209AS CA2

Cache Replacement - Nathan Portillo, Benjamin Cruz

[1]

***Abstract*—The aim of this paper is to introduce a methodology in creating a cache replacement system.**

I. Introduction

In the creation of this system, we tried a variety of different policies. SHIP performed best in the LOAD operations but failed to perform as well as basic LRU for Writebacks. Pseudo LRU and LFU also failed to perform for both LOAD and WRITEBACK operations. With these results in mind, we decided to keep the LRU because of its great WRITEBACK performance while attempting to combine it with another policy that may increase LOAD performance.

II. BIP

Because traditional LRU is not efficient for all types of workloads we introduce BIP to try and address multiple types. This BIP implementation is fairly standard, where we set a threshold of 0.1 to switch between inserting at MRU or LRU. The threshold for managing BIP was selected manually. Running a BIP implementation on its own the epsilon threshold changed results drastically from something like 0.1 -> 0.5. However, interestingly, once implemented within the hybrid policy, epsilon's effect becomes somewhat 'diluted'. Although the overall performance of the hybrid system increased, epsilon's effect on the entirety of the system decreased. It is possible that the workload may have high temporal locality, making the return diminishing. This was disappointing, as with the implementation of LRU and BIPtogether we expected to see strong performance for workloads of high temporal locality and streaming/weak temporal locality as each excels at those respectively.

III. Hybrid

The Hybrid system is a somewhat rudimentary approach in which we update the BIP and LRU cache states to start. Once updated, we begin to update a 3rd cache, referred to as the 'winner' cache. If there is a hit in BIP or LIP, we promote to MRU in our winner cache. If the line was not in LRU's MRU or in BIPs targeted position, we slowly degrade this line until it becomes a candidate for replacement. Another interesting development in the code was the difference in performance with the final 'winning' algorithm of the hybrid approach.

```
// Update Winner cache with hits from both LRU and BIP
for (uint32_t i = 0; i < LLC_WAYS; i++)
{
    // If there's a hit in either LRU or BIP, promote the way in the Winner cache
    if (lru[set][i] == 0 || bip[set][i] == 0)
    {
        winner[set][i] = 0; // Promote to MRU position
    }
    else
    {
        // Increment the position if the way is not promoted
        if (winner[set][i] < LLC_WAYS - 1)
        {
            winner[set][i]++;
        }
    }
```

Optimal Approach(Simple)

```
// Update ATDs based on hit/miss feedback
bool hit_x = (lru[set][way] == 0); // Simulate a hit for Policy X (LRU)
bool hit_y = (bip[set][way] == 0); // Simulate a hit for Policy Y (BIP)

atd_x[set] = hit_x;
atd_y[set] = hit_y;

// Adjust Saturating Counter based on ATD outcomes
if (hit_x && !hit_y) {
    sctr[set]++;
} else if (!hit_x && hit_y) {
    sctr[set]--;
}

// Promote in the Winner cache based on the policy selected by the saturating counter
uint32_t policy_winner = (sctr[set] >= 0) ? 0 : 1; // Select Policy-X if counter is non-negative, otherwise Policy-Y
if (policy_winner == 0) { // If Policy-X wins, use LRU's decision
    if (lru[set][way] == 0) {
        winner[set][way] = 0;
    }
} else { // If Policy-Y wins, use BIP's decision
    if (bip[set][way] == 0) {
        winner[set][way] = 0;
    }
}

// Demote all other ways if not promoted
for (uint32_t i = 0; i < LLC_WAYS; i++) {
    if (winner[set][i] != 0) {
        winner[set][i]++;
        if (winner[set][i] >= LLC_WAYS) {
            winner[set][i] = LLC_WAYS - 1;
        }
    }
}
```

Saturating Counter and Tag Approach

A similar trend to what we saw with many approaches, if we targeted LOAD operations, we would see significantly less WRITEBACK hits but when targeting WRITEBACK operations, we would see significantly less LOAD hits. This trend continued through with the two 'winning' approaches as well. The saturating counter tag approach yielded about **43 times more hits** on LOAD operations. However, we also saw about **10 times reduction** in writeback. Because of this, we opted with the 'Optimal Approach' which was much simpler but offered a slight improvement over the baseline. The Saturating Counter approach performed significantly worse than the baseline due to the reduced WRITEBACK hits.

IV. LIRS

The last approach we did was Low inter-reference recency set which is referred to as LIRS. According to some references, LIRS can perform better than LRU. It uses reuse distance as a metric for dynamically ranked accessed pages to make a replacement decision. LIRS also addresses the limits

[1]

of LRU by using recency to evaluate inter-reference recency to make a replacement decision. Overall, it was a very straightforward approach to implement. Our results with our LIRS implementation were very good and gave the best performance of 5.05 compared to 5.07 with the hybrid approach.

```cpp
//////////////////////////////////////////
//                                      //
//        LIRS replacement policy       //
//     Benji Cruz, benjicruz@g.ucla.edu //
//                                      //
//////////////////////////////////////////

#include "../inc/champsim_crc2.h"

#define NUM_CORE 1
#define LLC_SETS NUM_CORE*2048
#define LLC_WAYS 16
#define MAX_LIRS_STACK 9 // (LLC_WAYS + 1);

struct Block {
    bool valid;
    bool high_low; // true is high inter reference, false low interference
};

Block lirs_blocks[LLC_SETS][LLC_WAYS];
uint32_t lirs_stack[LLC_SETS][MAX_LIRS_STACK];

// initialize replacement state
void InitReplacementState()
{
    cout << "Initialize LIRS replacement state" << endl;
    for (int i = 0; i < LLC_SETS; i++) {
        for (int j = 0; j < LLC_WAYS; j++) {
            lirs_blocks[i][j].valid = false;
            lirs_blocks[i][j].high_low = false;
        }
        for (int k = 0; k < MAX_LIRS_STACK; k++) {
            lirs_stack[i][k] = UINT32_MAX; // Initialize stack with invalid indices
        }
    }
}
```

```cpp
// find replacement victim
// return value should be 0 ~ 15 or 16 (bypass)
uint32_t GetVictimInSet (uint32_t cpu, uint32_t set, const BLOCK *current_set, uint64_t PC, uint64_t paddr, uint32_t type)
{
    for (int i = 0; i < MAX_LIRS_STACK; i++) {
        if (lirs_stack[set][i] != UINT32_MAX && !lirs_blocks[set][lirs_stack[set][i]].high_low) {
            return lirs_stack[set][i];
        }
    }
    for (int i = 0; i < LLC_WAYS; i++) {
        if (!lirs_blocks[set][i].high_low) {
            return i;
        }
    }
    return lirs_stack[set][MAX_LIRS_STACK - 1];
}

// called on every cache hit and cache fill
void UpdateReplacementState (uint32_t cpu, uint32_t set, uint32_t way, uint64_t paddr, uint64_t PC, uint64_t victim_addr, uint32_t type,
    uint8_t hit)
{
    if (hit) {
        for (int i = 0; i < MAX_LIRS_STACK; i++) {
            if (lirs_stack[set][i] == way) {
                for (int j = i; j > 0; j--) {
                    lirs_stack[set][j] = lirs_stack[set][j - 1];
                }
                lirs_stack[set][0] = way;
                break;
            }
        }
    } else {
        uint32_t victim_way = GetVictimInSet(cpu, set, nullptr, PC, paddr, type);
        lirs_blocks[set][victim_way].valid = true;
        lirs_blocks[set][victim_way].high_low = true;
        for (int i = MAX_LIRS_STACK - 1; i > 0; i--) {
            lirs_stack[set][i] = lirs_stack[set][i - 1];
        }
        lirs_stack[set][0] = victim_way;
    }
}


// use this function to print out your own stats on every heartbeat
void PrintStats_Heartbeat(){}

// use this function to print out your own stats at the end of simulation
void PrintStats(){}
```

V. Results

Overall, we were able to increase the baseline performance from 5.11 to 5.05. Some of the policy implementations we tried offered significant increases in hits for LOAD operations but greatly reduced WRITEBACK hits. In an attempt to curb this issue, we implemented a hybrid approach that would excel at multiple types of workloads. However, with a ‘full hybrid’ approach, we saw similar issues in balanced LOAD and WRITEBACK operations. So, finally, we opted to simplify the system and allow the winner cache to prompt on LRU or BIP hits. This gave us a slight increase in LOAD hits for all workloads with a slight reduction in WRITEBACK hits. We saw that in certain traces, workloads in which LRU severely underperformed (for LOADS), BIP was able to increase the amount of hits by a decent margin while also retaining LRUs great WRITEBACK hit performance.