Cache simulator
Optimization and vectorization

Žan Palčič
Faculty of Science, Information and Computing Sciences
Utrecht University

***Abstract*—This paper provides an overview of work done for the class assignment on the subject of cache hierarchy and its implementation. We outline the basic design, provide the overview of our cache architecture, present implemented functionalities and evaluate different eviction policies used in a cache hierarchy.**

I. Introduction

In today's computer systems the bottleneck to optimum performance is access to memory and transferring the data back and forth from CPU to RAM. According to Moore's law every 18 months, computer chip performance doubles, but this barely holds for CPUs and certainly not for the memory access. Hence, something needed to be done to overcome that physical distance between CPU and RAM, electricity constraints and various barriers, that unable us to physical query RAM fast enough to keep CPU busy. The solution that scientist came up with is that frequently used data must be near the CPU and thus enabling fast accesses – caches. The hierarchical architecture of memory becomes a standard that is unavoidable in today's modern computer systems.

Caches are computers memory components that are a part of the chip, silicon plate, of the CPU. It is realized in an SRAM technology, so values are static and don't have to be refreshed every n cycles which are expensive operations. Because of the small distance and it's advanced realization it is very fast, but also expensive.

In this paper, we will present our implementation of cache simulator, few eviction policies and argue about results.

II. Cache Architecture

Our cache simulator was designed with flexibility in mind. With easy-accessible defines in the header file you can choose how much levels will be in a cache hierarchy, define a size of each level, N-way set associativity, and for a cache as a whole, you choose eviction policy. With defines, you can also specify penalties to access each level, used for calculation of average memory access time (AMAT), enable real-time performance graph and also if you prefer, save results to $.csv$ file.

A. Associativity

Every time processor fetches data; a cache has to retrieve it from main memory and store it, cache it for the later use. Because caches are smaller compared to main memory, we expect many overlaps. If the replacement policy can freely choose where the data from RAM will be stored in the cache, it is called *fully associative* cache. This replacement policy is on hardware level impractical because it's time-consuming to check the tag of every cache slot. On the other hand, we can restrict each address from RAM to only one block in the cache and get maximum performance regarding speed, but on account of many overlaps,*direct mapped cache*[3]. The best solution is a compromise in which each entry in the main memory can go to N places in the cache. This replacement policy is called N-way set associative, and we also used it in our implementation.

B. Writing policy

Regarding writing to memory, we face two major problems in case we hit the data, and in the case we miss it.

Firstly, if we hit the data in a cache, data is present, the question emerges when to store it to lower memory level, now or maybe leave it for later and continue with work. There are two basic writing approaches; *Writing-back* and *Writing-through*. The following store's new data synchronously both to the cache and to the lower level memory, while *Write-through* policy writes only to current cache level, marks this entry as dirty and postpone storing to the lower level when this cache line needs to evict[3].

Secondly, in case of a miss, the question is, do we load data from a lower level or just write through and leave this cache level untouched. The first approach is called *No-write allocate* and as a name suggest there is no allocation of the data from a lower level and the new data is written directly to a lower level. On the contrary, *Write allocate* loads data to the current level and changes it accordingly like it was write hit.

We choose *Writing-back* approach with *Write allocate* policy.

C. Levels

Increasing size of a caches increases hit rate but likewise latency. To address this trade-off, multiple level cache hierarchy is presented. Each level down the hierarchy is larger and thus have higher hit rate and latency. The size, N-way set, and number of levels of cache depend mostly on the individual application, it's hardware implementation, etc. So here are many different approaches of different manufacturers.

As mentioned before, in our implementation we can choose a different number of levels for a cache hierarchy. On each level, we can set N-way associativity and it's s size.

III. Implemented functionalities

We supported all of the requested functionalities:

- A cache hierarchy (L1-L2-L3),
- various realistic eviction policies,
- real-time visualization of cache performance,
- wider data types.

In the header file *cache.h* user can choose from many defines to describe simulations behavior. With define *NUM_OF_LEVELS* we set number of levels of cache hierarchy, *LEVEL*[i]*_N_WAY_SET_ASSOCIATIVE* and *LEVEL*[i]*_SIZE* are corresponding to i-th level size and associativity. The only restriction is to set N-Way set associativity and size to the power of two.

Define *SUPPORT_WIDER_TYPES* enables using wider data types like 128 bit vectors and 64 bit pointers.

For eviction policy we can choose between *Random Replacement (RR)*, defined with *EP_RR*, Least Recently Used (LRU) - EP_LRU, *First In First Out (FIFO) - EP_FIFO* and *Most Recently Used (MRU) - EP_MRU*. Our decision must define *EVICTION_POLICY* with corresponding eviction policies defines (eg. *EP_LRU*).

For calculation of Average Memory Access Time, we define penalties for each level with defines such as *L*[i]*_PENALTY*, where i indicates level.

If we choose to see real-time performance per frame, *SHOW_PERFORMANCE* should be defined. For real-time graph also *SHOW_GRAPH*.

On each cache level, many counters are placed to measure performance. After each frame these few calculations take place. To show or select performance we must define *SELECTED_PERFORMANCE* to one of these defines:

- *PERFORMANCE_AMAT* - Average Memory Access Time,
- *LOCAL_MISS_RATE* - Local miss rate on individual cache level,
- *GLOBAL_MISS_RATE* - Global miss rate on individual cache level, miss rate in relation to all accesses to cache L1
- *READ_MISS* - number of all read misses,
- *WRITE_MISS* - number of all write misses,
- *N_ACCESSES* - number of access to current level.

We can also define few others defines such as *REAL_TIME_SCRHEIGHT*, to set blended performance screen hight, *SAVE_PERFORMANCE*, in case we want to save results to a *.csv* file and *INCLUDE_DRAM 1*, if we want to include a number of RAM accesses to real-time performance graph.

A. Functions

To support wider data types and also 64 bit version of simulator we redefined functions *Read32* and *Write32* to:

```
void Read(const uintptr_t address, const
    uint size, void* value);
void Write(const uintptr_t address, const
     uint size, const void* value);
```

With data type *uintptr_t* we defined pointer of appropriate size depending on compiled version. To use various data width with only one function, we set size in bytes and pointer to the value it should be written or to where the data from cache needs to be stored.

Functions

```
void LoadLine(const uintptr_t address,
    CacheLine& line);
void WriteLine(const uintptr_t address,
    const CacheLine& line);
```

are used for general communication between different levels of cache. If the current cache is last in the hierarchy, it points to RAM.

Realization of eviction policies are implemented with functions

```
void UpdateEviction(uint set, uint slot,
    uintptr_t tag);
uint Evict(uint set);
```

Evict function get "best" slot to be evicted, and *UpdateEviction* function updates eviction policy if changes need to be made. Every eviction policy first checks if there are any invalid sloth that can be evicted.

Random Replacement is the simplest eviction policy. When an overlap occurs, a random slot is taken and is replaced.

Least recently used discards the least recently used items first. It has to remember a time when each slot was accessed. To avoid overflow LRU updates history table on each eviction. It is time-consuming and also not optimized. There are plenty opportunities for optimizations in case of further development.

Most recently used it's the opposite of LRU and its implementation is also quite similar. First In First Out is similar to LRU, but it doesn't update history table on access but only when the data is fetched from the lower level.

IV. Results

We run few simulations to test different behaviors of the cache hierarchy. Our results are separated into three categories: Data type width, where we compare cache performance on 32-bit compiled version with the 64-bit version, Associativity, where we tested the influence of different sizes, number of levels and N-way set associativities and Eviction policies, where we analyzed the performance of different eviction approaches.

In the following results, if not explicitly said otherwise, a default number of cache level is 3, with 32KB 4-way set associative cache L1, 256KB 8-way set associative cache L2, and 2MB 16-way set associative L3. Default eviction policy is LRU.

A. Data type width

On a 64-bit version of application addresses of data are 64 bits long instead of 32 bits, which enables to address more memory in RAM. While 64-bit version comes with a significant number of benefits, a better cache hit is not one of those. While only all pointers/addresses are 64 bits long when we are dealing with pointers, we expect more misses. As can be seen on figure 1, this is true. On average, 64-bit version has approximately one clock cycle higher memory access, or in other words, miss ratio is greater for about 1% on each level. Luckily, in many applications, we are not dealing with a huge number of pointers.

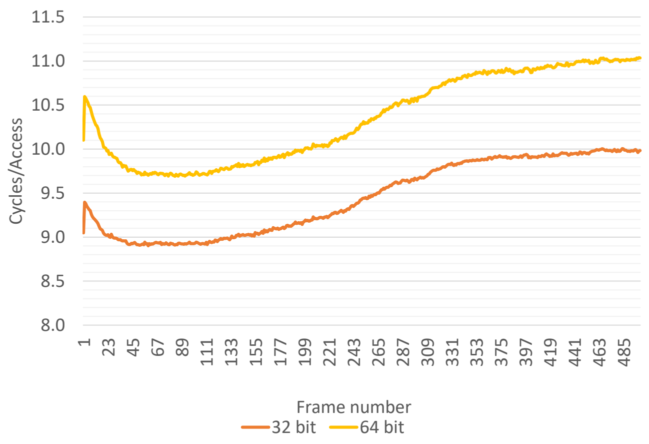

Figure 1. Average Memory Access Time 64 bit comparing to 32 bit version.

B. *Associativity*

To test the influence of a number of levels, N-way set associativity and size; we started with a simple 32KB 4-way set associative cache. With additional second level with the same configuration as the first one, we get worse results (figure 2). Because penalty to access L2 is high and hit ratio of L2 is not significant, performance is worse. We also noticed that size of cache level doesn't influence alone on the performance but only in combination with higher N-way set associativity. Although, it would benefit if the data were scattered all across the memory. Increasing number of levels does positively influence on average access time, but including that each lower level is bigger than predecessor and have larger sets. On random access and scattered data could only bigger cache would probably suffice to some extend. We got the best performance with 3 level cache with 32KB 4-Way set associative L1 cache, 256KB 8-way set associative L2 cache, and 2MB 16-way set associative L3 cache.

In practice manufactures must consider many trade-offs between latency, sizes, different N-Way set associativities and can't just freely add layers and sizes as we can in a simulation.

C. *Eviction policies*

As can be seen on figure 3, we conclude that most recently used eviction policy is not efficient in this particular type of application with high data locality. It is rather very inefficient, because we also obtained results with local miss rate over 100% which is not because of incorrect implementation, but because for almost every access to L2 we also have to evict and access L3. The more appropriate application would be

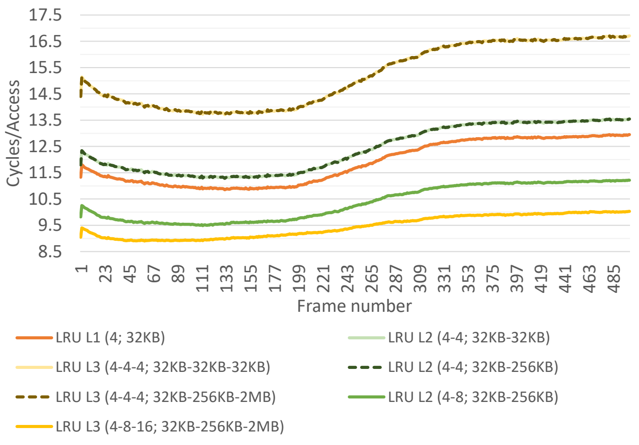

Figure 2. Comparison of various caches hierarchies.

where we access various data in a row, while repeatedly using one global data, (e.g., MRU Window Switching Order[2]).

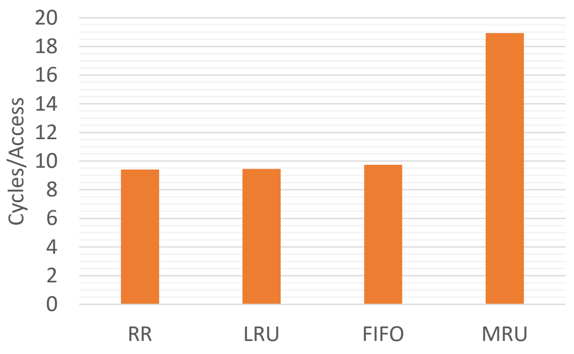

Figure 3. Average memory access time of different eviction policies.

In figure 4 it turns out, that Random Replacement has best average memory access time while LRU is very close. We

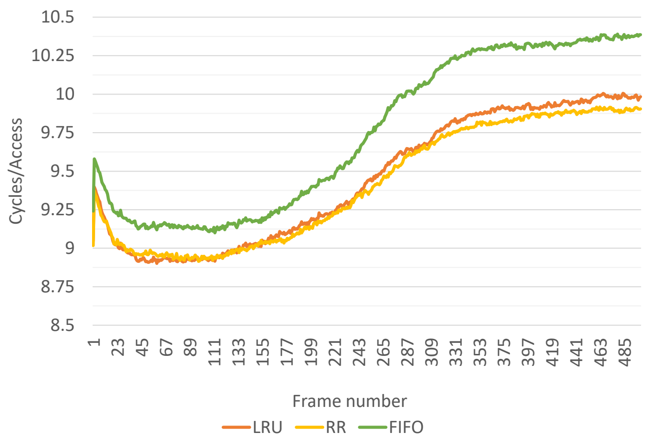

Figure 4. Average memory access time in LRU, RR and FIFO eviction policy per frame.

know this depends on different situations, but with such simple realization on a hardware level, no wonder that some ARM processors are using Random Replacement. They do that with additional more deterministic and sophisticate approach but the base idea is behind randomization[4], [1].

Also to be fair LRU is also implemented with higher complexity and better results in real life applications.

As can be seen in figure 5, LRU on cache L1 outperforms RR and FIFO eviction policy, but later on L3 (figure 6, it is slightly worse than RR. Because penalty to access RAM is much higher consequently RR is on average better. But this depends on latency or penalty determined by a user. We can also notice how cache misses slowly increases on L3 in general, because of its size and because most hits happen on L2 and L1.

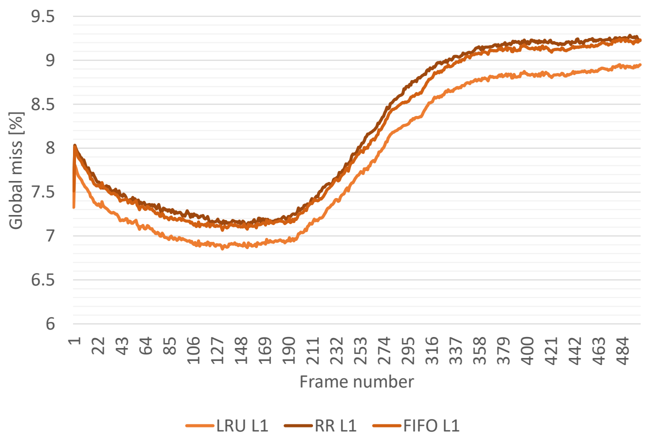

Figure 5. Global miss on L1 with different eviction policies.

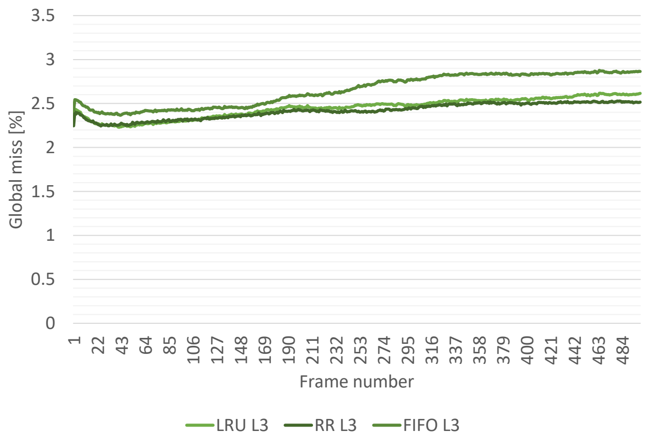

Figure 6. Global miss on L3 with different eviction policies.

References

[1] ARM l210 cache controller. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0284g/DDI0284.pdf. Accessed: 2017-10-02.

[2] MRU window switching order. https://en.wikipedia.org/wiki/MRU_Window_Switching_Order. Accessed: 2017-10-02.

[3] Ulrich Drepper. What every programmer should know about memory. 2007.

[4] Shuchang Zhou. An efficient simulation algorithm for cache of random replacement policy. In *Proceedings of the 2010 IFIP International Conference on Network and Parallel Computing*, NPC'10, pages 144–154, Berlin, Heidelberg, 2010. Springer-Verlag.