N-Way Associative Cache Documentation

Jade Chang

May 06, 2018

CONTENTS

Implementation of N-way Set-associative cache

author: Yu-Ju Chang

This module serves as a in-memory N-way Set-associative cache which user could use to store items(key and value pairs) and quickly access them. The type of the keys and values could be any type but all the keys and all the values should be the same type.

Set-associative cache is used in hardware implementation, yet here the module provides a software simulation of it. For details of Set-associative cache, see here:

https://en.wikipedia.org/wiki/Cache_Placement_Policies#Set_Associative_Cache

Input values: Several values are needed for using this cache module. They are cache_size, n_way, b, key_type, value_type.

cache_size (int): *cache_size* is the size of the cache we can use to save items. if cache size == n, it means we can store at most n items. cache size should be larger than n_ways * number of sets * offset size.

n_way(int): *n_way* is used to specify how many ways/lines in a cache set.

b(int): *b* is used to calculate offset size. offset size is 2^b.

key_type and value_type could be any type but all keys should be the same type and so do all values.

Following values are optional. Users are free to use.

replacement(`ReplacementPolicy`, optional): *replacement* is to set the cache replacement policy. User could either to pass a subclass of *ReplacementPolicy* or pass a string to specify the *LRU* or *MRU* policy. Default setting is *LRU*.

> For details about LRU, see here:
>
> https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU
>
> For details about MRU, see here:
>
> https://en.wikipedia.org/wiki/Cache_replacement_policies#Most_recently_used_(MRU)

hash(:func:, optional): *hash* is to provide the hash function that used to hash keys of the items. Default setting is to use python's built-in hash function.

thread_safe_mode(bool, optional): when *thread_safe_mode* == True, means the class is thread safe, One thing must be noted is that thread safe mode will have worse performance regrading of time since lock is costly. Default setting is True (enable thread safe mode).

Operations: The cache module provide three operations.

1. set_value(key, value): users could put items(key and value pairs) into the cache.

   The key will be hash with the provided hash function, and then get set number, offset index and tag from the hashed

   (a) if the one line has the same tag, replace/put the item into correspond offset directly

   (b) if full, choose one line to evict based on evict policy

   (c) if not full, place it into a new line and update the new line

   If there is any collision of key, then the old item will be replaced directly. Replacement policy object should be updated under all of the situations.

2. **get_value(key): users could get items(key and value pairs) from the cache.**

   The key will be hash with the provided hash function, and then get set number, offset index and tag from the hashed

   (a) if it is not in the cache, throw an exception

   (b) if in the cache, return the object and adjust current replacement policy object.

   Replacement policy object should be updated under all of the situations.

3. **delete(key, value): users could delete items(key and value pairs) from the cache.**

   Similiar to get_value and set_value, the key will be hash with the provided hash function, and then get set number, of

   (a) if the item not in the cache, do nothing

   (b) if in the cache, delete the object and adjust current replacement policy object.

   Replacement policy object should be updated under all of the situations.

Test: Please see cache_test.py to see the unit test code.

Usage: To use the module, start with following lines:

```python
import cache
cache_size, n_way, b = 16, 2, 2
key_type, value_type = int, int
#initialize the cache with your own setting
test_cache = cache.Cache(cache_size, n_way, b, key_type, value_type)
test_cache.set(1, 3)
value = test_cache.get(1)
test_cache.delete(1, 3)
```

Example use case: An in-memory cache could be used on an application server to store data associated with user id, so we could avoid to get data from database for every request.

CHAPTER ONE

OVERVIEW OF CLASSES

`Cache`(cache_size, n_way, b, key_type, value_type)	Cache class serves as a cache to store cache sets, each cache set will have cache lines to store items (a key & value pair).
`CacheSet`(n_way, offset_size[, replacement, . . .])	CacheSet class serves as a cache set in a cache to store cache lines, and each cache line will store items (a key & value pair).
`CacheLine`(offset_size[, tag, thread_safe_mode])	CacheLine class serves as a cache line in the cache.
`LRU_MRU`([policy, thread_safe_mode])	LRU_MRU class keep the accessed order and quick get the cache lines for the needs of LRU/MRU policy.
`ReplacementPolicy`	ReplacementPolicy class is an interface to allow user to inherit and implement their own replacement policy.
`Node`(tag, index)	Node class is a node used in implementation of doubly linked list.
`DoublyLinkedList`([thread_safe_mode])	DoublyLinkedList class is a doubly linkedlist used in implementation of LRU/MRU policy.

CHAPTER

TWO

CACHE

class cache.**Cache**(*cache_size*, *n_way*, *b*, *key_type*, *value_type*, *replacement=None*, *hash=<built-in function hash>*, *thread_safe_mode=True*)

Bases: `object`

Cache class serves as a cache to store cache sets, each cache set will have cache lines to store items (a key & value pair).

__init__(*cache_size*, *n_way*, *b*, *key_type*, *value_type*, *replacement=None*, *hash=<built-in function hash>*, *thread_safe_mode=True*)

The __init__ method of a cache is used to initialize a cache.

Args: cache_size (int): *cache_size* is the cache size we can use to save items. if cache size == n, it means we can store at most n items. cache size should be larger than n_ways * number of sets * * offset size.

n_way(int): *n_way* is used to specify how many ways/lines in a cache set.

b(int): *b* is used to calculate offset size. offset size is 2^b.

key_type(key_type): *key_type* is used to specifiy the key type of the item.

value_type(value_type): *value_type* is used to specifiy the key type of the item.

replacement(`ReplacementPolicy`, optional): *replacement* is to set the cache replacement policy. User could either to pass a subclass of *ReplacementPolicy* or pass a string to specify the *LRU* or *MRU* policy. Default setting is *LRU*.

hash(:func:, optional): *hash* is to provide the hash function that used to hash keys of the items. Default setting is to use python's built-in hash function.

thread_safe_mode(bool, optional): when *thread_safe_mode* == True, means the class is thread safe, One thing must be noted is that thread safe mode will have worse performance regrading of time since lock is costly. Default setting is True (enable thread safe mode).

delete(*key*, *value*)

delete is to delete the item which has the inputed key and value.

Args: key(key_type): *key* is the key of the item which is going to be deleted.

value(value_type): *value* is the value of the item which is going to be deleted.

Returns: if the value exist and be successfully deleted, return True; if not successfully deleted, return False; otherwise return None.

get_offset_index(*hash_result*)

get_offset_index is to get the index offset (which slot in offset) based on the hash result.

Args: hash_result(int): *hash_result* is the result of hash the key of the item.

Returns: an int to indicate which index in the offset the item should be in.

`get_set_num`(*hash_result*)
get_set_num is to get the set number (which set) based on the hash result.

Args: hash_result(int): *hash_result* is the result of hash the key of the item.

Returns: an int to indicate which set the item should be in.

`get_tag_num`(*hash_result*)
get_tag_num is to get the tag number based on the hash result.

Args: hash_result(int): *hash_result* is the result of hash the key of the item.

Returns: an int to indicate the tag of the item.

`get_value`(*key*)
get_value is to get an item(a key and value pair) from the cache by a key.

Args: key(key_type): *key* is the key of the item.

Returns: if the value exist, return the value of the key. Otherwise return None.

`is_valid_input`(*cache_size*, *n_way*, *total_sets*, *offset_size*, *b*)
is_valid_input is to check if the values are valid.

Args: cache_size (int): *cache_size* is the cache size we can use to save items. if cache size == n, it means we can store at most n items. cache size should be larger than n_ways * number of sets * offset size.

n_way(int): *n_way* is used to specify how many ways/lines in a cache set.

total_sets(int): *total_sets* is how many sets we should have.

offset_size(int): *offset_size* is how large the offset should be according to *b*.

b(int): *b* is used to calculate offset size. offset size is 2^b.

Returns: True if the inputs are valid, False otherwise.

`set_value`(*key*, *value*)
set_value is to put an item(a key and value pair) into the cache.

Args: key(key_type): *key* is the key of the item.

value(value_type): *value* is the value of the item

Returns: True if successful, None otherwise.

CACHESET

class cache.**CacheSet**(*n_way*, *offset_size*, *replacement='LRU'*, *thread_safe_mode=True*)

Bases: object

CacheSet class serves as a cache set in a cache to store cache lines, and each cache line will store items (a key & value pair). A cache might have more than one cache sets.

__init__(*n_way*, *offset_size*, *replacement='LRU'*, *thread_safe_mode=True*)

The __init__ method of a cache is used to initialize a cache set.

Args: n_way(int): *n_way* is used to specified how many ways(lines) in a cache set.

offset_size(int): *offset_size* is used to set offset size.

replacement(ReplacementPolicy, optional): *replacement* is to set the cache replacement policy. User could either to pass a subclass of *ReplacementPolicy* or pass a string to specify the *LRU* or'MRU' policy. Default setting is *LRU*.

thread_safe_mode(bool, optional): when *thread_safe_mode* == True, means the class is thread safe, One thing must be noted is that thread safe mode will have worse performance regrading of time since lock is costly. Default setting is True (enable thread safe mode).

delete_value(*tag*, *offset*, *value*)

delete_value is a function to delete the item in a cahce line which has the inputed key(trasferred to tag and offset) and value.

Args: tag(int): *tag* is the tag of the hashed item key.

offset(int): *offset* is the offset of the hashed item (in a cache line).

value(value_type): *value* is the value of the item which is going to be deleted.

Returns: if the value exist and be successfully deleted, return True; if not successfully deleted, return False; otherwise return None.

get_line(*tag*)

get_line is a function to get a cache line which has the same tag. By design, each tag is unique in one cache set

Args: tag(int): *tag* is the tag of the targeted cache line.

Returns: if the line exist, return the line. Otherwise return None.

get_value(*tag*, *offset*)

get_value is a function to get an item(a key and value pair) from the cache by a key(trasferred to tag and offset).

Args: tag(int): *tag* is the tag of the hashed item key.

offset(int): *offset* is the offset of the hashed item (in a cache line).

Returns: if the value exist, return the value of the key. Otherwise return None.

set (*value*, *tag*, *offset*)

set is a function to put an item(a key and value pair) into the cache line in a cache set. items in replacement policy will be updated accordingly. The hashed key is transferred to tag and offset to be used to find the item in the cache line. set will first go to find if there is any tag matches the tag of the key of item we are going to put into, if so, we will directly find the offset and put the item (cache hit);if there is no matching tag, then we will either fill the value into an empty cache line or evict a cache line if there is no empty cache line(cache miss). Note, when we evict a cache line, the whole line will be cleared and the tag will be replace with the new tag of the item we are goint to put into it.

Args: value(value_type): *key* is the key of the item.

tag(int): *tag* is the tag of the hashed item key.

offset(int): *offset* is the offset of the hashed item.

Returns: True if successful, None otherwise.

CHAPTER

FOUR

CACHELINE

class cache.**CacheLine**(*offset_size*, *tag=None*, *thread_safe_mode=True*)

Bases: object

CacheLine class serves as a cache line in the cache.

__init__(*offset_size*, *tag=None*, *thread_safe_mode=True*)

The __init__ method of a cache is used to initialize a cache line.

Args: offset_size(int): *offset_size* is used to set the offset size in a cache line.

tag(int, optional): *tag* is the tag of a line. When the line is empty, it is set to be None. Default value is None.

thread_safe_mode(bool, optional): when *thread_safe_mode* == True, means the class is thread safe, One thing must be noted is that thread safe mode will have worse performance regrading of time since lock is costly. Default setting is True (enable thread safe mode).

clearline()

clearline is a function to clear the whole line. It is handy when the whole line is needed to be evicted.

delete(*offset_index*, *value*)

delete is to delete the item which in the offset index and match the value.

Args: offset_index(int): *offset_index* is the index of the cache line offset where the item shoule be located at.

value(value_type): *value* is the value of the item which is going to be deleted.

Returns: if the value exist and be successfully deleted, and if the line become an empty line, then it also will needed remove from the LRU replacer, so return the tag of it; if the line isn't empty, then return None. if not successfully deleted, return False.

get(*offset_index*)

get is to get an item(a key and value pair) from the cache line.

Args: offset_index(int): *offset_index* is the index of the cache line offset where the item will be get from.

Returns: if the value exist, return the value of the key. Otherwise raise an error.

get_tag()

get_tag is a function to get the tag of the current cache line.

Returns: return the tag of the current cache line .

match_tag(*tag*)

match_tag is a function to see if a tag is the same as the tag of the current cache line.

Args: tag(int): *tag* is the tag of a line.

Returns: return True if a tag is the same as the tag of the current cache line. False Otherwise.

`set`(*offset_index*, *value*)
set is to put an item(a key and value pair) into the cache line.

Args: offset_index(int): *offset_index* is the index of the cache line offset where the item will be put into.

value(value_type): *value* is the value of the item

`set_tag`(*tag*)
set_tag is a function to set tag of the current cache line.

Args: tag(int): *tag* is the tag of a line.

CHAPTER

FIVE

REPLACEMENTPOLICY

class cache.**ReplacementPolicy**

Bases: `object`

ReplacementPolicy class is an interface to allow user to inherit and implement their own replacement policy.

delete(*tag*, *delete_result*)

delete is a function to update the replacement policy object after a value is ask to be deleted from a cache line. delete counts as an access, so the replacement policy needed to be updated too. If the line doesn't become empty, we update the order; if the line is empty then delete the whole line from replacement policy.

Args: tag(int): *tag* is the tag of the cache line.

delete_result(bool): *delete_result* is value we got after delete a value from the cache line. None means the line is non-empty after delete, True means the line became an empty line.

get_size()

get_size is a function to get the number of the items/cache lines in this object.

Returns: an int value of the size of the linked list and the hash table.

insert(*tag*, *cache_line_index*)

insert is a function to call when one item is accessed by user. When an item is accessed by users, we need to update the order of cahce lines (put it into the end of the linked list). If the item/cache line is not accessed before, we need to both update the list and the hash table.

Args: tag(int): *tag* is the tag of the line.

cache_line_index(int): *cache_line_index* is the index of the line. i.e. the index of the line in the cache set. It will be used to find the line faster when we need to evict/update the line.

victim()

victim is a function to choose the victim cache line to evict based on current replacement policy.

Returns: Return a tag and index i of the victim cache line. If there is no item in the linked list/hash table, return None.

CHAPTER

SIX

LRU_MRU (DEFAULT REPLACEMENTPOLICY)

class cache.**LRU_MRU**(*policy='LRU'*, *thread_safe_mode=True*)

Bases: `cache.ReplacementPolicy`

LRU_MRU class keep the accessed order and quick get the cache lines for the needs of LRU/MRU policy.

__init__(*policy='LRU'*, *thread_safe_mode=True*)

The __init__ method of a LRU/MRU replacement policy is used to initialize a LRU/MRU object. Default policy is LRU.

Args: policy(string, optional): *policy* is to set the replacement policy. User could pass a string to specify the *LRU* or *MRU* policy will be used. Default setting is *LRU*.

thread_safe_mode(bool, optional): when *thread_safe_mode* == True, means the class is thread safe, One thing must be noted is that thread safe mode will have worse performance regrading of time since lock is costly. Default setting is True (enable thread safe mode).

delete(*tag*, *delete_result*)

delete is a function to update the replacement policy object after a value is ask to be deleted from a cache line. delete counts as an access, so the replacement policy needed to be update too. If the line doesn't become empty, we update the order; if the line is empty then delete the whole line from replacement policy.

Args: tag(int): *tag* is the tag of the cache line.

delete_result(bool): *delete_result* is value we got after delete a value from the cache line. None means the line is non-empty after delete, True means the line became an empty line.

get_size()

get_size is a function to get the number of the items/cache lines in this object.

Returns: an int value of the size of the linked list and the hash table.

insert(*tag*, *i=0*)

insert is a function to call when one item is accessed by user. When an item is accessed by users, we need to update the order of cahce lines (put it into the end of the linked list). If the item/cache line is not accessed before, we need to both update the list and the hash table.

Args: tag(int): *tag* is the tag of the line.

i(int, optional): *i* the index of the line. i.e. the index of the line in the cache set. It will be used to find the line faster when we need to evict/update the line.

victim()

victim is a function to choose the victim cache line to evict based on current replacement policy.

Returns: Return a tag and index i of the victim cache line. If there is no item in the linked list/hash table, return None.

CHAPTER

SEVEN

NODE

class cache.**Node**(*tag*, *index*)

Bases: object

Node class is a node used in implementation of doubly linked list.

__init__(*tag*, *index*)

The __init__ method of a Node is used to initialize a Node. Here, a node represent a cache line and used to decide the evict order of cache lines.

tag(int): the tag of the cache line the node represents.

index(int): the index of the cache line in the cache set.

prev(node): the previous node of the current node.

next(node): the next node of the current node.

get_index()

get_index is a function to get the cache line index of the current node.

Returns: return a cache line index of the current node.

get_next()

get_next is a function to get the next node of the current node.

Returns: return a node which is the next node of the current node.

get_prev()

get_prev is a function to get the previous node of the current node.

Returns: return a node which is the previous node of the current node.

get_tag()

get_tag is a function to get the cache line tag of the current node.

Returns: return a cache line tag of the current node.

set_next(*next*)

set_next is a function to set the next node of the current node.

Args: next(*Node*): *next* is the node that will be the next node of the current node.

set_prev(*prev*)

set_prev is a function to set the previous node of the current node.

Args: prev(*Node*): *prev* is the node that will be the previous node of the current node.

CHAPTER

EIGHT

DOUBLYLINKEDLIST

class cache.**DoublyLinkedList**(*thread_safe_mode=True*)

Bases: `object`

DoublyLinkedList class is a doubly linkedlist used in implementation of LRU/MRU policy.

__init__(*thread_safe_mode=True*)

The __init__ method of a DoublyLinkedList is used to initialize a doubly linked list.

Args: thread_safe_mode(bool, optional): when *thread_safe_mode* == True, means the class is thread safe, One thing must be noted is that thread safe mode will have worse performance regrading of time since lock is costly. Default setting is True (enable thread safe mode).

get_head()

get_head is a function to get the head of the linked list.

Returns: an node which is head of the linked list.

get_tail()

get_tail is a function to get the tail of the linked list.

Returns: an node which is tail of the linked list.

insert(*node*)

insert is a function to insert a node into the tail of the doubly linked list.

Args: node(*Node*): *node* is the node that will be inserted into the list.

remove(*node*)

remove is a function to remove a node from the list.

Args: node(*Node*):*node* is the node that will be removed from the list.

Returns: return True when operation is done as expected, False otherwise.

PYTHON MODULE INDEX

INDEX