Cong Hui
CSCE 614 term project
Dec 8 2013

1.Abstract

Cache design has been an major topic of computer architecture world for decades, various solutions and designs have been proposed. One of the key factor that effects the overall performance of the cache greatly is the design of the L1 cache, usually it is a on-chip cache that is closet to the CPU and is expected to have the fastest speed in order to keep the CPU as busy as possible, thus maximize the utilization of it. There are several major implementations of the cache, E.g. Directed-mapped, n-way associate caches, etc. However, the design principle behind those is similar: taking advantage of the temporal and spacial locality, especially the temporal locality in L1 cache design plays a significant role. This paper is to propose a new design intended to utilize the temporal locality, similar to the classical LRU but with a significant accuracy improvement.

2.Introduction

Throughout years of computer architecture design, since the advent of the cache hierarchy, L1 cache design has been the focus of computer designer, it is closest to the CPU and has the highest access speed, thus has the higher priority when it comes down to improve the overall performance of the hierarchy. The key principle behind designing a cache memory is its ability to store those items that are access regularly. In other words, in principle, the size of the cache doesn't really matter as long as it always contain the

“correct” items for CPUs to access. Nevertheless, it is impractical in real cases because there is no feasible approach to “predict”, E.g loading correct values in cache. Therefore, an obvious improvement could be done to cache is to enlarge the physical size of the cache. It works because it enhances the probability of storing the correct values. However, this approach only delays the process of solving the problem, not actually solving the problem since the physical space will be eventually run out. As a result, a clever mechanism that could correctly store those values in the limited cache space while swapping out those less useful values is needed. Least Recent Use (LRU) is a replacement algorithm intending to maximize the probability of keeping the the most useful values in the cache. It takes advantage of the fact that there is a strong temporal correlation between items that are accessed in adjacent time, then it saves those items in the cache, hoping that they could be accessed again in near future. It seems to be a correct assumption that items are naturally closed in terms of access time, but failure comes when frequent items are sparsely located and are far apart. This leads to major drawback of the LRU cache design because if most of the frequent items are arranged this way, LRU is expected to perform poorly, E.g. Low hit rate, because those frequent items could be squeezed out by less frequent items due to their sparsity. Therefore, this paper proposes a innovative approach to mitigate this problem, instead of using one queue to keep track of the access time of those

items in the cache, use an additional queue to keep track of those repetitive items are potentially far apart. Consequently, this design is to inherit the simplicity of LRU replacement algorithm while mitigating the major drawback it presents. The paper is organized as following: section 3 illustrates the architecture of the design and further talks about implementations in more details; section 4 analyzes the performance with metrics compared to other common implementations; section 5 concludes the project and discuss about future application for it.

3.1 Architecture

the two queues pseudo-LRU design is to replace the traditional LRU replacement algorithm when cache is running out of space. Like it sounds, the architecture involves using two queues to keep track of which cache block to be replaced. The key difference from LRU, E.g. single queue, is that it uses another queue to maintain those items that are hit repeatedly, figure 1. This prevents items that are spaced apart to not be wedged out by those seldom items. The replacement algorithm could be used on any multiple-way associative cache except for directly-mapped cache, because for directed-mapped cache, the replacement position is fixed and no algorithm is needed.

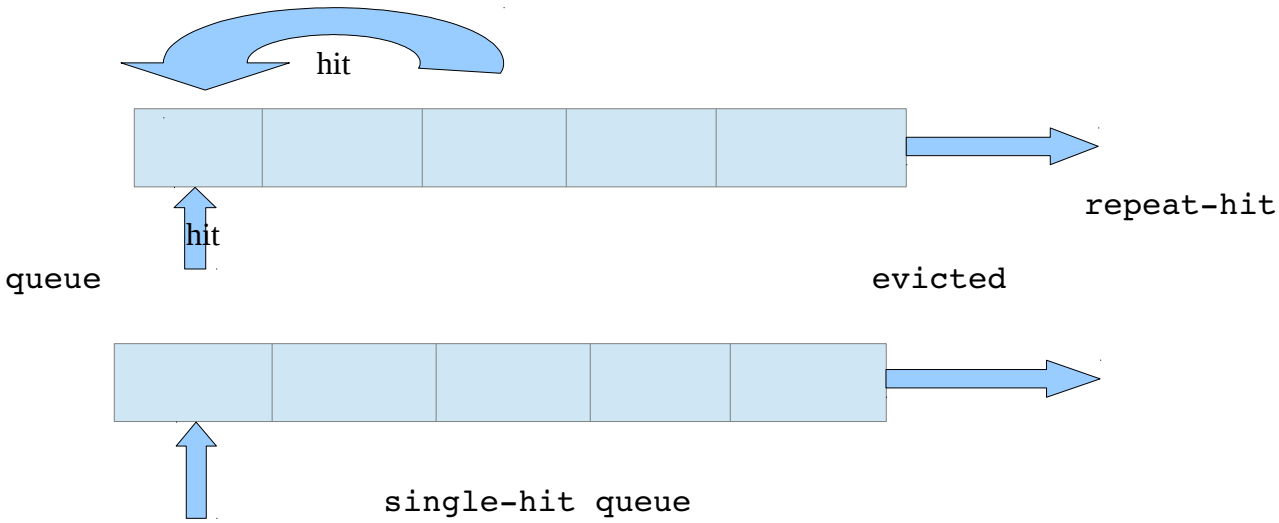

Figure 1

The work flow goes like following: every time a new cache block is brought into the cache, after a compulsory miss on starting up or some block was chosen to be swapped out, a pointer to the block is push into the single queue. If any items in the single queue is hit again, it was de-queued from the single queue and push into the repeat-hit queue. If an item in repeat-hit queue is hit again, it is inserted back to the head of the repeat-hit queue in order to prevent it from being swapped out. The eviction condition is met when either queue reaches up to its capacity and the tail of the queue is evicted. In other words, the least recent used item is swapped out and follow the pointer, the cache block could be overwritten to make room for new item, figure 2. Also another feature of this design is that in the repeat-hit queue behaves like a LRU queue: the most recent used cache blocks are kept in the leading positions of the

queue and have less likely to be swapped out soon. This is based on the temporal assumption that recent data blocks are likely to be accessed again.

3.2 Potential drawback

This proposed approach is solely based on the temporal correlation, neglecting any potential spatial correlation between access. For instance, for sequential access on a consecutive array, adjacent cells of the array should be brought into the cache to take advantage of this strong spatial correlation. However, in the design of this two queues cache, the philosophy is absent, therefore, for a sequential access, it shouldn't generate any satisfactory results, E.g. Consecutive misses.

Another flaw is the overhead of this design: it requires additional data structures to keep track of modification of the two queues. Even though it could be minimal. E.g. O(1)

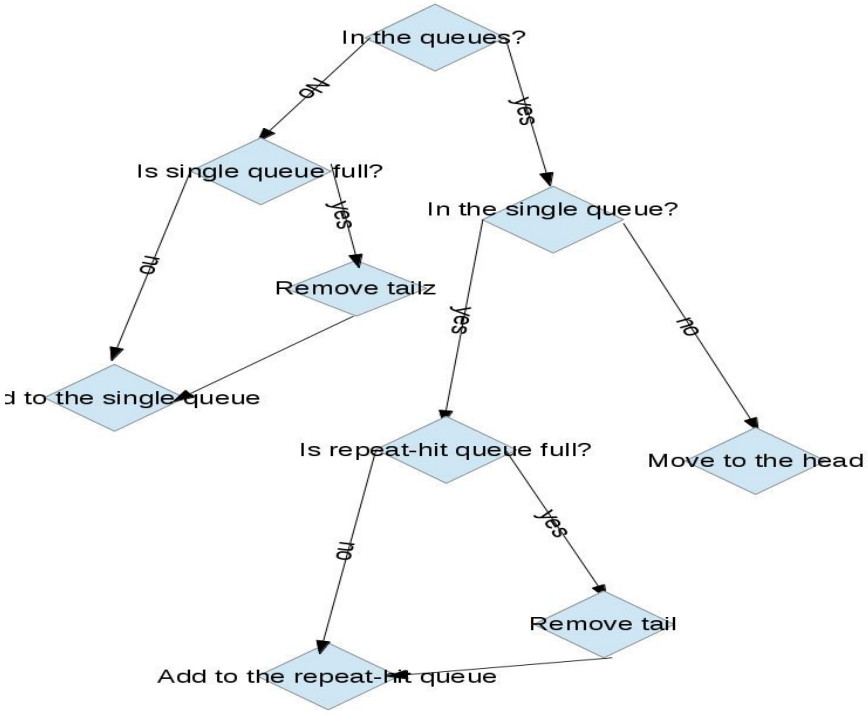

figure 2

operations on insertion and deletion from the queue, but a linear search on the repeat-hit queue is needed to find the node that should be inserted to the head. However, an additional hashtable could be used guarantee a O(1) access on those queue nodes, it incurs extra costs as well. Therefore, unless the performance gain with higher hit rates could compensate for the extra overheads, it should be considered carefully when to choose this implementation.

4. performance analysis

As an enhanced version of LRU, the 2Q implementation got better performance than a LRU implementation in terms of hit rates, especially when there are spaced accesses to the cache. In order to simulate different scenarios when real computers are at work. There were a few test cases I generated to mimic the real memory accesses: First, when accessing sequentially on a consecutive array and there are no duplicates addresses, LRU and 2Q implementation performed similarly bad. It was reasonable because all addresses are unique,which nullify the point of cache; Second, a sample test case where duplicates were presented was given as an input to the cache, where duplicates are relatively closed to each others. The feedback was quite positive where there were considerable hits in both implementations. However, at this point, the 2Q implementation didn't really shine compared to the standard LRU cache, because its capability to record spaced apart items wasn't highlighted. Therefore, a last test case was run to complement this: duplicates

were intentional placed apart and the returned results were quite promising: The 2Q implementation had significant higher hit rates than its rival's. Additionally, a random replacement algorithm was chosen to compare the results as well. Overall, the 2Q implementation had a better average performance.

LUR vs 2Q hit rate

	No duplicates	Adjacent duplicates	Sparse duplicates
LRU	0%	72%	75%
2Q	0%	74%	86%
Random	0%	61%	63%

Table 1

4.3 future improvement

Future improvements could be added such as adding more comparisons with other schemes, even further generalize the 2Q scheme to NQ schemes to see if it necessarily leads to a better performance. Also, in the paper, we were only comparing performance with hit rates, neglecting other factors like the real accesses time. Therefore, it is unclear that such higher hit rates could necessarily yield a better performance when more factors come into play: such performance gain could be offset by the additional overheads it generates and the complexity of the implementation.

5. Conclusion

L1 cache plays a major role in modern computers because it is

closest to the CPU and has the fastest access time in the memory hierarchy. Computers' overall performance heavily depend on the effectiveness of the L1 cache because it significantly shortens the access time to the main memory. Therefore, in theory, if there was infinitely large cache, the performance of computers could be greatly enhanced. However, it is impractical when there is a physical constraint on the size of the cache E.g. Number of transistors could be packed on chips and the cost of manufacturing high-speed cache. Therefore, a key improvement under those circumstances is to develop an replacement algorithm which maximize the probability that correct items are presented in the cache. Classical LRU algorithm has been around for decades and suffers from several major drawbacks: the inability to perform effectively when the inputs are spaced apart. Consequently, future improvement based on this schema was developed to mitigate this problem: 2Q replacement algorithm was developed to take advantage of the simplicity of LRU scheme while overcoming the shortcomings. An additional queue was added to keep track of those repeat-hit items that are placed apart. In contrast to its LUR cousin, it performed heuristically better while bearing minimal overheads.

Appendix

Reference

1.“The gradient-based cache partitioning algorithm” William Hasenplaugh, Pritpal S. Ahuja, Aamer Jaleel, Simon Steely Jr., Joel Emer
January 2012,Transactions on Architecture and Code Optimization (TACO) , Volume 8 Issue 4

2.“Dynamic access distance driven cache replacement” Min Feng, Chen Tian, Changhui Lin, Rajiv Gupta October 2011,Transactions on Architecture and Code Optimization (TACO) , Volume 8 Issue 3

3.“Reducing cache misses through programmable decoders” Chuanjun Zhang
January 2008 Transactions on Architecture and Code Optimization (TACO) , Volume 4 Issue 4

4. “Exploiting reuse locality on inclusive shared last-level caches” Jorge Albericio, Pablo Ibáñez, Víctor Viñals, Jose María Llabería
January 2013 Transactions on Architecture and Code Optimization (TACO) , Volume 9 Issue 4

5. Theodore Johnson, Dennis Shasha: “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm”. VLDB 1994: 439-450

Division of work

I've worked on this project alone throughout the semester.

Code snippet

Hash.h

```
#ifndef HASH_H
#define HASH_H

/*
 2q cache implementation
*/

typedef struct Node Node;
typedef struct TwoQ TwoQ;
typedef struct HashNode HashNode;

struct Node {
  Node* n;
  Node* p;
  int addr;
};

//f == 0 indicate the node is in the single queue,
//f == 1 in the double queue otherwise
struct HashNode {
  Node* node;
  HashNode* n;
```

```
  int f;
};

struct TwoQ {
  unsigned int sqs;
  unsigned int sqc;
  unsigned int dqs;
  unsigned int dqc;
  Node* shead;
  Node* stail;
  Node* dhead;
  Node* dtail;
  HashNode* hash;
  unsigned int hashsize;
};

static TwoQ* q;

unsigned int hit;

static int Tag (int addr);

void init (int n);
```

```
static HashNode* Find (int addr);

static void AddHead(Node* head,Node* node);

static Node* RemoveTail(Node* tail);

static void AddHash(HashNode* hashnode,int addr);

static HashNode* FindHash(int addr);

static Node* FindNode(Node* head,int addr);

static int TagMask;

static void setTagMask();

void Insert (int addr);

#endif
```

Hash.c

```
#include "Hash.h"
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
```

```
#define Hash(addr) (addr & (q->hashsize - 1))

#define Tag(addr) ((TagMask & addr))

void setTagMask()
{
  int c = 0;
  int tmp = q->hashsize;
  while (tmp >>= 1) {
    c++;
  }
  int i;
  int tag = 0x0;
  for (i = c; i < sizeof (int) * 8; i++) {
    tag |= (0x01 << i);
  }
  TagMask = tag;
}

void Init (int n)
{
  //n has to be a power of 2
  assert (! ( (n) & (n - 1) ) );
```

```
//init q
q = (TwoQ*) malloc (sizeof (TwoQ) );
q->sqc = n >> 1;
q->sqs = 0;
q->dqc = n >> 1;
q->dqs = 0;
//init single queue
q->shead = (Node*) malloc (sizeof (Node) );
q->stail = (Node*) malloc (sizeof (Node) );
q->shead->n = q->stail;
q->stail->p = q->shead;
q->stail->n = NULL;
//init double queue
q->dhead = (Node*) malloc (sizeof (Node) );
q->dtail = (Node*) malloc (sizeof (Node) );
q->dhead->n = q->dtail;
q->dtail->p = q->dhead;
q->dtail->n = NULL;
//init hash
q->hashsize = n;
q->hash = (HashNode*) malloc (n * sizeof (HashNode) );
int i = 0;
for (i = 0; i < n; i++) {
  q->hash[i].n = NULL;
```

```
  }
}

//find the block
HashNode* Find (int addr)
{
  int index = Hash (addr);
  HashNode* p = q->hash[index].n;
  while (p) {
    if (p->node->addr == addr)
      return p;
    p = p->n;
  }
  return NULL;
}

void Insert (int addr)
{
  HashNode* hashnode = Find (addr);
  if (hashnode) {
    hit++;
    //if it is in the multiple queue
    if (hashnode->f) {
      //put it to the head of the queue
```

```
  Node* p = FindNode (q->dhead, hashnode->node->addr);
  //remove the node
  Node* target = p->n;
  p->n = target->n;
  if (target->n)
    target->n->p = p;
  AddHead (q->dhead, target);
}
//in the single queue
else {
  //remove the node from the single queue
  Node* p = FindNode (q->shead, hashnode->node->addr);
  Node* target = p->n;
  p->n = target->n;
  if (target->n)
    target->n->p = p;
  q->sqs--;
  //remove the mulitple queue tail if necessary
  if (q->dqs >= q->dqc) {
    Node* tail = RemoveTail (q->dtail);
    q->dqs--;
    //update the hashtable
    HashNode* p = FindHash (tail->addr);
    HashNode* hash = p->n;
```

```
      assert (p->n);
      p->n = hash->n;
      free (tail);
      free (hash);
    }
    //bump into the head the multiple queue
    AddHead (q->dhead, target);
    q->dqs++;
    hashnode->f = 1;
  }
}
else {
  //remove the single queue tail
  if (q->sqs >= q->sqc) {
    Node* tail = RemoveTail (q->stail);
    q->sqs--;
    //update the hashtable
    HashNode* p = FindHash (tail->addr);
    HashNode* hash = p->n;
    p->n = hash->n;
    free (tail);
    free (hash);
  }
  //add to the single queue
```

```
    Node* node = (Node*) malloc (sizeof (Node) );
    node->addr = addr;
    q->sqs++;
    AddHead (q->shead, node);
    //update the hashtable
    HashNode* hashnode = (HashNode*) malloc (sizeof (HashNode) );
    hashnode->f = 0;
    hashnode->node = node;
    AddHash (hashnode, addr);
  }
}

void AddHead (Node* head, Node* node)
{
  //insert in the head
  Node* tmp = head->n;
  head->n = node;
  node->p = head;
  node->n = tmp;
  tmp->p = node;
}

void AddHash (HashNode* hashnode, int addr)
{
```

```
  int index = Hash (addr);
  HashNode* p = q->hash + index;
  while (p->n) {
    p = p->n;
  }
  p->n = hashnode;
  hashnode->n = NULL;
}

Node* RemoveTail (Node* tail)
{
  Node* tmp = tail->p->p;
  Node* target = tail->p;
  if(tmp)
    tmp->n = tail;
  tail->p = tmp;
  return target;
}

HashNode* FindHash (int addr)
{
  int index = Hash (addr);
  HashNode* p = q->hash + index;
  while (p->n && p->n->node->addr != addr ) {
```

```
    p = p->n;
  }
  return p;
}

Node* FindNode (Node* head, int addr)
{
  Node* p = head;
  while (p->n && p->n->addr != addr ) {
    p = p->n;
  }
  return p;
}
```