Cache simulation

Maguell Sandifort (3898903) Sam de Redelijkheid (4025458)

October 4, 2016

1 Implemented Features

1.1 N-Way Set Associative Cache

During initialisation, the full cachesize of a cache level is split up into $cSize/SLOTSIZE/NWAYN$ parts, called Parkinglots. Here $NWAYN$ determines the amount of potential slots per cacheline, by default set to 8. Each of these ParkingLots contain $NWAY$ cachelines and some extra eviction-data used for some of the eviction policies. Each cacheline contains $SLOTSIZE$ bytes of data, a memory address tag and an ltag used for the LRU eviction policy/ The first two bits of the adress tag are used to store the valid and dirty tags. Furthermore a cache has a READ and WRITE function and a ReadMiss function, to handle any cache misses.

1.1.1 READ

When reading from the cache, an address is provided. To find out what ParkingSlot the address belongs to a slotmask is used on the address. The slotmask is only takes the bits that are relevant to determine the ParkingSlot from an address and is defined as $(cSize/SLOTSIZE/NWAYN) - 1) << 6$. The bitshift skips the first 6 offset bits. The given ParkingSLot is then treated as a Fully Associative Cache. First each cacheline slot is checked to see if the address is already present in the cache and if it is, it is returned. If the address cannot be found, it is a cache miss, which is then handled in ReadMiss. In ReadMiss, the requested data is read from memory. Before it is returned, it needs to be stored in the cache. First is checked if a cacheline slot without a valid tag is found. If there is one, overwrite the cacheline slot with the requested data, tag it as valid and return the data. Otherwise, a slot must be evicted to make room for the new data. If the evicted slot has a dirty tag, then the old data in the slot is written to memory before overwritten by the new data and it will be tagged as valid. The new data is then returned.

1.1.2 WRITE

When writing to the cache, an address and the data to write is provided. First the READ method is called. This ensures that the address that needs to be written to will be placed on the stack. Then each cacheline slot in the ParkingSLot is checked until the slot with the required address is found. This slot is then overwritten with the new data and its dirty flag will be set, so if this slot is later evicted, its data will be written to memory first.

1.2 Eviction Policies

The following eviction policies have been added:

- Random
- First In First Out (FIFO)
- Bit-Pseudo Least Recently Used (BPLRU)
- Tree-Psuedo Least Recently Used (TPLRU)
- Modified Tree-Psuedo Least Recently Used (MPLRU)
- Least Recently Used (LRU)

It is possible to switch between policies by changing EVICTION in the cache.h header.

1.2.1 Random

The random policy is arguably the simpelest eviction policy, as any time a new cacheline needs to be added a random cacheline is removed and replaced by the new line.

1.2.2 FIFO

The FIFO algorithm is implemented using a pointer which holds the index of the most recently updated element. When a new cacheline needs to be added to the cache the pointer is increased by one, and the item at the new index is replaced by the new item.

1.2.3 Bit-PLRU

BPLRU approximates the LRU algorithm by saving a marker bit with each cacheline. Whenever a cacheline is used it's bit is set to 1, when an cacheline needs to be removed an item with it's bit set to 0 will be chosen. If all lines are set to 1, all lines are set to 0.

1.2.4 Tree-PLRU

TreePLRU approximates LRU by saving the lines in a tree. Each of the nodes holds one marker bit denoting whether the least recently used item is to the left or to the right. When an item is added you follow the bits in one direction and set all bits in the opposite direction. When an item is used all bits on the path towards it are set to point away from it.
In our implementation the nodes of the tree are saved in a list of bits, where the centre bit is the root of the tree, and it's children are always at the centre of the left or right half of the array.

1.2.5 MPLRU

Another caching algorithm we implemented is a modified version of the TPLRU algorithm based on the work of Ghasemzadeh et al. [1]. In this version each node that has four or more leaves below it is not updated directly. The algorithm keeps a separate copy of the tree for all nodes below which there are more than 2 leaves. When a call is done to the array the actual tree is updated with the values from the second tree while the spare tree is updated with the new values.

1.2.6 LRU

Finally regular LRU as described by Wikipedia [2] was implemented. Each cacheline holds an extra uint *ltag* which functions as age-bits. If a given address is read from or written to the cache, that Cacheline's *ltag* will be set to 0, while the *ltag* of the other CacheLines present in the same ParkingLot will be increased by one. When one of the CacheLines needs to be evicted, the slot with the highest tag will be selected for eviction. If the selected slot is tagged as dirty, the data will first be written to memory. Afterwards the data of the selected slot will be overwritten, its ltag set to 0 and the ltags of the other cachelines will be increased by one.

1.3 Cache Hierarchy

A cache hierarchy with 3 cache levels was implemented. The Cache and Memory classes have been made a subclass of the abstract MemCac class. Each Cache has a MemCac assigned to it. In game.cpp, 3 Cache objects are instatiated. Each is given its size, its level (for debug info) and its lower cache (L2 for L1, L3 for L2 and Memory for L3). Now, instead of reading from or writing to memory, a cache reads from or writes to its lower level MemCac (which could be another cache or memory, depending on its level).

1.4 Real-Time Visualisation

The program collects data on the caches performance during its run. Each tick, some new information is placed in the console. First the Total Memory Access cost is shown in 1000 cycles. After that, for each cache:

- Line 1: Level of the cache, total Hits and Losses this run for both read and write

- Line 2: Amount of read calls this tick, split into hits and misses. The misses are split up into evictions and cacheadds (overwriting an invalid slot, only happens when the cache is not filled yet)
- Line 3: Amount of write calls this tick, split into hits and misses. The misses are split up into evictions and cacheadds (overwriting an invalid slot, only happens when the cache is not filled yet)

Pressing any key will pause the cache simulation, so the information can be read easily. During the run a graph is also drawn. Each tick, the amount of hits for L1, L2 and L3 are measured, as well as MEM hits (L3 misses). These are used to determine the cycle cost of this tick. The graph shows the distribution of cycle costs this tick over L1, L2, L3 and mem hit costs. Light Green is used for the L1 costs, Dark Green for L2 costs, Yellow for L3 costs and Red for MEM costs. The blue dots indicate the total cycle costs of that tick, divided by 10000. The higher the dot, the higher the cost that tick. If a dot overlaps with one of the other (not Light Green) colors, it is made slightly brighter for readibility. If the graph reaches the end of the screen it will overwrite the existing graph, following a black bar. The console and graph keep updating until the program is done.

1.5 Larger Data Types

Support for larger datatypes is added in the form of the READB16, WRITEB16, READB32 and WRITEB32 methods, which perform read and write operations for 16 and 32 bit integers respectively. Write operations are implemented by extracting the consequtive bytes and entering them in order in the cacheline. Read operations reverse this process by merging the individual bytes back into one number.
Bytes part of a bigger data type enjoy no special status in the cache, hence it is perfectly possible to alter the first 8 bits of a 32 bit integer by writing to the first byte. Our implementation does not allow for big numbers to straddle multiple caches and will crache when this is done.

2 Results

We measured the performance for each eviction policy for different sized caches as can be seen in Table-1. Both Random and FIFO appear to be generally outperformed by the other algorithms, with FIFO performing the worse in all situations except for the largest cache, in which case the worse performing algorithm is Random.
For it's simplicity the BPLRU algorithm performs surprisingly well. While it has some trouble keeping up wth the other algorithms in the larger caches the algorithm climbs up to the second rank when space becomes scarce.
While holding the best rank with the biggest cache size MPLRU deteriorates quickly when less space is available.
TPLRU and PLRU seem to yield similar performance, both vying for the first place. PLRU does perform notably better in the smallest caches whereas TPLRU seems to perform slightly better when more space is available.

Table 1: Cost per algorithm on different sized caches

	n	$n/2$	$n/4$	$n/8$
Random	7002	7705	9952	10952
FIFO	6917	8265	10360	11345
BPLRU	6702	7973	8898	9162
TPLRU	6526	7374	8636	9433
MPLRU	6524	7536	9042	9803
PLRU	6556	7705	8610	8880

Table containing the cost for each algorithm to complete the task in thousand cycles for different cache sizes n. In the first column n is 8192, 16384 and 65536 bytes for the L1, L2 and L3 cache respectively. In each following column n is half the size of the column before.

3 Work Division

- Basic Cache: Both
- Random and LRU: Maguell
- Other Eviction Policies: Sam
- Cache Hierarchy: Maguell
- Real Time Visualisation: Maguell
- Larger Data Types: Sam
- Report: Both

References

[1] H. Ghasemzadeh, S. Sepideh Mazrouee, and M. R. Kakoee, "Modified pseudo lru replacement algorithm," in *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, ECBS '06, (Washington, DC, USA), pp. 368–376, IEEE Computer Society, 2006.

[2] Wikipedia, "Cache algorithms." https://en.wikipedia.org/wiki/Cache_algorithms.