UNIVERSITÀ DI PISA

MULTILEVEL CACHE PROJECT

Computer Architecture course 2018/2019

Eugenia Petrangeli
Carmelo Aparo
Alexander De Roberto
Daniele Nicolai
Leonardo Fontanelli

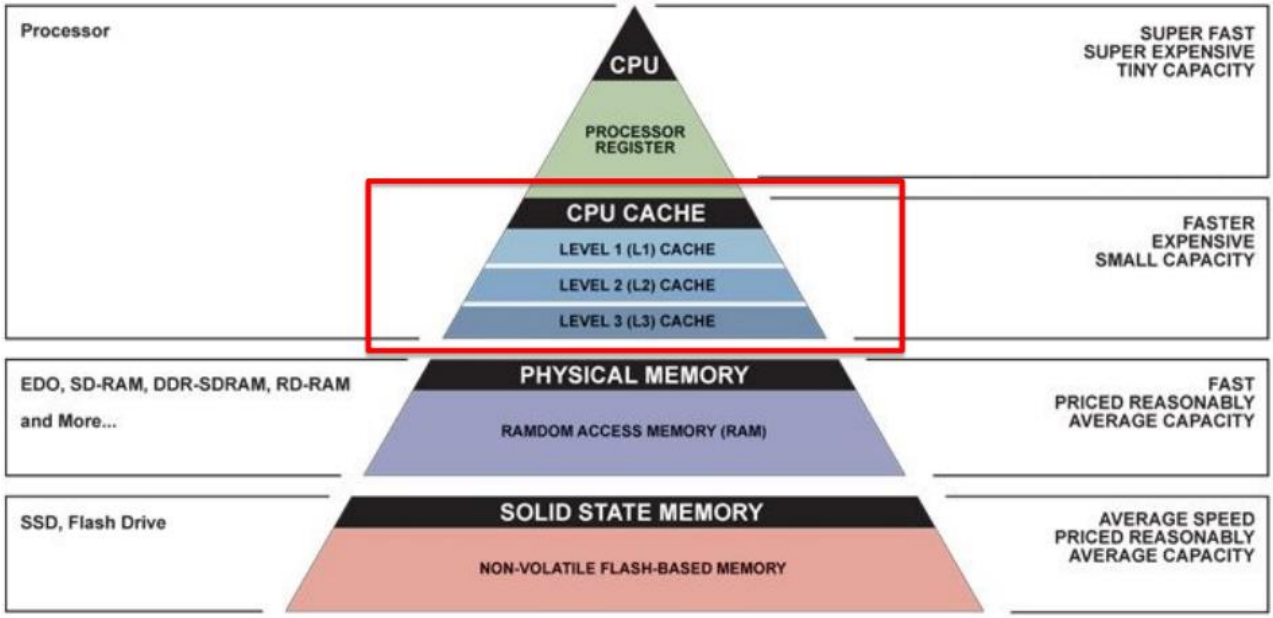
Index

1. Introduction

1.1 Overview

In the history of computer and electronic chip development, there was a period when increases in CPU speed outpaced the improvements in memory access speed. The gap between the speed of CPUs and memory meant that the CPU would often be idle. CPUs were increasingly capable of running and executing larger amounts of instructions in a given time, but the time needed to access data from main memory prevented programs from fully benefiting from this capability. This issue motivated the creation of memory models with higher access rates in order to realize the potential of faster processors. This resulted in the concept of cache memory which greatly improved data access latency.

Whenever the data is required by the processor, it is fetched from the main memory and stored in the smaller memory structure called a cache. If there is any further need of that data, the cache is searched first before going to the main memory. The benefits of caches are obtained by using a technology providing very short access time, however the cost of such technology is very high with respect to the main memory, so it was not feasible for a computer system's cache to approach the size of main memory. In order to solve this problem ideas such as adding another cache level (second-level), as a backup for the first-level cache were proposed. Nowadays the concept of multi-level caches is generally a better model of cache memories.



The advantages of using cache can be proven by calculating the Average Memory Access Time (AMAT):

$$\mathbf{AMAT = t_{accessLi} = t_{hitLi} + (p_{missLi} * t_{penaltyLi})}$$

where

$$\mathbf{t_{penaltyLi} = t_{hitL(i+1)} + (p_{missL(i+1)} * t_{penaltyL(i+1)})}$$

The hit time for caches is much less than the hit time for the main memory, so the AMAT for data retrieval is significantly lower when accessing data through the cache rather than main memory. In the case of a cache miss, the purpose of using such a structure will be rendered useless and the computer will have to go to the main memory to fetch the required data. However, with a multi-level cache, if the computer misses the cache closest to the processor (L1) it will then search

through the next-closest levels of cache and go to main memory only if these methods fail. The general trend is to keep the L1 cache small and at a distance of 1 or 2 CPU clock cycles from the processor, with the lower levels of caches increasing in size to store more data than L1, hence being more distant but with a lower miss rate. This results in a better AMAT. The number of cache levels is designed keeping in mind the trade-offs between cost, AMATs, and size.

Cache memory levels may be governed in some different ways, depending on its inclusion policy, which may be inclusive, exclusive or non-inclusive non-exclusive (NINE).
With an inclusive policy, all the blocks present in the upper-level cache must be present in the lower-level cache as well. Each upper-level cache component is a subset of the lower-level cache component. In this case, since there is a duplication of blocks, there is some wastage of memory. However, the miss penalty is reduced because if there is a miss in the upper level then there is a higher probability that the block is in the lower level.
Under an exclusive policy, all the cache hierarchy components are completely exclusive, so that any element in the upper-level cache will not be present in any of the lower cache components. This enables complete usage of the cache memory. However, there is a high memory-access latency when there is a miss.
The above policies require a set of rules to be followed in order to implement them. If none of these are forced, the resulting inclusion policy is called non-inclusive non-exclusive (NINE).

1.2 Project choices

The maximum memory available in a 16 bits system (65536 bytes) is pretty little, so the ratio between the total main memory and the total cache amount is relatively small, so the advantages of fast checking can be exploited without losing too much in capacity, unless caches are very small. For this project we have chosen to implement the inclusive cache.
A disadvantage of inclusive cache is that whenever there is a replacement in L2 cache, the (possibly) corresponding lines in L1 also must get replaced in order to maintain inclusiveness. This is quite a bit of work, and would result in a higher L1 miss rate. So, a decision of using a "PREDETERMINATED" policy in all levels except the first was taken.
Because of the decision of maintaining an inclusive organization of the data, we implemented a write through policy for store operations. This policy ensures that the data is stored safely as it is written throughout the hierarchy.
The last but not least choice in the design is the cache block size. There is not a specified value for this parameter, this decision is up to the user. The only constraint is that blocks in all the levels must be the same size. This allow us to handle the inclusivity properly and do not use backward invalidation.

1.3 Brief summary of operation

The proposed multi-level cache module acts as interface for the whole system, it receives the requests coming from the CPU and forwards them to the underlaying modules.
If the request is a LOAD, the *MultilevelCacheModule* forwards it to all cache levels (composed by *AssociativeCache* modules) starting from the first until it gets a HIT, otherwise a request to the main memory is generated.
If the request is a STORE, it is propagated through the whole hierarchy up to the main memory. When the request reaches the memory interface of the *MultilevelCacheModule*, the store operation is delegated and an acknowledgement is returned back to the CPU without waiting for the completion of the operation, so this module can be defined partially sequential.

An arbitrary number of levels can be instantiated, however a suggested two-level configuration, described later, is proposed. A zero-level configuration is also supported, in that case, the *MultilevelCacheModule* simply manages the conversion from the 16 bit-wise word of the CPU and the 32 bit-wise word of the memory.

2. Interaction between levels

In the following sections are described the main interactions between levels of cache in the case of an inclusive, two levels cache that operates in write-through.

We suppose that in case a miss occurs, when needed the victim block is passed to the following levels of cache in order to maintain the inclusion property without using the back-invalidation mechanism.

2.1 Load operation

When the processor performs a load operation to a specific address in the main memory, the following three cases could happen.

The first is the case when the first level of cache have the word requested by the CPU and so return directly it.

Figure 1: read operation with hit in first level

The following happen when we have a request for a word that is not present in the first level of cache, but it is in the second level.

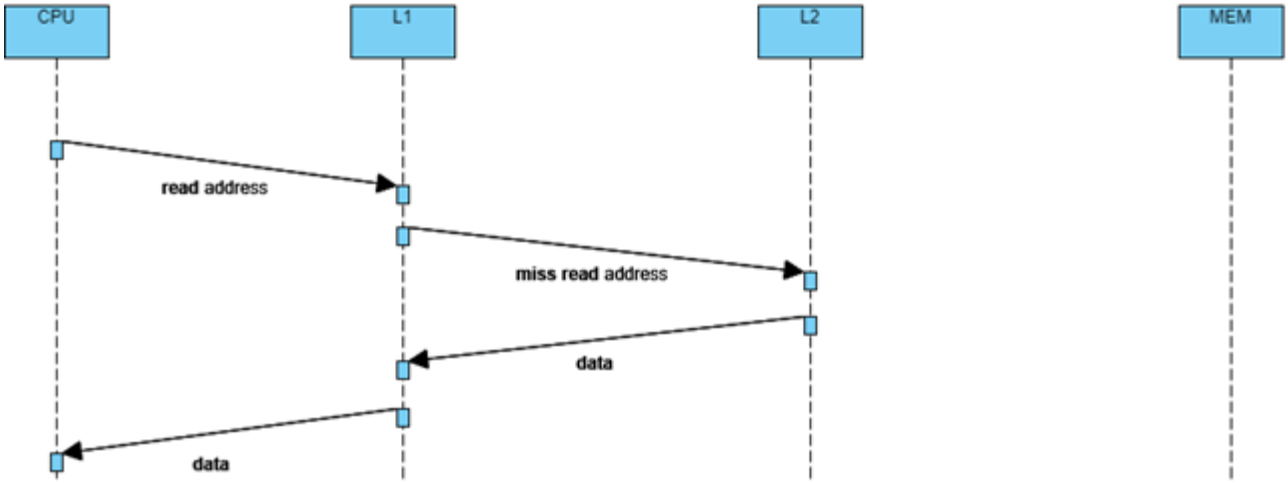


Figure 2: read operation with miss in first level only

And finally, we have the case when we have to access to the main memory in order to load and allocate in the levels of cache the entire block containing the word requested and then return it.

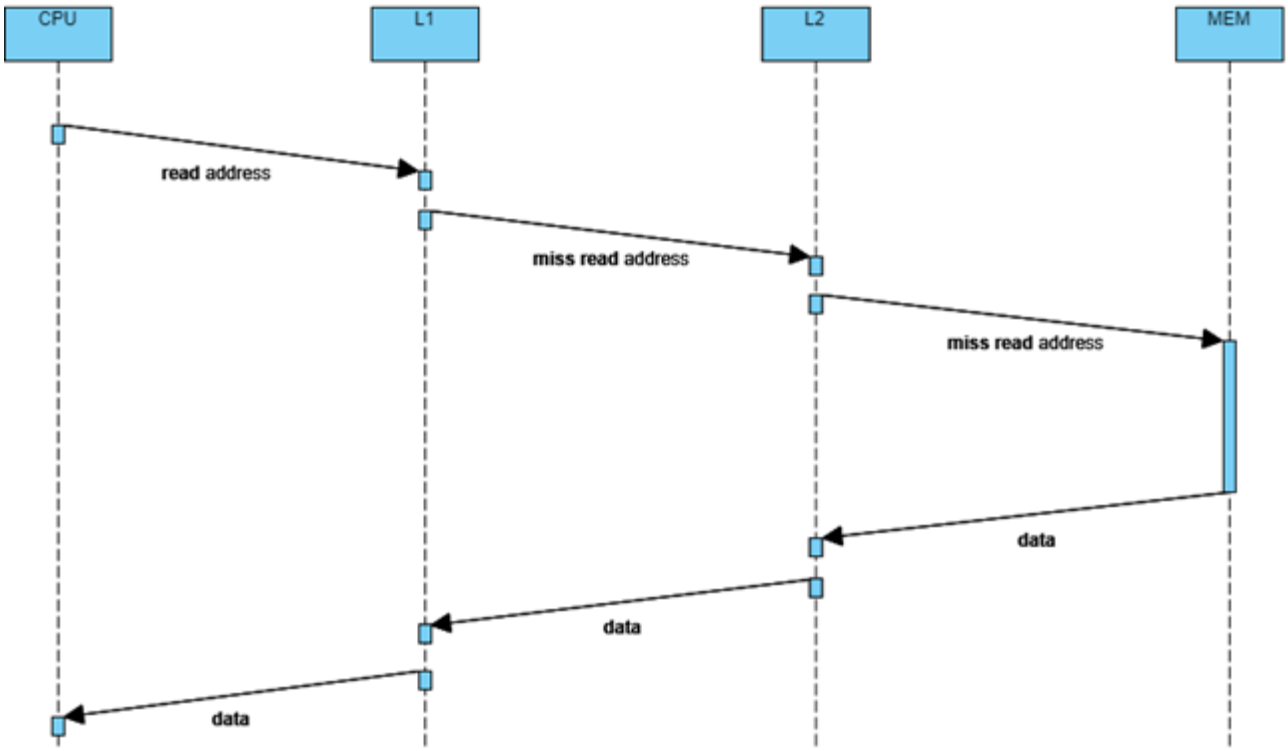


Figure 3: read operation with miss in both levels

2.2 Store operation

In case the processor performs a store operation to a specific address of the main memory we have this kind of interactions considering that we operate in write-trough.

Below is described the case that happens when the block, to which the data to write belongs, is present in the first level of cache. After we wrote successfully the data in the first level of cache we have to propagate the write operation also to the following levels.

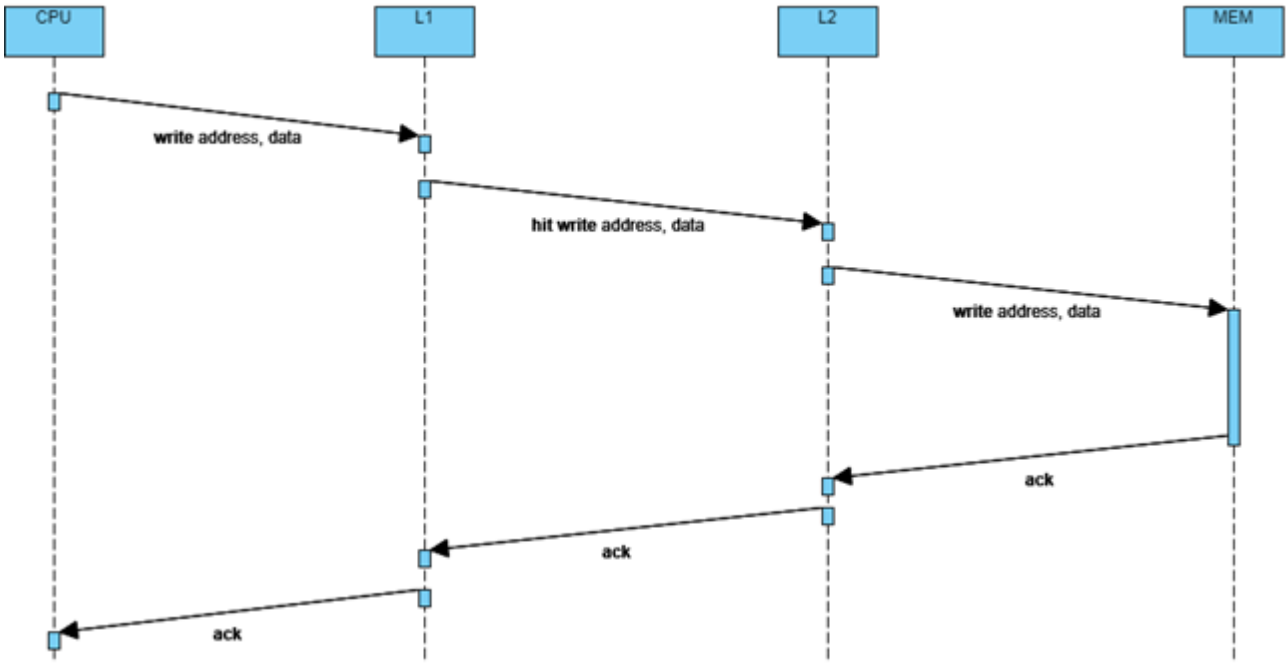


Figure 4: write operation with hit in first level

In case the block is not present in one level of cache we have different interactions depending of the particular policy chosen.

In the following sections are described the two main policies adopted: write-allocate and write-no allocate

2.2.1 Write operation with write-allocate policy

In case we chose to operate with write-allocate, when a miss occurs on a level during a store operation, we first have to allocate the block to which the data belongs before to complete the write operation.

The following two images describe the possible miss cases that could happen in this two-level cache: a miss in the first level only or a miss in both levels.

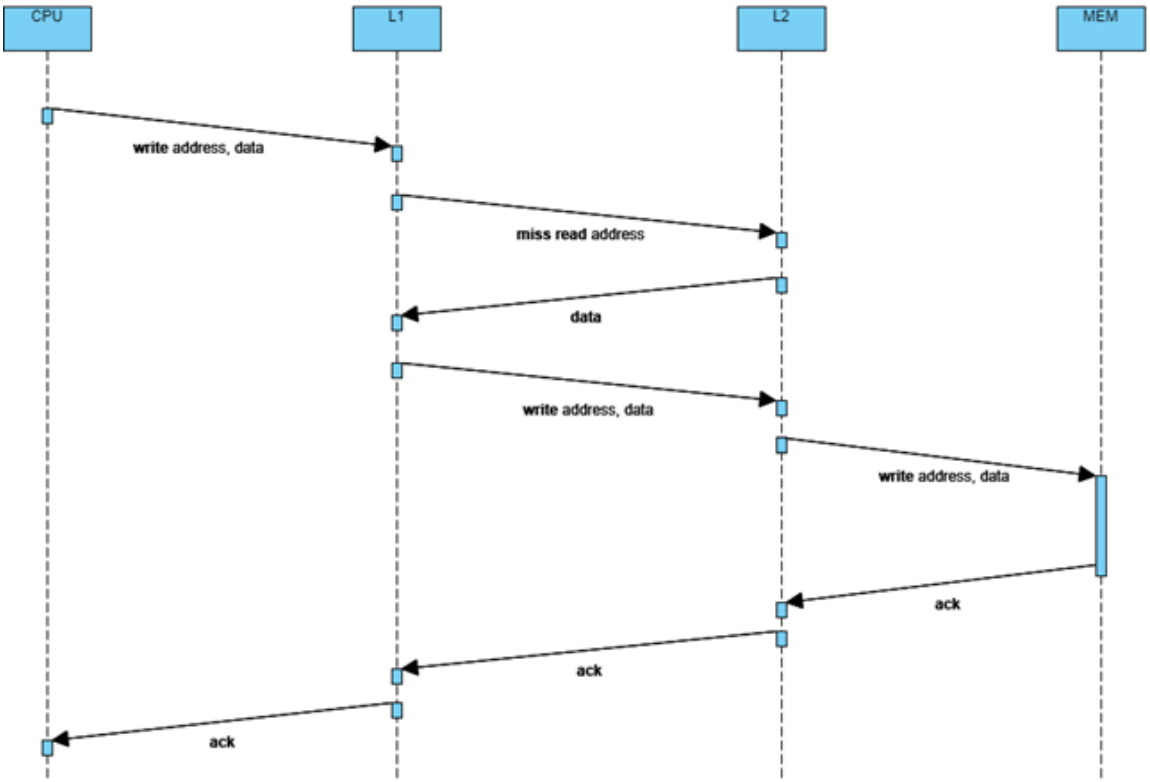


Figure 5: write operation with miss in first level only in write-allocate policy

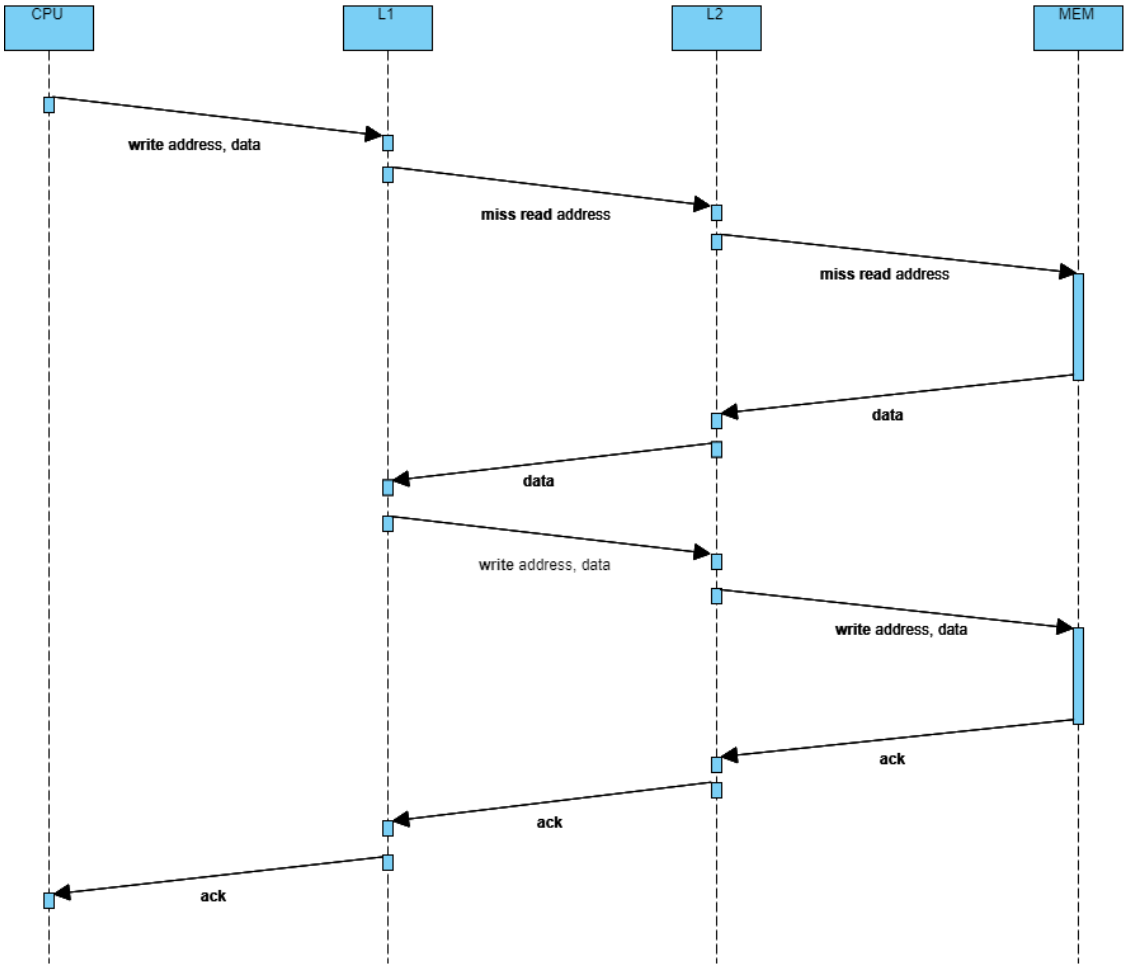


Figure 6: write operation with miss in both levels in write-allocate policy

2.2.2 Write operation with write no-allocate policy

In this case when a miss occurs in a level of cache, we propagate the store operation directly to the following levels.

This are the two cases of a miss in the first level only and the case of miss in both levels.

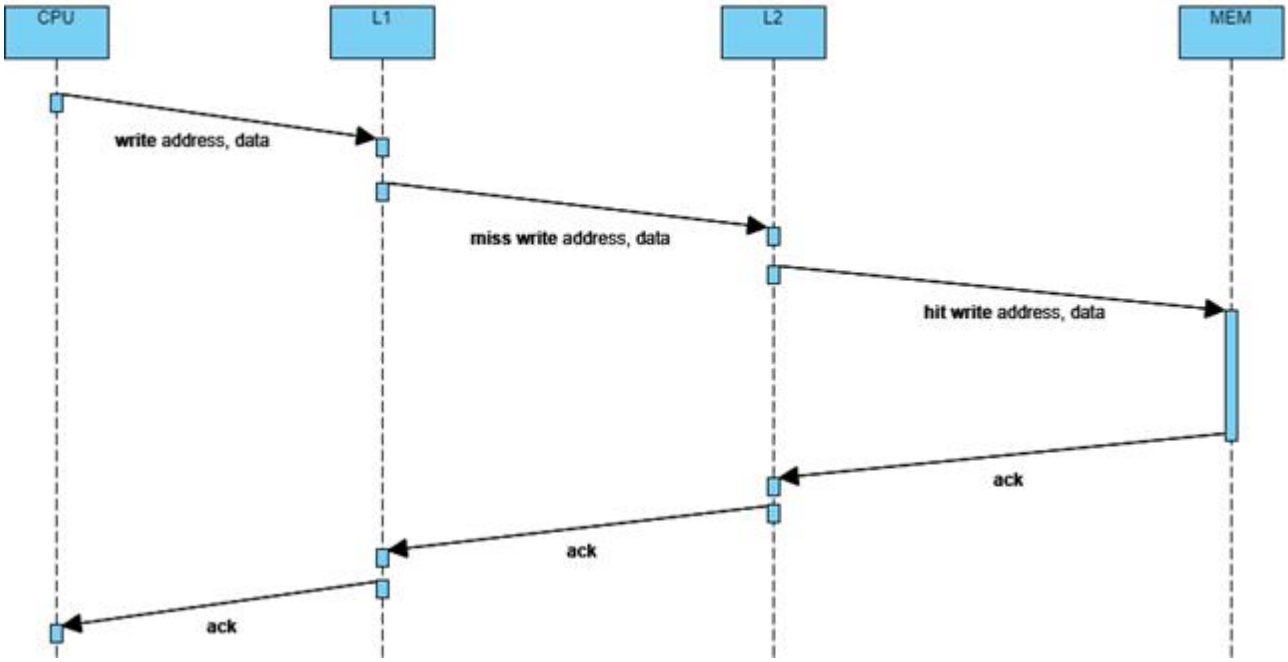


Figure 7: write operation with miss in first level only in write no-allocate policy

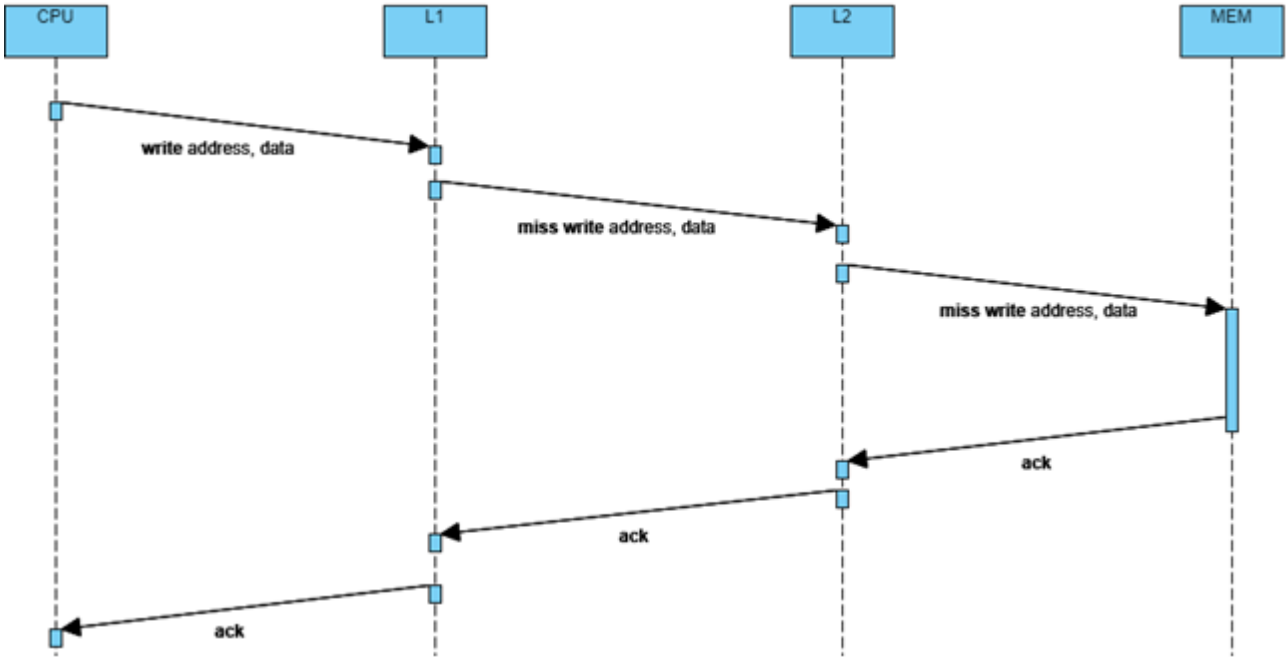


Figure 8: write operation with miss in both levels in write no-allocate policy

3. Interfaces

In this section we describe the interfaces used by *MultilevelCacheModule* and the *AssociativeCache* module. The second one is used by the *MultilevelCacheModule* to instantiate the levels of caches.

Then we describe the interfaces for the communications among the *CPU*, *MultilevelCacheModule*, *AssociativeCache* and *Memory* modules.

3.1 Multilevel cache interface

The following is the *MultilevelCacheModule* interface:

```
MultilevelCacheModule(System &sys, Bus &b, string name, uint numLvlCache = 2, cache_parameters *par = NULL, int priority = 0);
```

The arguments of the interface are:

- *sys* that is a reference to the system module in the simulator and it is used to add the *AssociativeCache* instantiated to the module list;
- *b* is a reference to the bus between the last level of cache and the memory and it is used to transfer data from/to the memory;
- *name* is the name of the module;
- *numLvlCache* is the number of levels of caches;
- *par* are the parameters of the cache levels;
- *priority* defines the priority of the module in the simulator.

There are three possibilities to instantiate a *MultilevelCacheModule*:

1. The first one is to choose the number of levels to use and to specify the parameters for each level;
2. The second one is to use the default configuration without choosing the number of levels and the parameters of the caches;
3. The third one is to use zero levels of cache. Obviously in this case the system has no caches.

3.1.1 Parametric interface

This is the interface with the highest degree of freedom, in fact in this case it is possible to specify each parameter of the system.

The first thing to do in order to define a *MultilevelCacheModule* is to choose the number of caches and to instantiate an array whose elements are *cache_parameters* structures.
Each element of the array defines the parameters for a single cache.

The order of the elements in the array indicates the cache level.
So, the first element is the first level, the second one is the second level and so on.

Now, we have to define the *cache_parameters* structure, which is used to choose the parameters of the levels of cache:

```
struct cache_parameters{
    HIT_POLICY writePolicy;
    MISS_POLICY allocationPolicy;
    ReplacementPolicy replPolicy;
    uint ways;
    uint cacheDim;
    uint blockDim;
};
```

- *writePolicy* is an enumerator that specifies if the write policy is “write through” or “write back”;
- *allocationPolicy* is an enumerator that defines if the allocation policy is “write allocate” or “write no allocate”;
- *replPolicy* is an enumerator that specifies the replacement policy;
- *ways* is the number of blocks within a set;
- *cacheDim* is the dimension of the cache;
- *blockDim* is the dimension of the blocks.

The following is an example of a parametric instantiation of the *MultilevelCacheModule*:

```
    cache_parameters *parameters = new cache_parameters[3];
    parameters[0].writePolicy = WRITE_THROUGH;
    parameters[0].allocationPolicy = WRITE_ALLOCATE;
    parameters[0].replPolicy = ReplacementPolicy::PLRU;
    parameters[0].ways = 1;
    parameters[0].cacheDim = 256;
    parameters[0].blockDim = 32;

    parameters[1].writePolicy = WRITE_THROUGH;
    parameters[1].allocationPolicy = WRITE_ALLOCATE;
    parameters[1].replPolicy = ReplacementPolicy::PLRU;
    parameters[1].ways = 1;
    parameters[1].cacheDim = 1024;
    parameters[1].blockDim = 32;

    parameters[2].writePolicy = WRITE_THROUGH;
    parameters[2].allocationPolicy = WRITE_ALLOCATE;
    parameters[2].replPolicy = ReplacementPolicy::PLRU;
    parameters[2].ways = 1;
    parameters[2].cacheDim = 2048;
```

```
parameters[2].blockDim = 32;


MultilevelCacheModule *multi = new MultilevelCacheModule(system, bus, "MLC",3, parameters);
```

3.1.2 Default interface

Using this interface, we can define a multilevel cache with 2 levels and that uses default parameters.

To use it, we can simply define a *MultilevelCacheModule* without specifying the number of levels and the parameters.

The default parameters for the first level are:

- Write policy: write through;
- Allocation policy: write allocate;
- Replacement policy: pseudo LRU;
- Number of ways: 1 (direct mapped);
- Cache dimension: 256 bytes;
- Block dimension: 32 bytes.

The default parameters for the second level are:

- Write policy: write through;
- Allocation policy: write allocate;
- Replacement policy: pseudo LRU;
- Number of ways: 4;
- Cache dimension: 1024 bytes;
- Block dimension: 32 bytes.

The following is an example of a default instantiation:

```
MultilevelCacheModule *multi = new MultilevelCacheModule(system, bus, "MLC");
```

3.1.3 Zero levels interface

This interface allows us to use a system without caches. So, in this case the communication is direct between the CPU and the memory.

To use this interface, we can simply specify 0 as number of levels without defining parameters.

The following is an example of the interface:

```
MultilevelCacheModule *multi = new MultilevelCacheModule(system, bus, "MLC",0);
```

3.2 Associative cache interface

The following is the *AssociativeCache* module interface:

```
AssociativeCache(System& sys, string name, string prev_name, string next_name,
        unsigned n_ways, unsigned cache_size, unsigned block_size, unsigned mem_unit_size,
        HIT_POLICY write_policy, MISS_POLICY alloc_policy,
        ReplacementPolicy repl_policy, int priority=0);
```

The arguments of the interface are:

- *sys* is a reference to the System module of the simulator;
- *name* is the name of the module;
- *prev_name* is the name of the module in the previous level;
- *next_name* is the name of the module in the next level;
- *n_ways* is the number of blocks in a set;
- *cache_size* is the cache dimension;
- *block_size* is the blocks dimension;
- *mem_unit_size* is the number of bytes that the cache has to return during a read operation;
- *write_policy* is the write policy to use;
- *alloc_policy* is the allocation policy to use;
- *repl_policy* is the replacement policy to use;
- *priority* is the priority of the module inside the orchestrator.

When the *MultilevelCacheModule* instantiates the *AssociativeCache* modules, it defines always:

- *name* = Li, where i is the number of the level of the cache. For example, the first level is L1;
- *prev_name* = “UpperLvl” and *next_name* = “LowerLvl”. In this way the *MultilevelCacheModule* answers to the massages that use these destinations and the associative cache has no vision about who is the previous and the next level;
- *mem_unit_size* = the block size of the previous level. In this way each cache can return the number of bytes of the block of the previous level of cache during a read operation;
- *write_policy* = write through because the write back policy has not been implemented in the *MultilevelCacheModule*.

All the other parameters are defined in the *cache_parameters* structure passed to the *MultilevelCacheModule* constructor.

3.3 Messages

The *MultilevelCacheModule* communicates with the *CPU* module, the *AssociativeCache* module and the *Memory* module.
For the communication with each one of them it is defined a different type of message.

3.3.1 CPU message

The communication with the *CPU* module is done using the following structure:

```
struct memory_message{
    bool        type;
    uint16_t    address;
    uint16_t    data;
```

```
};
```

- *type* is a Boolean value which if it is false it means that the operation is a read one, otherwise it is a write one;
- *address* is a number on 16 bits that contains the address of the word to read/write;
- *data* is a word on 16 bits. If the operation is write, it contains the word to write, otherwise it is used to contain the word to read in the response message to the CPU.

When the operation is a write one, an ACK is sent to the CPU in order to notify the end of the operation.

3.3.2 Cache message

The communication with the *AssociativeCache* module is done by means of the following structure:

```
struct cache_message {
    block target;
    block victim;
    bool type;
};
```

where *type* is a Boolean value that is equal to false for read operations and true for write ones.

Instead the fields *target* and *victim* are *block* structures:

```
struct block {
    uint16_t address;
    uint16_t *data;
};
```

This one is composed by two fields:

- *address* is the address of the block to read/write;
- *data* is a pointer to the block to write during a write operation or it contains the read block or the read word during a read one.

The field *target* contains the *block* structure of the requested block.

Instead the *victim* field contains the *block* structure of the block that is chosen by the replacement policy. Obviously if there is no replacement, it is empty.

3.3.3 Memory message

The communication between the multilevel cache and the memory is done by using a bus interface.

When the multilevel cache wants to read/write on the memory, it sets the bus by using the *set* function of the class *Bus*. Then it sends an empty message to the memory module to notify that there is a request.

During the read operation when the memory answers, it sends an empty message to the *MultilevelCacheModule* to notify that there is the response in the bus and the multilevel cache gets the response by using the *get* function of the *Bus* class.

Both the *set* and *get* functions use a pointer as argument to a *Bus_status* structure, whose fields are:

- *request* that is the type of the request (read/write);
- *address* that is the address of the word to read/write;
- *data* that contains the word to write during a write operation or it contains the word to read when the memory answers during a read operation.

During the *get* and *set* operations there is also the possibility that the bus is busy, so the multilevel cache has to wait a certain amount of time and try again to access to it.

4. Implementation

In this section we describe how the *MultilevelCacheModule* is implemented, looking to the code of the main functions.

4.1 Constructor

The aim of the constructor is to instantiate the associative caches and the resources needed to work.

So, the multilevel cache stores the number of levels and the dimension of the blocks of the last level of cache. The last information is important in order to know how many words have to be read from the memory when the last level of cache requests a block.

If there are zero levels of cache, the multilevel cache has to coordinate the communication between the CPU and the memory, so it uses as dimension of the last level block the dimension of a memory word (the minimum dimension that can be read from memory that is in our case 32 bits), because the CPU wants to read/write a single word of 16 bits for each operation.

Then the constructor instantiates the *AssociativeCache* modules initializing them with the parameters passed as argument or with the default ones.

If the parameters are not passed and the number of levels is different from the default one, the constructor stops the execution, because it is an incoherent configuration.

However, since the *MultilevelCacheModule* wants to realize an *inclusive* multilevel cache, not all the configurations are accepted.

4.1.1 checkCacheParameters function

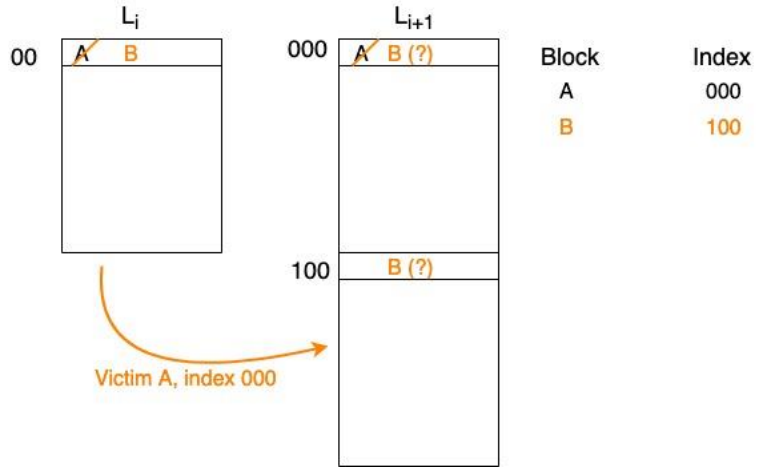
The following is the list of the meanings of the conditions:

1. The first level of cache must not use the *PREDETERMINED* replacement policy, since there is no level before it and for this reason it can't receive the victim from another level;
2. The number of blocks in a set can't be zero, obviously it has no sense;

3. The dimension of the cache must be bigger than the dimension of a block, also this one is an obvious condition;
4. The dimension of a block cannot be zero bytes and it has to be a power of 2;
5. The dimension of the cache cannot be zero bytes and it has to be a power of 2;
6. The number of blocks must be divisible by the number of ways, otherwise the sets have a different number of blocks inside;
7. The write back policy is not accepted since it is not supported by this module;

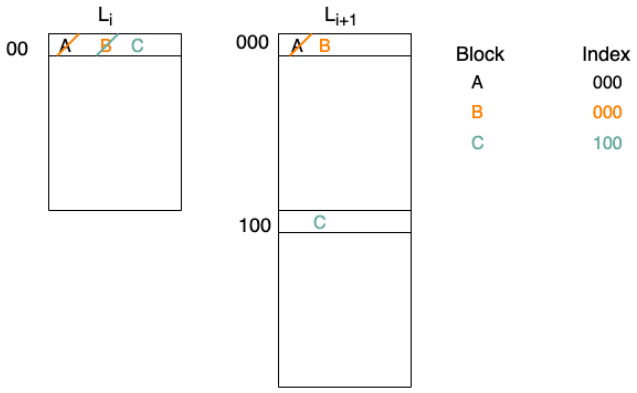
From now on, there are conditions that check the parameters of a level with respect to the parameters of the next level in order to maintain the *inclusivity* of the multilevel cache without using the *backward invalidation*, that is not implemented.

8. If the i-th level is *direct mapped*, the next level i+1 must not use *PREDETERMINED* as replacement policy, because for each index in Li there can be more than an index in L(i+1) and in this way the victim address cannot correspond to the address of the block to add. For example if the block to insert has index 100 for the L(i+1)-th cache and 00 for the i-th level (because the i-th level is smaller and it uses only the first two bits for the index), there

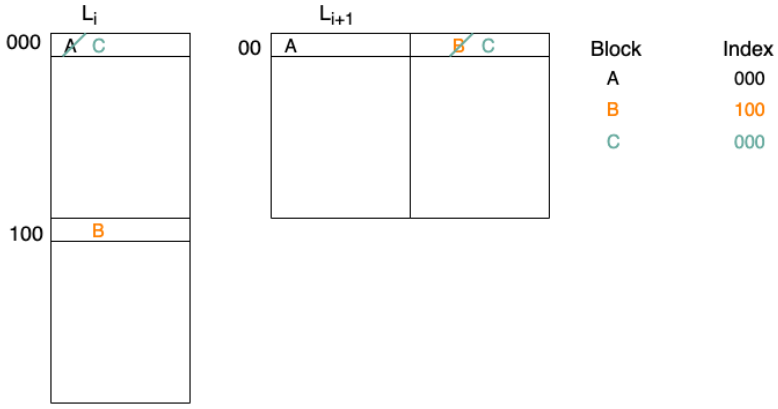


is the possibility that the victim chosen by i-th level has address 000 that is the same for the i-th level, but different for the (i+1)-th one.

In addition, the number of entries of the (i+1)-th level has to be equal or bigger than the i-th level, because in this way when we need to replace a block in both levels, since the i-th level is *direct mapped*, the new block can be inserted only in one position and there are no other blocks with the same index.
So when the block is inserted in the (i+1)-th level the replacement policy will replace a block that isn't no more in i-th level for sure.
For example, if the block to insert in i-th level has index 00, the i-th level replace the block with index 00 and the only block in that position is the new one.
When the (i+1)-th will replace a block, it will replace a block with index x00 and it will insert the new block in that position. It is not important the block replaced from (i+1)-th level since it has address 00 for the i-th level and in that position there is only the new block (that is obviously also in (i+1)-th level);



The inclusivity is manteined



The inclusivity is not manteined

9. If the i-th level is *not direct mapped*, the (i+1)-th level has to use the *PREDETERMINED* replacement policy.

   In fact, if the levels are *not direct mapped*, we can choose the same victim in both levels in order to maintain the *inclusivity*.
   So, using the *PREDETERMINED* replacement policy I can send the victim chosen by i-th level to the (i+1).

   In addition, the number of entries in the i-th and the (i+1)-th levels has to be equal. It is needed since otherwise the *PREDETERMINED* replacement policy doesn't work, because if the (i+1)-th level has more entries than the i-th, then for an index of a set in i-th, there are more indexes in (i+1)-th level.
   It is the same reasoning for which we can't use the *PREDETERMINED* policy in the direct mapped case (look to point 8).

   At the end, the number of ways of the (i+1)-th level has to be equal or bigger than the i-th ones, because if the number of entries is equal and the number of ways of the (i+1)-th is smaller, it means that the dimension of the (i+1)-th level is smaller than the i-th level.
   Obviously in this case the multilevel cache cannot be inclusive since a smaller cache (the

(i+1)-th level) can't contain all the blocks of a bigger one (the i-th level);

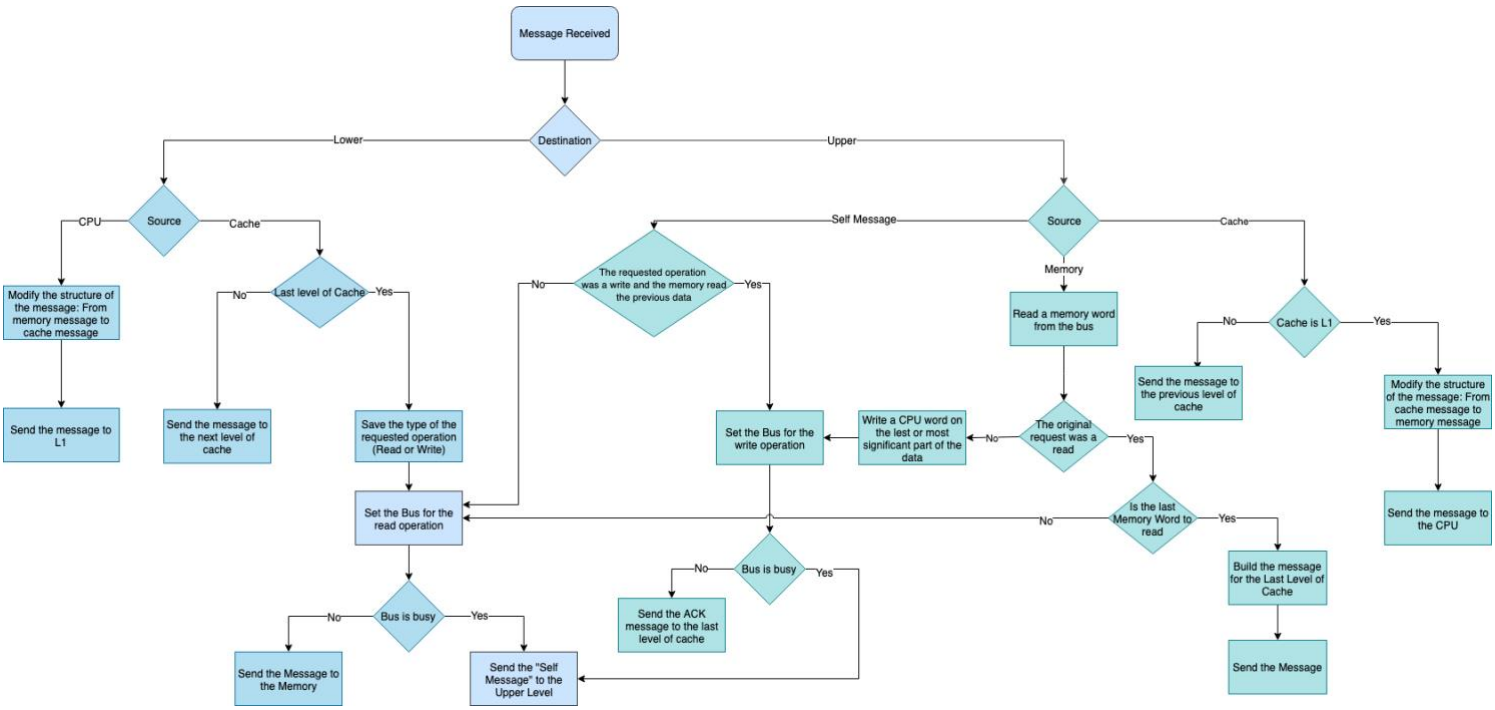
10. All the caches must have the same block dimension, otherwise when there is a replacement and the (i+1)-th level blocks are larger than the i-th ones, the (i+1)-th level has to send a *backward invalidation* for the additional words that are in its block and that can be in other blocks in i-th level cache.

4.1.2 onNotify function

This function handles all the messages destinated to the *MultilevelCacheModule*. This module is identified by two names: *LowerLvl* and *UpperLvl*. All messages whose destination is Lower*Lvl* are directed to the memory, on the contrary, all messages directed to the CPU are addressed by the destination name *UpperLvl*.
Is important to underlying that the Memory Word is bigger than the CPU Word so every time the *MultiLevelCacheModule* wants to write a data on the memory has to:

1. Read the Memory Word from the Memory
2. Write the CPU Word on the most or less significant part of the data
3. Write the updated data on the Memory

The following flowchart shows the behavior of the function *onNotify()*



The main functions used can be grouped into two categories:

- Functions that manage the read and write operations:
  1. `void initializeReadFromMemory(uint16_t wordAddress, uint blockSize);`

     Is called every time a read/write operation is requested on the Memory. Saves the address requests, computes the address of the first word in the block to read/write and computes the address of the first word in the consecutive block. This last address is used to determine when the read/write operation is finished.
  2. `void setReadFromMemory();`

     Sets the bus with the data address that the *MultilevelCaheModule* wants to read. If the bus is busy then a *SelfMessage* is sent in order to retry the read after a certain delay, otherwise notifies the Memory that a new operation is ready on the bus.
  3. `void getReadFromMemory();`

     Is called when the Memory notifies the presence of data on the bus. This function reads a Memory Word and updates the address of the next Memory Word to read. If the bus is busy it means that the data is not ready. When the Memory notifies the presence of the requested data, it must be present, otherwise an error has occurred.
  4. `void setWriteOnMemory();`

     Sets the bus with the Memory Word and its address that the *MultilevelCaheModule* wants to write. If the bus is busy, then a *SelfMessage* is sent to the *UpperLvl* in order to retry the write after a certain delay. If the bus is free:
     - Send an ACK to the last level of Cache (if there are any, otherwise directly to the CPU)
     - Notifies the Memory that a new operation is ready on the Bus

     It may happens that the CPU receives the ACK message before the Memory has read the data on the bus, therefore the CPU can send another operation without waiting for the Memory since is usually slower. However, the new operation sent by the CPU can't be performed until the Memory has finished reading the old data from the bus.
  5. `void writeOnMemoryWord();`

     Writes the CPU word on the most or less significant part of the Memory word.
- Functions that convert the structure of the message
  1. `cache_message* cpuToCache(memory_message* cpu_mstr);`

     CPU and Cache use different message structures in order to communicate. This function is called when exists at least one cache level. It converts the message structure from *memory_message* to *cache_message*.
  2. `memory_message* cacheToCpu(cache_message* cache_mstr);`

     Is called when exists at least one cache level and convert the message structure from *cache_message* to *memory_message*.

5. Simulator

Once the Multilevel Cache module has been defined and written in every part, there is the need to test and verify the correctness of the execution. Obviously, with the module by itself we are not able to test the code, the reason is that we need the connection and the interaction with the other parts of the architecture. Hence, the next necessary step of the workflow is to design a simulator of the modules of the architecture in order to see if the Multilevel Cache satisfy the requests in a correct way.
Reasoning about the interaction of the Multilevel Cache in the computer, the simulator will be composed by a CPU module, an Associative Cache module and a Memory module.

5.1 CPU Simulator

The first module that has been necessary to implement in the simulator is obviously the CPU, which sends the READ/WRITE requests to the Lower Level.
Without entering immediately in the deep of the details of the architecture, from a simulation point of view and particularly in our specific purpose, the CPU requires READs with an arbitrary address and WRITEs for arbitrary address and data as well.
As already said, the Multilevel Cache coordinates the communication between CPU, Cache and Memory; So, it will take the requests from the CPU and will forward them to the other modules no matter of the specific contents of the requests.
Passing to the details of the implementation: in order to have a "programmable" CPU we defined a boolean array in the *simulator_define.h* file, and every entry of the array state if we want to have a read (false) or write (true) request from the CPU simulator.
Hence, every time the CPU has to send a request it will check for the specific entry referred to the current iteration, in order to trigger the correct command. The class header has a variable called *indexMessage* that take trace of the current iteration and is incremented every time a request is sent to the *Lower Level*.
Starting from the class constructor, this has been implemented with the purpose of triggering immediately the first request. So, it has to build a new message (the *buildMagicStruct* function has been written for this) and send it to the lower level.
In the details of the *buildMagicStruct* we can see that in order to generate a new message, will use the *rand()* function for the address and data fields. For the type of the operation will assign the specific entry of the array in the *simulator_define.h* file.
Moving to the *onNotify* function, which is called every time that the CPU Module receives a message. The CPU receives only two kind of messages, that are the answer to the requested action: a response to a read will have a magic struct containing the data requested, and a response to a write will have a magic struct NULL because is the ACK coming from the other modules for a well completed operation. Hence, the *onNotify* prints the message received and, if *indexMessage* is still less than the number of entries of the array, will trigger immediately another message, calling the *buildMagicStruct* function. In this way we'll have a sort of cycle of messages, where every received message (that is the notification of a completed operation) will cause a new operation request.

5.2 Memory Simulator

The second module in order of importance in our simulator (we recall that the *Multilevel Cache* can work even without the Caches) is the Memory Module, that has to satisfy the requests of the CPU.
The idea of the Memory in our purpose is very simple, because it doesn't have to really give back a precise data to the CPU, that's because we don't need to execute any particular flow of program, but only to test the functionalities of the *Multilevel Cache*. Hence, when it has to answer to a read it will generate a random value for the data field and when there is a write to perform it will not actually store the value anywhere.
Going deep in the implementation, we start analyzing the *onNotify* function: as already said, it only has to blindly answer to the requests, so we don't need to implement some kind of array like in the CPU simulator case.

As always in this kind of simulator, the *onNotify* function is triggered every time the module receives a message, so the first check that we have to perform, is to see if the message is destined to the Memory.
This module can receive from the *Upper Level*, which is the Multilevel Cache, and from itself (a *Self Message*). In the case of *Upper Level*, the message is the notification that in the bus will found a request, so the Memory will interrogate the bus with the *get* function and if it's not busy (otherwise the program fails), in the *status* variable will found the details of the request.
If the request is a read, will put a random value in the *data* field of the *status* variable, otherwise if a write is requested, simply does nothing. For a read request, we need to put the data in the bus, so we can use the *set* function to interrogate the bus. If the bus is free, we can send the notification to the *Upper Level* and the operation is over. Otherwise, we need to send a *Self Message* to try again to interrogate the bus, in this case we use a bool variable called *saved* in order to remember our self to interrogate the bus again and conclude the operation.
Hence, when a *self Message* is received from the module, and the variable *saved* is true, we try again to use the *set* function of the bus, and if it's free, the operation is completed like before.

5.3 Cache Simulator

The last module implemented in the *Simulator*, is the *Cache Module*. First of all, speaking about the behavior of the module, we implemented a matrix where the idea is similar of the one in the CPU module, where every entry represents the action to take when the module has a request.
So, the line of the matrix is referred to the specific level of Cache, the level number "i" will refer to the line "i". The column will be the one that will determine the answer, with a *true* we'll expect a HIT, and with a *false* we'll expect a MISS.
In this simulator, for the purpose of testing the Multilevel Cache, we implemented a NO ALLOCATE cache, that's because in this way we still test every kind of request and every function of our module.
In the implementation of the *onNotify* function, we have to make a first distinction between the direction of the message.
The first case taken into account is the case when the message is coming from the left, so it is a CPU request and obviously can be a read or write request. For both operations we use the Boolean array to decide the answer to the request, the static variable index of the header, will determine the column to consider and it is incremented after every iteration.
Considering a HIT case of a read, we populate the array *target.data* with a cache block generated with a random function using a for cycle and then we can send it to the previous level with a message.
Passing to the MISS case of a read, the only thing that the module have to do is to select a victim, in order to simulate the replacement policy.
In the write request we can still have the miss and hit cases, in both cases, given that we're implementing a NO ALLOCATE policy, the request will go through the levels, directly to the memory.
Now we're dealing with the case of a message coming from the right, which is the memory and going to the CPU. Here we can have two cases as well, the read and a write.
In case of write, we're having an ACK coming from the last level of cache and we only have to propagate it to the *Upper Level*.
In case of read, if we are in the L1 level, we have to give the data to the CPU, but in this case we have the data inside the block, so we have to select the correct word in the cache block and send it to the CPU.

5.4 Chosen Delay in the Simulator.

In this kind of simulator, which is an event driven one, the timeline proceeds with messages. At every message receival, an event is generated and the *onNotify* function of every module is called. In this way events can be handled, and the simulation can proceed. In this scenario, the delay with which the message is delivered to the modules takes a fundamental importance, and it must be chosen in a coherent way with the respect to the architecture.
Another fundamental aspect of the *Orchestrator module*, that handle the communication between every other module, is that the time is marked by ticks. For the analysis of the delays we assume that a tick corresponds to a clock cycle.
The first delay taken into account is the *CPU to cache* delay, which is the delay of the message coming from the CPU module. In our case we set the value to 0, that because the CPU has already added his delay at the message directed to the Multilevel Cache.
The second delay of the module is the *Cache to Cache*, considering the clock cycle needed in order to access the caches, we put the value 11.
The *self-Message* delay has been chosen to 2, this delay is useful because it has been used when, for example, we found the bus busy and we need some time to try again to take it.
The *Cache to Memory* delay has been chosen to 100 because, the time to access the memory must be larger than the one to access the caches, so the value has to be larger than the *Cache to Cache* delay, and has been considered with the respect to the clock cycles in order to complete an access.
For *Cache to Cpu* delay we put the value 4, and same as before we considered the clock cycles in order to reach the CPU from the Lower Level.

5.5 Parametric Constructor

For the Multilevel Cache Module we defined a Parametric Constructor as already said in the 4.1 paragraph, thanks to this we're able to pass many parameters and build the Cache as we prefer. Obviously we must check for the correctness of the parameters of the function, so we defined the *check_parameters* described in the 4.1.1 paragraph.
In the testing phase we defined some "wrong parameters" in order to check if the *check_parameters* correctly calls the exit function for errors.
For this test we consider the case of 2 levels of cache.

- TEST 1: Dimension of the cache blocks > Dimension of the cache.
- TEST 2: We put a value for number of blocks which is not divisible by the value of the dimension of the cache.
- TEST 3: Considered L1 as Direct Mapped, we put a value for the dimension of the cache L2 which is less than the dimension of cache of L1.
- TEST 4: In this case we considered L1 as Direct Mapped and L2 as PREDETERMINED.
- TEST 5: We put L1 as Chosen Victim.
- TEST 6: We put a value equal to 0 for the number of ways of the cache.
- TEST 7: We set the value 0 for the dimension of the cache block.
- TEST 8: With the same philosophy as the previous test, we tried to put 0 for the dimension of the cache.
- TEST 8: In this test we try to set the block as a non-power of two.
- TEST 9: As the previous test, we tried to put a value non-power of two for the dimension of the cache.

- TEST 10: We tried to put a *Write Back* as a Policy.
- TEST 11: We selected different values for the number of blocks for different caches.
- TEST 12: In this case we put the L1 caches as direct mapped, and for the L2 Caches we put a value for the number of entries which is less than the one of the previous cache.

5.6 Coming messages from the CPU and answers from the Cache Modules

In order to achieve the perfect functioning of the simulator, we need to exploit every possible case and combination for READ and WRITE from the CPU and HIT and MISS for the lower level of caches in a NO ALLOCATE policy, to see if every case have the correct response.
The test has been done with 2 levels of caches.

- TEST 1: (CPU) READ -> (L1) HIT
- TEST 2: (CPU) READ -> (L1) MISS -> (L2) HIT
- TEST 3: (CPU) READ -> (L1) MISS -> (L2) MISS -> Memory Access -> (L2) -> (L1) -> (CPU)
- TEST 4: (CPU) WRITE -> (L1) HIT -> (L2) PROPAGATION -> Memory Access -> ACK -> (L2) -> (L1) -> (CPU)
- TEST 5: (CPU) WRITE -> (L1) MISS -> (L2) HIT -> Memory Access -> ACK -> (L2) -> (L1) -> (CPU)
- TEST 6: (CPU) WRITE -> (L1) MISS -> (L2) MISS -> Memory Access -> ACK -> (L2) -> (L1) -> (CPU)

5.7 Valgrind Test

In order to check for memory leaks or error related to memory handling, we performed a Valgrind Analysis, that shows us that the memory is allocated and deallocated in a correct way.

```
--3262-- REDIR: 0x5474950 (libc.so.6:free) redirected to 0x4c30cd0 (free)
==3262==
==3262== HEAP SUMMARY:
==3262==     in use at exit: 0 bytes in 0 blocks
==3262==   total heap usage: 750 allocs, 750 frees, 89,002 bytes allocated
==3262==
==3262== All heap blocks were freed -- no leaks are possible
==3262==
==3262== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==3262== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Proceeding with the analysis of the code and the executable, we performed a Kcachegrind analysis that show us the following graph: