Enhancement of Cache Performance by reducing the Last Level Cache Pollution

Project for Computer architecture (*EEL 5764*)

Gaurav Yeole
Dept. Of Electrical and Computer Engineering
University Of Florida
Gainesville, Florida, USA
gauravyeole@ufl.edu

Shahbaaz Ahamad Shafeeq Ahamad
Dept. Of Electrical and Computer Engineering
University Of Florida
Gainesville, Florida, USA
sahamad@ufl.edu

Vishal Suresh
Dept. Of Electrical and Computer Engineering
University Of Florida
Gainesville, Florida, USA
vishasure@ufl.edu

Abstract - A powerful last-level cache (LLC) significantly affects the general execution of the processor. In cutting edge processors LLCs are not exceptionally beneficial as they experience the ill effects of low hit rates created by the huge segment of cache-blocks that stay in the last level till execution, in the wake of being brought into the LLC when a cache miss happens. These blocks instigate cache contamination and have no commitment in the store hits. . This paper exhibits a compelling substitution algorithm that exchanges amongst LRU and SCIP to choose which block must be replaced. That is, we utilize bypass buffer found by SCIP for replacement and in the event that there are no polluted blocks in LLC, we utilize victim cache given by LRU calculation. Our proposed algorithm is evaluated utilizing an intensive simulation environment where its adequacy and performance-improvement abilities are shown.

I. INTRODUCTION

Cache pollution depicts circumstances where an executing computer program loads information into CPU cache pointlessly, consequently bringing about other valuable information to be ousted from the cache into lower levels of the memory hierarchy, deteriorating performance. The conduct of cache decides system performance because of its capacity to connect the pace between the processor and principle memory. After blocks are being brought into the last level cache, vast portions of cache blocks don’t get re-accessed. These blocks incite cache pollution and thrashing. This paper displays a effective cache management algorithm which enhances the performance of the selective cache insertion and bypassing algorithm to further build the hit rate and alleviate the cache pollution.

Portion of Last level cache blocks never get re-accessed to while living in the cache between the time the block is brought into the cache and the time it is expelled from the cache. As such, these never get re-accessed to or don't get any hits while in the Last Level Cache. Amid this time these blocks superfluously fill profitable cache space and cause cache pollution and whipping on the off chance that they had supplanted more helpful cache blocks while they were brought into the LLC. This is called cache pollution. The time that the polluted blocks stays in the LLC before being replaced increments enormously with the expansion in the cache sizes as the blocks need to spread a more extended path down to the least recently used position. Albeit bigger cache associativity enhances cache performance, it prompts to specific disadvantages. Consequently it is important to reduce the cache pollution.

By viably replacing never re-accessed blocks that occupy helpful cache space sooner than LRU does we can diminish cache pollution in the LLC. However we can reduce pollution just partially in light of the fact that the blocks as of now cause cache pollution because of replacing there blocks that may be required even now. This paper proposes a thought of using cache priority depending on its bypass buffer value. The bypass buffer value recognizes the blocks that are anticipated never to be re-gotten to while in the LLC and are given a low priority depending on the value of their bypass buffer and are not inserted into the LLC on a miss if present in higher level. Along these lines we will utilize the cache hierarchy capacity in a compelling way by keeping the blocks that are well on the way to be re-accessed.

II. RELATED WORK

Many techniques have been implemented to mitigate the cache pollution in Last level cache blocks. Some of them are: Data Valid Tag Splitting, Evicted Address Filter and Explicit Non-reusable Page Cache Management. However, there are certain limitations as mentioned below. Data Valid Tag Splitting technique [2] also known as Pease technique limits the polluting cache blocks to the blocks that are never accessed. In this approach, valid-bit in data cache tag is split into two bits as reading data valid bit (RVB) and writing data valid bit (WVB) and accordingly accessing strategies are applied to data cache. However, the cache blocks, which are accessed just once also contribute to the cache pollution. In the Evicted Address Filter (EAF) technique [3], cache is implemented using a bloom filter and a counter. EAF [4] only operates on a cache miss without changing the cache hit operation. If blocks having high reuse are evicted prematurely from the cache, such blocks are usually accessed soon after eviction. If missed block address is present in the EAF, such block is predicted as high reuse while other blocks are predicted as low reuse. However EAF has a significant storage overhead when compared to other prior techniques like re-reference interval prediction and adaptive insertion policy [5] for managing shared caches. In the Explicit Non-reusable Page Cache Management technique, the users need to explicitly specify the non-reusable page caches. SCIP maintains a small history table (known as bypass-buffer) which stores the history of access for each individual block in their previous life generations and the block is inserted or bypassed according to the access times and pollution block. Also, NRU is used for replacement. There might be pollution miss-prediction possible here.

SCIP algorithm	
On an LLC hit on Block B	B.used =1
On an LLC miss on Block B	if(bypassBuf[B] < 3)
	{
	bypassBuf[B]++;
	doBypass(B);
	}
	else{
	find a victim block V using NRU;
	bypassBuf[V] = V.u_bit*3;
	doInsert(B);
	B.u_bit = 0;
	}

Table 1: SCIP Algorithm

III. APPROACH

The basic idea behind this algorithm is to accommodate the cache blocks with a short access time in lower level cache L1. In this project, the concept of bypass-buffer from SCIP is used to divide the cache blocks into three categories based on their access time i.e., short, long or medium access time. Also, polluted bit is associated to each block. It is set to 1 or called polluted if it has low priority i.e., if the block is predicted to have short/long access time. Similarly, it is set to 0 or called non-polluted if it has high priority i.e., the block is predicted to have medium access time. Then, during replacement the polluted blocks are replaced and if no such blocks are available, then LRU is called. Hence, in this way only medium access blocks remain in LLC thus mitigating cache and improving the cache performance in the form of miss rates.

We have achieved a 3 level cache by altering the sim-cache.c file in simplesim-3.0 to analyze the cache miss rates in the LLC (L3 in this case). The standard code implements only two levels of cache. But, having three levels of cache would facilitate us to analyze the cache pollution in the LLC for different benchmarks.

Level1 Instruction	- 32KB	- 64B lines
Level1 Data	- 32KB	- 64B lines
Level2 Instruction	- 64KB	- 128B lines
Level2 Data	- 64KB	- 128B lines
LLC (unified cache)	- 64KB	- 256B lines

Flowcharts and algorithm for our project are as follows:

Proposed algorithm	

On an LLC hit on Block B	if (bypassBuf[B] > 3) {
	if (B.p_bit == 0) {
	B.used = 1;
	}
	else {
	B.p_bit = 0;
	}
	else {
	bypassBuf[B]++;
	}

On an LLC miss on Block B	If (LLC full) {
	
	Find Victim block V using LRU;
	bypassBuf[V] = V.u_bit*3;
	}
	else {
	if (bypassBuf[B] > 3 {
	B.p_bit = 0;
	}
	else {
	B.p_bit = 1;
	bypassBuf[B]++;
	}
	}

Table 2: Proposed Algorithm

1. LRU replacement algorithm is called only if there is no polluted cache in the block. Else, replace the polluted block. To know whether a block is polluted or not, p_bit is created. p_bit is 0 if not polluted.
2. In the same way u_bit is used to indicate whether the block was accessed when the block is presented in the cache.

3. Update p_bit to 0 0r 1 and u-bit to 0 or 1 on a miss or hit by using the following algorithm.

When there is miss on block B in LLC:

I. If LLC is full i.e., if all the blocks are polluted, find victim block V using LRU and set the count for block V in bypassbuffer to 3.

II. Else,

A. If block B is accessed more than 3 times, i.e., bypassBuffer[B] > 3, set B.p_bit to 0 i.e., indicate that block B is not polluted.

B. Else, block B is regarded as polluted i.e., B.p_bit is set to 1 and the access counter for block B is incremented ie., bypassBuffer[B].

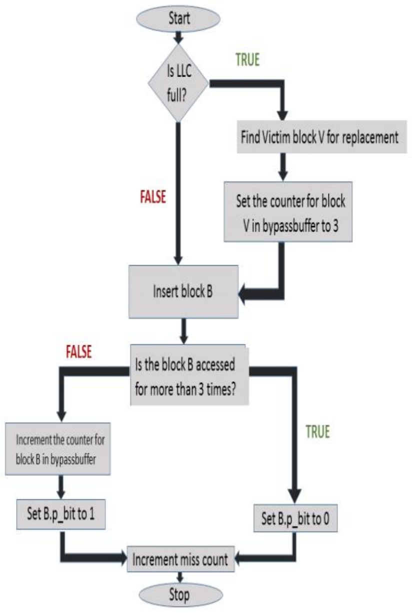

Start
Is LLC full?
TRUE
FALSE
Find Victim block V for replacement
Set the counter for block V in bypassbuffer to 3
Insert block B
FALSE
Is the block B accessed for more than 3 times?
TRUE
Increment the counter for block B in bypassbuffer
Set B.p_bit to 1
Set B.p_bit to 0
Increment miss count
Stop

Figure 1 Flowchart for miss occurence

When there is a hit on Block B in LLC:

I. If block B is accessed more than 3 times

a. If block B is polluted, set B.used=1.

b. Else, set B.polluted = 0 i.e., change and regard that block B is not polluted.

II. Else, increment the access count for block B in bypassbuffer i.e., increment bypassBuffer[B].

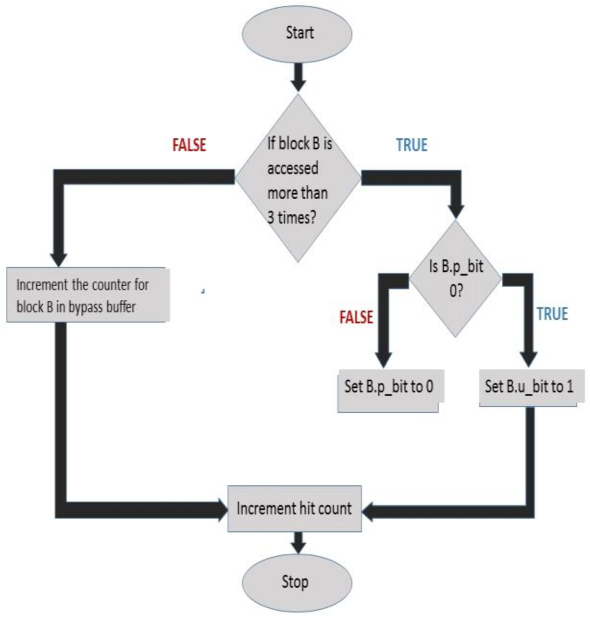

Start
FALSE
If block B is accessed more than 3 times?
TRUE
Increment the counter for block B in bypass buffer
Is B.p_bit 0?
FALSE
TRUE
Set B.p_bit to 0
Set B.u_bit to 1
Increment hit count
Stop

Figure 2 Flowchart for hit occurrence

IV. EVALUATION METHODOLOGY

Since we are actualizing a cache insertion and replacement algorithm, our principle center is around lessening the cache pollution in the last level cache. This straightforwardly thinks about the cache miss rates on the diverse cache levels. The CPU time relies on upon the hit rates and the miss rates. On the off chance that it is a cache hit then we can take the hit cycle to be one clock cycle. However on a miss the block must be brought from the memory. This will bring about slows down and the quantity of slow down cycles relies on upon the quantity of cache misses and the miss punishment.

Memory stall cycles = Memory accesses * Miss Rate * Miss Penalty

In order to evaluate overall CPU performance, the CPU base execution cycles has to be include in the equation.

CPU time = (CPU execution cycles + Memory stall cycles) *Cycle time (2)

Clearly, above equations tell us that we can evaluate the performance of the insertion and replacement algorithms based on cache miss rates and victim cache replacement rates.

V. EXPERIMENTAL SETUP

The SimpleScalar tool was used with the base configuration set as the PISA (Portable Instruction Set Architecture) that has 135 instructions with 4 instruction formats to carry out our experiment.

Cache Configuration: We used L1 Instruction cache which holds both instruction and data. The L1 instruction have 64KB cache size, 32 bytes line size, 4-way set associativity and follow Least Recently Used replacement policy. All the changes have been made with respect to LLC (Last Level Cache) i.e. L3 cache which supports 64 bytes line size and 16-way set associativity and is used to analyze LRU (least recently used) and the proposed algorithm.

Benchmarks: Four benchmarks namely go, ccl, compress and perl of the benchmark, SPEC95 are used for the analysis of miss rates for the aforementioned cache replacement policies. Each of these benchmarks are explained as below later.

go: This is an AI algorithm for playing game called Go. 2stone9 is input file for this benchmark. It runs for almost 550 million instructions.

cc1: This benchmark is basically C compiler, and it compiles the pre-processed input file, which produces an assembly file as output. It executes near about 120 million number of instructions.

Compress: This benchmark generates an in-memory buffer of data, compresses it to another in-memory buffer, then decompresses it. It uses the compression algorithm from an old UNIX utility of the same name. Its input file, *compress.in*, specifies how large a random buffer to generate and seeds the generation process. The version in the inputs directory operates on a 136000 byte file, and causes around 400 million instructions to execute.

perl: This benchmark is rudimentary perl interpreter. Perl scripts and their corresponding inputs are inputs of this benchmark.

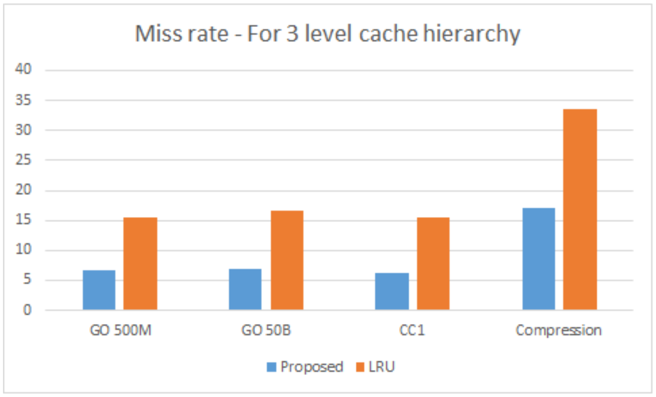
We have achieved a 3 level cache by altering the sim-cache.c file to analyze the cache miss rates in the LLC (L3 in this case). The standard code implements only two levels of cache. But, having three levels of cache would facilitate us to analyze the cache pollution in the LLC for different benchmarks.

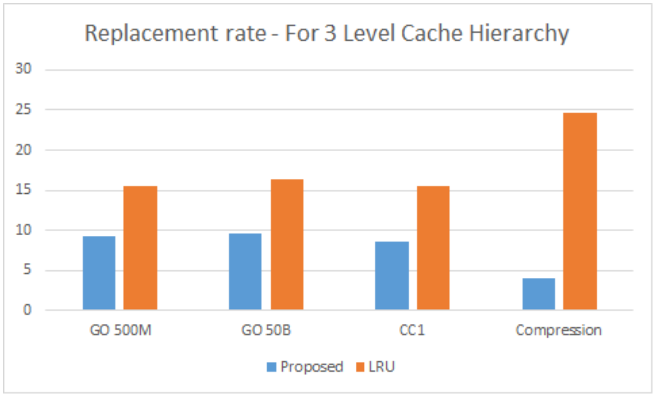
Cache Type	nsets	bsize	Associativity	Replacement Policy
L1 Instruction	64B Lines	32KB	4-way	LRU
L1 Data	64B Lines	32KB	4-way	LRU
L2 Instruction	128B Lines	64KB	8-way	LRU
L2 Data	128B Lines	64KB	8-way	LRU
LLC	256B Lines	64KB	16-way	LRU

Table 3

VI. RESULTS AND ANALYSIS

Based on miss rates of LRU replacement policy for specific benchmarks and cache configurations, we have analyzed the results as shown in Graphs below. LRU works fine when the working set is not larger than the instruction sets or application. LRU inefficiently utilizes the cache when newly inserted blocks have no temporal locality after insertion. We have run our algorithm for 4 different benchmarks from SPEC95 using LRU and the proposed replacement policy for different LLC cache sizes as shown below. On comparing, we observe that for benchmarks 'GO', 'CC1 and COMPRESSION the proposed algorithm has less miss rates than LRU. For the rest, the miss rates are equal for both. The reason for this might be because those benchmarks are not integers or floating points intensive.





VI. CONCLUSION AND FUTURE WORK

To condense our work, we have implemented a cache technique that uses a bypass buffer to monitor the number of times a block gets re-accessed to when in LLC. This data will choose whether a block must be embedded into the LLC with high priority or not. This strategy has been altogether assessed in the simulator and the outcomes are plotted. These outcomes exhibit the capacity of our algorithm to enhance cache execution and proficiency and demonstrate that our algorithm accomplishes good cache performance and efficiency. In future, this same benchmark can be tried for various benchmarks including SPEC2000. Proposed Algorithm can be adjusted for the multi-core system and its execution can be tried on shared LLC of multi-center system

VI. REFERENCES

[1] Jayesh Gaur, Mainak Chaudhuri & Sreenivas Subramoney "Bypass and Insertion level Algorithms for Exclusive Last-level Caches," ISCA 2011 Proceedings of 38th annual international symposium on Computer Architecture.

[2] Liu Song-He, Song Huan-Sheng, Qi Shu-Min, Zhang Jun; "Achieving Non-polluting Cache Accessing by Using Data Valid Tag Splitting" 2012 2nd International

Conference on Computer Science and Network Technology.

[3] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, & T. C. Mowry; "The evicted address filter: a unified mechanism to address both cache pollution and thrashing" in PenChung Yew; Sangyeun Cho; Luiz DeRose & David J. Lilja, ed., 'PACT', ACM, pp. 355366 (2012).

[4] Jongwon Kim; Jinkyu Jeong; Hwanju Kim; Joonwon Lee, "Explicit nonreusable page cache management to minimize last level cache pollution" Computer Sciences and Convergence Information Technology (ICCIT), 2011 6th International Conference.

[5] A.Jaleel, W.Hasenplaugh, M.Oureshi, J.Sebot, S.Steely, Jr., and J.Emer. "Adaptive Insertion Policies for managing shared caches" PACT '08 Proceedings of the 17th international conference on Parallel architectures and compilation techniques (2008).