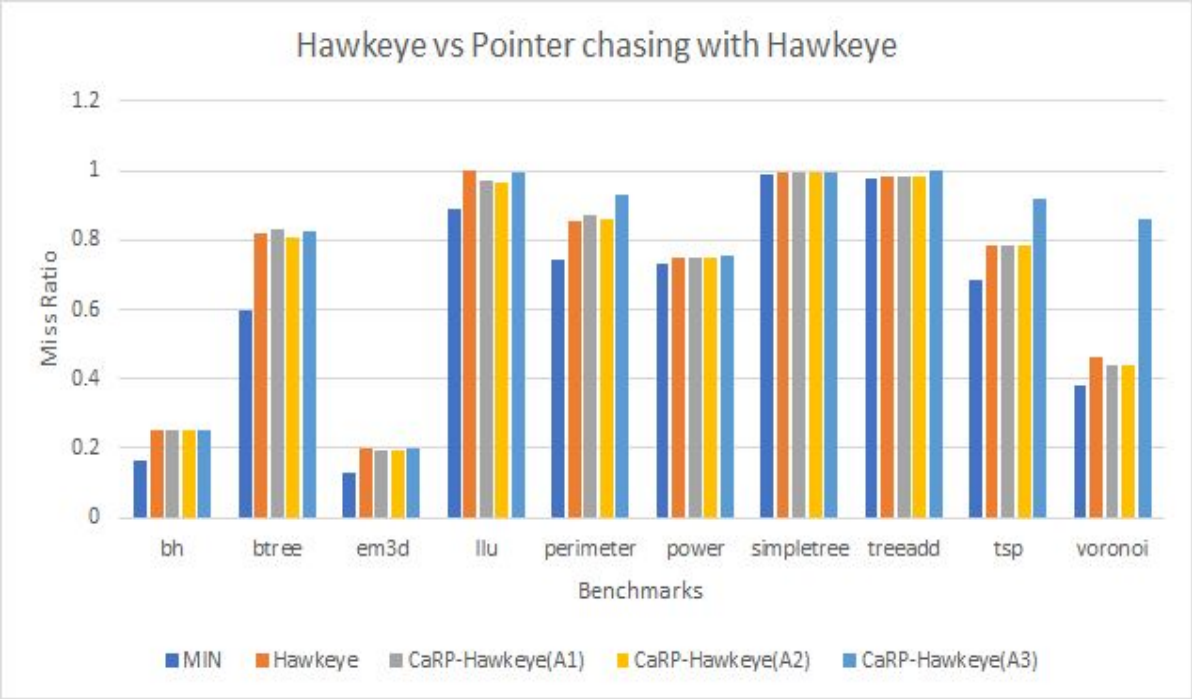

Figure 8: Comparing miss rate with Hawkeye and MIN against the third approach(self-adjusting value with T_H dependent on misses incurred in the previous run across all benchmarks) for T_H manipulation.

In the third approach, the threshold value to detect pointer chasing is dependent on the number of misses incurred in the previous run. (T_H < misses; T_H = misses-1). This approach however performed similar or worse for most of the benchmarks. This could be due to the ad-hoc behaviour between the previous and the next run. The fact that we simply set the threshold value according to previous behavior might have been too aggressive and hence, even worsened the original cache policy performance.

CONCLUSION

It is seen that even though LRU is very popular policy employed in modern caches, other policies perform better than LRU when it comes to pointer chasing benchmarks. It has been proven in [5], [6] and [7] that these policies also perform better than LRU on general class of workloads altogether. We found that there is a very little scope of improvement (6-7%) when it comes to pointer chasing benchmarks. Hence, it would be better to use much better strategy such as DRRIP to get better miss rates on L2 cache.

We tried to improve these by detecting pointer chasing behavior by exploiting how memory access happen during program execution, by tracking consecutive misses on a particular PC and declaring it as pointer chasing instruction if the number goes above a certain threshold. We saw a little improvement on btree and power benchmarks, very minor improvement in em3d, bh, and llu. However, we did not see any significant improvement overall behavior.

We think that the 6-7% bound of possible performance improvement imposed by MIN is very high and conservative to find any significant improvement for our particular scenario and come up with one fits-all solution. If one wants to improve any further, they might have to do very specific policy with respect to type of data structure and its access parameters.

REFERENCES

[1] K. Hsieh *et al.*, "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016.

[2] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 115–126, 1998.

[3] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," *ACM SIGPLAN Notices*, vol. 31, no. 9, pp. 222–233, 1996.

[4] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2, pp. 111–121, 1999.

[5] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, p. 381, 2007.

[6] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, p. 60, 2010.

[7] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[8] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[9] M. C. Carlisle and A. Rogers, "Supporting Dynamic Data Structures with Olden," in *Lecture Notes in Computer Science*, 2001, pp. 709–749.

[10] "Olden benchmarks simulations." [Online]. Available: https://www.irisa.fr/caps/people/truong/M2COct99/Benchmarks/Olden/Welcome.html. [Accessed: 11-May-2018].