UPPSALA
UNIVERSITET

IT 13 089
Examensarbete 30 hp
December 2013

Learning Cache Replacement Policies using Register Automata

Guillem Rueda Cebollero

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Learning Cache Replacement Policies using Register Automata

Guillem Rueda Cebollero

Processors are a basic unit of the computer which accomplish the mission of processing data stored in the memory. Large memories are required to process a big amount of data. Not all data is required at the same time, few data is required faster than other. For this reason, the memory is structured in a hierarchy, from smaller and faster to bigger and slower. The cache memory is one of the fastest elements and closest to the processor in the memory hierarchy.

The processor design companies hides its characteristics, usually under a confidential documentation that can not be accessed by the software developers. One of the most important characteristics kept in secret in this documentation is the replacement policy. The most famous replacement policies are known but the hardware designers can apply modifications for performance, cost or design reasons.

The obfuscation of a part of the processor implies many developers to avoid problems with, for example, the runtime. If a task must be executed always in a certain time, the developer will take always the case requiring more time to execute (also called "Worst Case Execution Time") implying an underutilisation of the processor.

This project will be focused on a new method to represent and infer the replacement policy: modelling the replacement policies with automaton and using a learning process framework called LearnLib to guess them. This is not the first project trying to match the cache memory characteristics, actually a previous project is the basis to find a more general model to define the replacement policies.

The results of LearnLib are modelled as an automaton. In order to test the effectiveness of this framework, different replacement policies will be simulated and verified. To provide a interface with a real cache memories is developed a program called hwquery. This program will interface a real cache request for using it in Learnlib.

Handledare: Martin Stigge
Ämnesgranskare: Wang Yi
Examinator: Roland Bol
IT 13 089
Tryckt av: Reprocentralen ITC

Contents

Chapter 1

Introduction

In the last years the evolution of processors was focused on incrementing the speed for transferring and processing. Some components in the processor design are still the same: for example, the cache memory is one of the fastest memories in the memory [5].

The cache memory is an important part in the architecture of all processors. How this component works is important for the required time to process the program. The cache memory is widely used nowadays in the computer processors, from desktop computers to more critical environments as assembly lines in factories. In those environments the time required for executing a piece of code is critical: all processes are synchronized in time and an error can stop all the different tasks in the line of the factory.

The main characteristic of the cache memory is being one of the fastest in the memory hierarchy [10]. This characteristic allows to process the data quickly. The disadvantage of this level in the hierarchy is the cost; the cache is a limited resource and must be well managed: not all the data blocks can fit in it, but if a block of data is required, the time penalty of moving the block from lower level of the hierarchy can affect the process timing.

For this reason, the blocks on the cache memory are managed by a replacement policy. The strategies adopted by this policy can affect to the performance of the program in execution, for example, if the required data to process is not in the cache. It is important to take in consideration how this replacement policy works to know which blocks will be at runtime in the cache memory.

The information related about the cache design is sometimes confidential and the company saves the rights to publish this information. This enterprise strategy doesn't help developers to create programs with a realistic worst-case execution time (WCET). Inefficiency and underutilisation of the processor are the consequence of developing with unrealistic WCET [7] [13].

Trying to get all specifications of the cache memory was the goal of recent work [1]. The author of the thesis defines a procedure to characterize the replacement policy used in the cache. This characterization is called permutation vectors and

it derives the replacement policy after getting the response of an input vector of queries. Unfortunately, this characterization can not deal with all the block replacement policies. As an example, Most Recently-Used policy (MRU) is not possible to define using the permutation vector model because it is too limited. It is reasonable to find a new way to derive which replacement policy is managing the blocks in the cache memory.

A Register Mealy Machine (RMM) is a finite state machine extended with data registers, with defined states and transitions including conditions between them. With this model it is possible to define via a finite state machine how the data structure is working. LearnLib is a framework based in the work *Inferring semantic interfaces of data structures* [4] and it sets an environment to make characterizations by RMM of a given black-box system requests.

In this thesis the objetives are:

- Test if the LearnLib environment will work with possible block replacement policies (Chapter 3), different block replacement policies were simulated and connected to LearnLib to verify the results of the framework. The policies simulated are *First-In First-Out* (FIFO), *Least Recently Used* (LRU), *Pseudo-LRU* (PLRU or Tree-LRU) and *Most Recently Used* (MRU). This policies are the common replacement policies in the processors. The simulation sends a request and this request is answered by the status of the cache memory: miss, the block was not in the cache memory, or hit, the block was in the cache memory.

- The second contribution of this thesis is to develop a function in C which takes a vector of blocks and sends requests to the cache memory to get back a vector indicating which ones where a miss or hit (hwquery, chapter 4). This is useful for a future developments of this project when LearnLib is used on a real hardware instead of simulations.

Chapter 2

Technical Background

2.1 Cache Memory

The memory hierarchy is divided basically in 4 levels in order from fastest to the slowest: CPU registers, cache memory, main memory and permanent memory. Registers are the fastest and the smallest memories in the hierarchy, the space increases in the next levels because the speed reduces, making the cost per bit lower [5] (Figure 2.1).

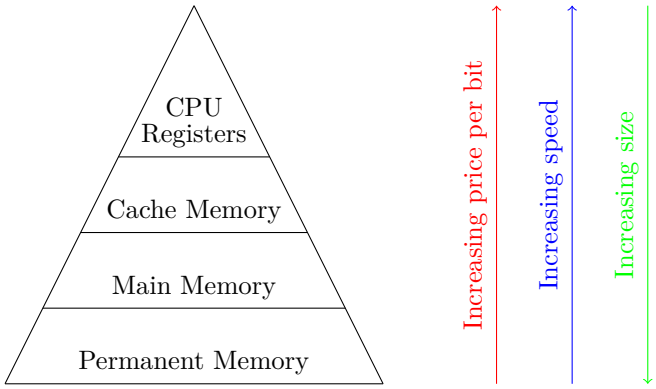


Figure 2.1: Memory Hierarchy, from [11]

This organization is useful because not all data is required by the program in runtime. Only few blocks of data will be as high as possible at the memory hierarchy. This concept is called *locality of reference*.

For accessing the data required by the program, each level in the hierarchy is mapping the next level. In the case of cache memory, it is mapping the main memory. The cache memory has three different mapping methods [11] [5] :

- Direct-mapping: A certain block of the main memory is located in modulo size of the cache memory. This mapping does not require a replacement

algorithm and has a simple hardware associated on it. The problem is the hit ratio, lower compared with other mapping methods. As a consequence, the performance drops in the case of accessing to locations with the same index.

- Associative: Much more flexible than direct mapping, allows to place any main memory block in any cache block position. In this method will be necessary a replacement policy to administrate. When all cache memory blocks are occupied, the policy will decide which block must be evicted when inserting a new one. For the limitations of the model, the cache memory size must be reduced to be possible economically.

- Set-associative: divides the cache in sets and each set will be placed determined number of blocks of the main memory. This number is set size or associativity. Each set has the same properties as an associative mapping method.

2.1.1 Hit and Miss

The request time of access to cache memory divides the requests in two groups [10]:

- Miss: the block was not in the cache memory. Then the block must be requested to the next level of cache or to the main memory. This request will delay the process.

- Hit: the block was in the cache memory. The time to request this block at the cache memory should be the shorter than a miss.

In French the word *caché* means hidden. In fact, the cache memory is one of the components black-boxed in the processor, the developer has not the chance of direct control. Unfortunately, miss can affect seriously to the runtime of a program [5].

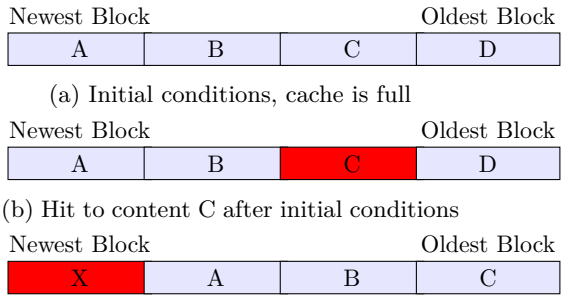
The only reference is a device furnished in some processor architectures: the performance counter, with this it is possible to measure the number of misses.

2.1.2 Replacement Policies

In the cases of associative and set-associative mappings, the replacement policy will decide which blocks must be evicted from the cache memory in a case of a miss.

FIFO

FIFO is acronym of First-In, First-Out. After all the blocks are occupied, in case a new block must be inserted, this policy takes the first block inserted and replaces for the new one. In a case of hit, the cache memory does not change the state (Figure 2.2).



(a) Initial conditions, cache is full

(b) Hit to content C after initial conditions

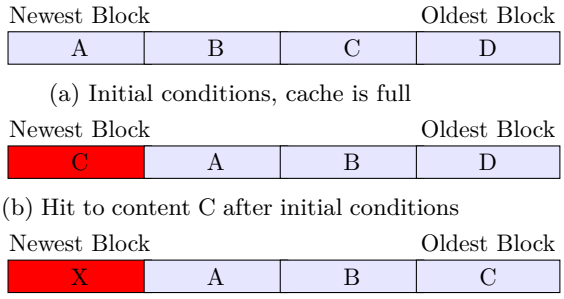
(c) Miss with content X after initial conditions

Figure 2.2: Cache memory of 4 blocks using FIFO

LRU

LRU means Least Recently Used and the idea of this policy is to replace the blocks less used in the cache memory. After the starting point, where all the blocks will be occupied, in case a new block must be accessed and it is not present in the cache memory, LRU will put this new block in the position of the oldest block without a hit. In case of a hit, the hit block is moved to the position of newest block (Figure 2.3).

The main difference between LRU and FIFO is after a hit. FIFO does not changes the order of the blocks in the cache, LRU instead puts the hit block into the newest position.



(a) Initial conditions, cache is full

(b) Hit to content C after initial conditions

(c) Miss with content X after initial conditions

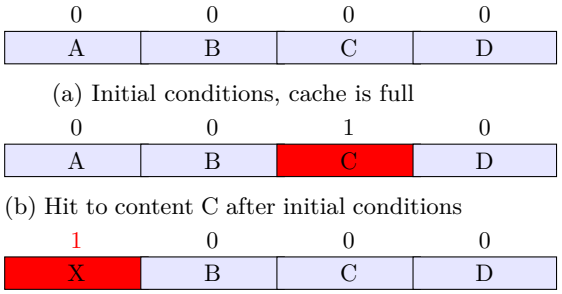
Figure 2.3: Cache memory of 4 blocks using LRU

MRU

Most Recently Used (MRU) is a policy using vector (state vector) where the state of each block is referenced by 0 or 1. 0 indicates the block can be evicted

and 1 indicates block was requested recently.

In a case of a hit, the position in the state vector pointing to the hit block will be changed to 1. In a case of miss, the policy evicts the first block tagged with 0 in the state vector. In booth cases if the state vector has all positions equal to 1, the policy transforms all positions to 0.(Figure 2.4)



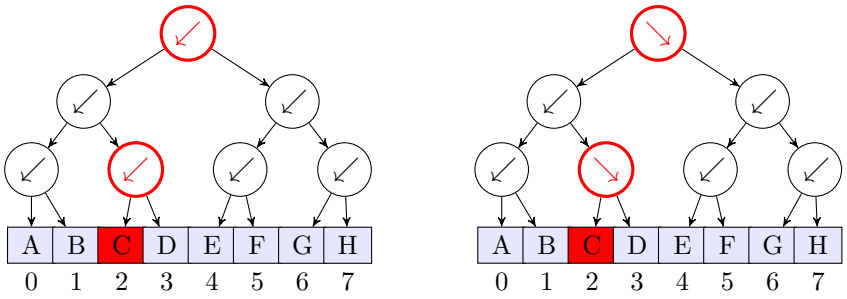
(a) Initial conditions, cache is full

(b) Hit to content C after initial conditions

(c) Miss with content X after initial conditions

Figure 2.4: Cache memory of 4 blocks using MRU

PLRU



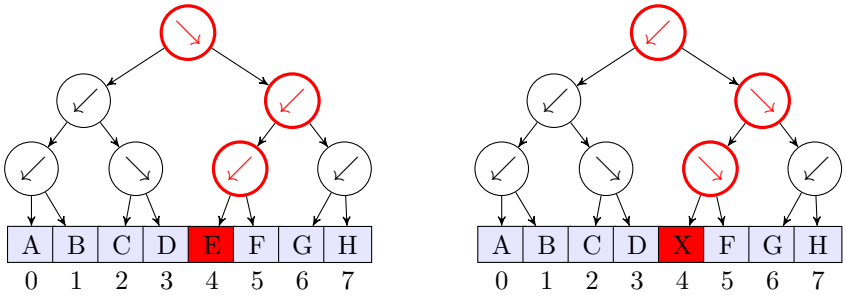
(a) Before hit on block containing C

(b) After hit on block containing C

Figure 2.5: PLRU replacement policy hit in a cache memory of 8 blocks

PLRU is one of the most common block replacement policies in the last generations of cache memory. Widely used by the industry, it is the most common policy in the commercial products of AMD and Intel [2].

Pseudo-LRU (also called Tree-LRU) consists in saving the status of the cache memory in a binary tree [12]. This tree is pointing to which position of the cache memory is the block to evict and be replaced in case of a miss.



(a) Before miss (b) After miss

Figure 2.6: PLRU replacement policy miss in a cache memory of 8 blocks

The tree has a number of inner nodes as blocks the cache memory has minus one. This complementary array to the cache memory is shorter than LRU or MRU.

To show the procedure of this replacement policy (Figures: 2.5,2.6), bits can be interpreted as arrows to left or right in child nodes. Each position of the tree contains an arrow pointing which block is target to evict in a case of a miss (Figure 2.6). In case of a hit, all the nodes on the path to the corresponding leaf will be checked to don't point to this leaf.

In case of a miss, the position evicted will be the leaf pointed by the tree. After the block is replaced, the algorithm will repeat the procedure of changing all nodes from the leaf to the top of the tree to don't point to the replaced block.

2.2 ChiPC

ChiPC is the tool developed by Andreas Abel in the context of the Master Thesis *"Measurement-based inference of the cache hierarchy"* [1]. In this thesis, different tools to infer the technical characteristics of the cache memory are described.

One of the important features of this tool is inferring the block replacement policy of the cache memory using permutation vectors.

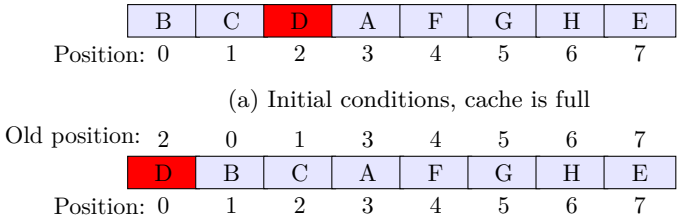
2.2.1 Permutation Vectors

Permutation vectors are a model for cache replacement policies used in *"Measurement-based inference of the cache hierarchy"* [1] and specified in *"Measurement-based modeling of the cache replacement policy"* [2]. It works for replacement policies where the contents can be described solely as a vector, describing the hypothetical positions of the elements.

A group of Permutation Vectors Π_i^X describes for X a block replacement policy and i all the position changes happening when the cache has a hit on the way

number i. In the Figure 2.7 is an example of a 8 blocks cache with a LRU replacement policy, after and before of a hit in the element in the position 2. Above the blocks is the index of content in the block and above the position of the block. In 2.7.a is the status before the hit and in 2.7.b after the hit. The permutation vector is the next one:

$$\prod_2^{LRU} = (2;0;1;3;4;5;6;7)$$



(a) Initial conditions, cache is full

(b) Hit block listed as 2

Figure 2.7: Example of learning by PV for LRU in hit in content in position 2

To infer which policy is working on the cache memory, it is necessary to restrict the possible states of the cache memory caused by hit each one of the blocks in the cache memory as well as after a miss. The consequence of this restriction is to summarize in a different vectors which will be the possible cache memory states of a block replacement policy.

Each replacement policy has a specific evolution between the requests of different blocks in the cache memory. In [2] is given an example of three replacement policies for a cache memories of 8 blocks (Figure 2.8). It is easy to see each replacement policy has a particular fingerprint.

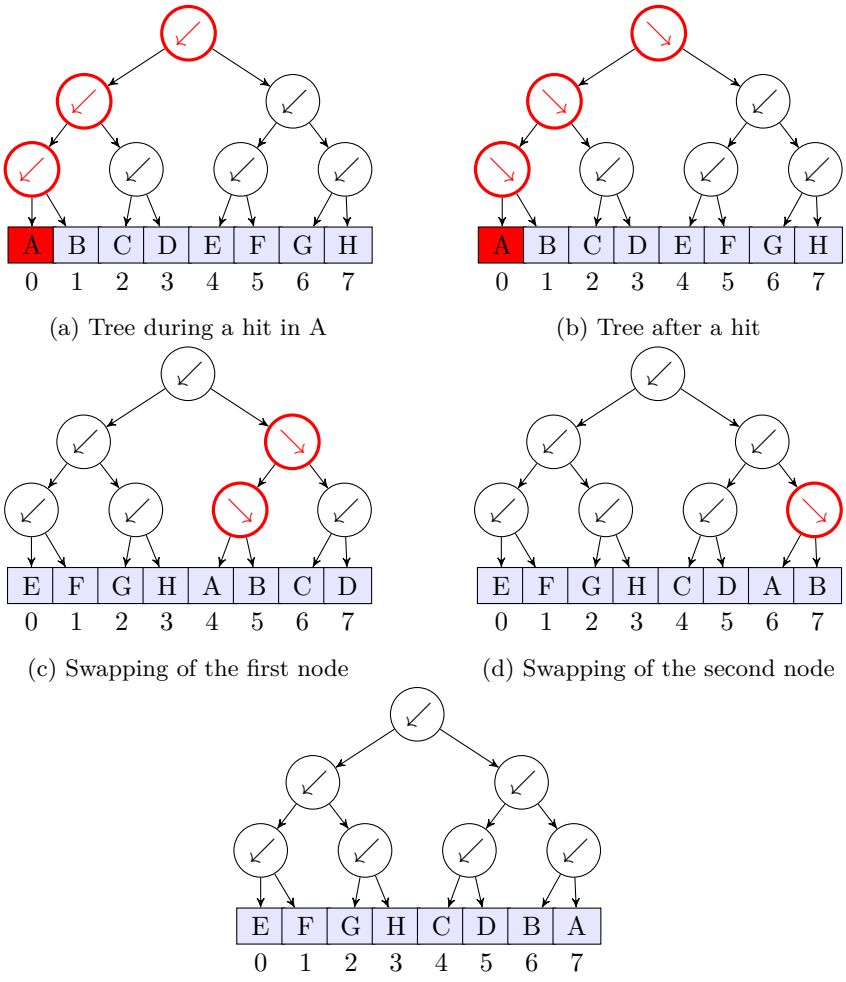
$$\begin{array}{lll}
\prod_0^{LRU} = (0;1;2;3;4;5;6;7) & \prod_0^{PLRU} = (0;1;2;3;4;5;6;7) & \prod_0^{FIFO} = (0;1;2;3;4;5;6;7) \\
\prod_1^{LRU} = (1;0;2;3;4;5;6;7) & \prod_1^{PLRU} = (1;0;3;2;5;4;7;6) & \prod_1^{FIFO} = (0;1;2;3;4;5;6;7) \\
\prod_2^{LRU} = (2;0;1;3;4;5;6;7) & \prod_2^{PLRU} = (2;1;0;3;6;5;4;7) & \prod_2^{FIFO} = (0;1;2;3;4;5;6;7) \\
\prod_3^{LRU} = (3;0;1;2;4;5;6;7) & \prod_3^{PLRU} = (3;0;1;2;7;4;5;6) & \prod_3^{FIFO} = (0;1;2;3;4;5;6;7) \\
\prod_4^{LRU} = (4;0;1;2;3;5;6;7) & \prod_4^{PLRU} = (4;1;2;3;0;5;6;7) & \prod_4^{FIFO} = (0;1;2;3;4;5;6;7) \\
\prod_5^{LRU} = (5;0;1;2;3;4;6;7) & \prod_5^{PLRU} = (5;0;3;2;1;4;7;6) & \prod_5^{FIFO} = (0;1;2;3;4;5;6;7) \\
\prod_6^{LRU} = (6;0;1;2;3;4;5;7) & \prod_6^{PLRU} = (6;1;0;3;2;5;4;7) & \prod_6^{FIFO} = (0;1;2;3;4;5;6;7) \\
\prod_7^{LRU} = (7;0;1;2;3;4;5;6) & \prod_7^{PLRU} = (7;0;1;2;3;4;5;6) & \prod_7^{FIFO} = (0;1;2;3;4;5;6;7)
\end{array}$$

$$\prod_{miss} = (7;0;1;2;3;4;5;6).$$

Figure 2.8: Permutation Vectors for LRU, PLRU and FIFO at associativity 8 defined in [2]

Normalization of PLRU

In the case of PLRU, the model in Permutation Vectors adds a special particularity: the normalization of the position of the cache memory blocks. The requirement of this normalization is only a restriction of the abstraction by permutation vectors. This abstraction can only express in one vector all possible combinations of the cache memory after a hit in certain block. This means that we need to remove the tree by normalizing it.



(a) Tree during a hit in A

(b) Tree after a hit

(c) Swapping of the first node

(d) Swapping of the second node

(e) After normalization

Figure 2.9: PLRU normalization after a hit on C in a cache memory of 8 positions

After an access to the cache memory, the vector containing the binary tree must be normalized: this implies to represent all changes in the tree in the Permutation Vector. The result is all i vectors of Π_i^{PLRU} will represent the same state tree.

The procedure for normalize the vector, after a hit in i position or miss, is swapping the positions of all the pointed elements in the cache memory from the top of the tree until the leafs until becomes same tree as the normalized tree. In the example, the normalized tree is all nodes pointing to the left (Figure 2.9).

2.2.2 Learning of Permutation Vectors

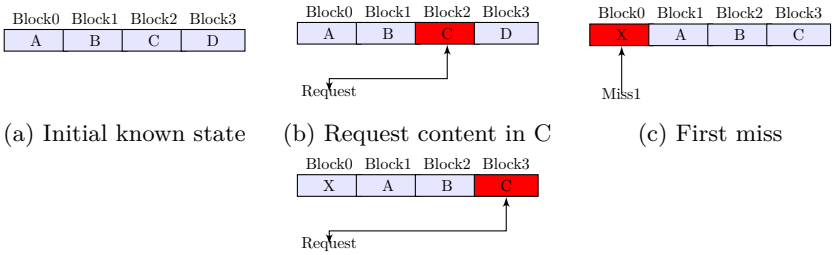
The procedure to get permutation vectors, described in [1], consists of four actions: setting a known memory state, hit on a particular block, realise a sequence of miss and realise a request the same block of the initial state.

The described three steps are repeated until the requested data is a miss. The second step is increased in each iteration. In order to get more trustable results, this process will be repeated and also the requested blocks will be alternated.

The next example consists in a known state where a 4 blocks cache memory has the data A,B,C,D as a known state and how ChiPC tool identifies where is each block after a miss.

Assume C is at block 3 (Figure 2.10):

1. Sets a known state (Figure 2.10a)
2. Do hit on C (Figure 2.10b)
3. Do 1st miss (Figure 2.10c)
4. Access to content C (Figure 2.10d)
    - Is a miss? yes, C was at block 3
    - Is a hit? yes, continue test



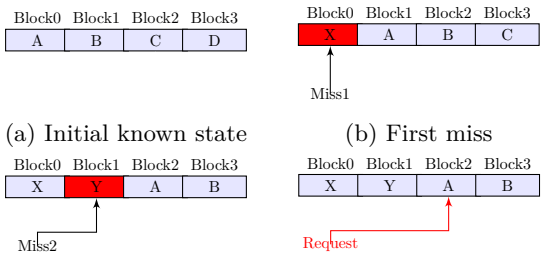
(a) Initial known state (b) Request content in C (c) First miss

(d) Request content in C

Figure 2.10: First iteration of the cache memory request

Assume C is at block 2 (Figure 2.11):

1. Sets a known state (Figure 2.11a)
2. Do hit on C
3. Do 1st miss (Figure 2.11b)
4. Do 2nd miss (Figure 2.11c)
5. Access to content C (Figure 2.11d)
    - Is a miss? yes, C was at block 2
    - Is a hit? yes, continue test



(a) Initial known state (b) First miss

(c) Second miss (d) Request content in C

Figure 2.11: Second iteration of the cache memory request

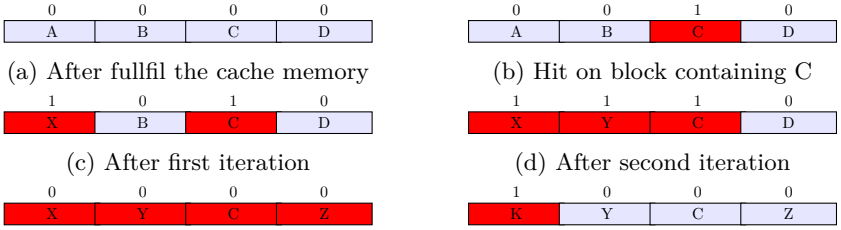
2.2.3 Restrictions

This definition is not flexible and can be impossible to define replacement policies where the vectors can be different with dependence of which block has a miss or hit.

One example of this restriction is with Most Recently Used (MRU). With Pseudo-LRU (PLRU) was possible to know the state of the tree using a normalization but is impossible to define in one vector all possible status of the vector state of MRU.

In the Figure 2.12 is the example of applying the learning proccess to MRU can not give a permutation vector. In the example, a cache memory with 4 blocks, the method of learning using a progresive number of miss can give a wrong result: the block is saved in the position 3 but can not be detected in less or equal number of iterations as the size of the cache memory.

The model limitation makes impossible to modelling MRU replacement policy into permutation vectors. MRU has more states than vectors can express. This is one of the most important basis to set the development of different techniques than permutation vectors to infer the block replacement policy in cache memories.



(a) After fullfil the cache memory

(b) Hit on block containing C

(c) After first iteration

(d) After second iteration

(e) After third iteration

(f) After fourth iteration

Figure 2.12: A example of problem for learning PV process in MRU

2.3 Register Mealy Machines

Mealy Machines are state machines, finite, with a series of inputs producing a series of outputs, inputs and outputs formed by words over finite alphabets. Created to abstract logical circuits in *"A Method to Synthesizing Sequential Circuits"* [9]. This concept was enhanced with registers in each state of the machine, helping to define the procedures in a data structure. The improved model is called Register Mealy Machines and is described in *"Demonstrating learning of register automata"* [4].

2.3.1 Model Description

The model is defined by:

- **Locations:** the same as a state in a finite automaton, location is the definition of a particular setting of the registers with transitions to other locations.
- **Registers:** used to store data from input words.
- **Transitions:** these elements are between states and defines the conditions to effectuate the transition from one location to other one. Each transition consists of:
  - **Input:** register or registers to verify by the guard to realize the operation to get the output.
  - **Guard:** in the transition, the guard is the condition for accept the input data to realize the operation and get the output.
  - **Operation:** after accepting the condition indicated on the guard, this field indicates the operation to realize in the registers of the location.
  - **Output:** final state of the register of the next location.

The graphical representation to join all of those concepts is represent in each *Locations* a particular state of the registers and a *Transitions* with the definition of the procedure to change of *Location* is in Figure 2.13.

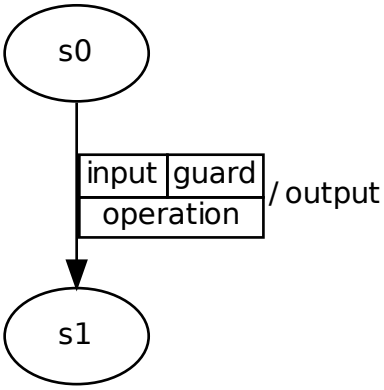


Figure 2.13: Simple automaton

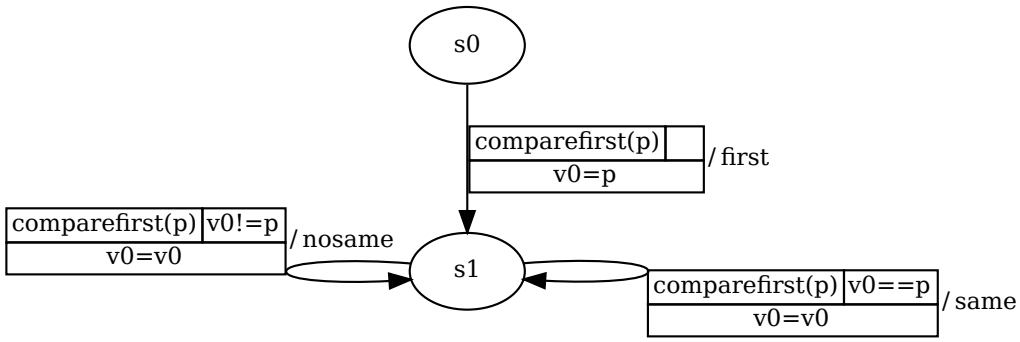


Figure 2.14: Evaluation of a number is the same as the first one introduced saving the last different introduced

As an example, the Figure 2.14 is an automaton to evaluate if the last input p is similar to the first one inserted.
The first input p will be saved in the register V_0. The first *Location*, $S0$, represents when the machine is started, and the second, $S1$, represents the comparison of the input value with the saved in the register V_0. The transition between $S0$ and $S1$ implies a change in the register, inserting the input number to V_0 and giving as output the message *first*.

The transitions in the second *Location* are two: one with a guard in case the inserted number input p is the same as in the register V_0 with *same* as output message and the other one when the input p is different, the output message is *nosame*. Booth transitions don't change the state of the register.

2.3.2 Learnlib

Learnlib is a framework implementing a learning process, form of inference [6]. This inference models the answers of the queries into a Register Mealy Machines (described in *"Inferring semantic interfaces of data structures"* [4]).

The framework is composed of the next components:

1. **AddAlphabet:** Sets an specific alphabet adding all symbols. This alphabet is to set a learning process, how to do the queries and which answers can be expected.
2. **Learn:** It is the main class in the learning process. Sends queries to QueryOracle to getting information about the analysed system.
3. **QueryOracle:** This process works in conjunction with Learn. Sets a driver to execute queries and getting answers.
4. **SearchCounterExample:** In a case of get an RMM, tries to get an Counterexample. In a case of getting a counterexample sends the process to Handle this counterexample. In a case of not getting more counterexamples, sends the process with the found automata to ShowObservations and ShowHypothesis.
5. **HandleCounterExample:** In a case of find a counterexample, process it and sends it again to the learning process to get another example RMM.
6. **ShowObservations and ShowHypothesis:** after SearchCounterExample gets a valid RMM, shows all information related to this hypothetical automata.

This components are illustrated in the figure 2.15 with the described interactions between them. This figure is from jABC, a visual interface to see the configuration of the LearnLib's learning process.

The required time to process a state depends of the number of counterexamples to handle. The number of counterexamples increases in order to infer the guards of each state [4]. In the worst case, the complexity for computing a RMM state will be exponential.

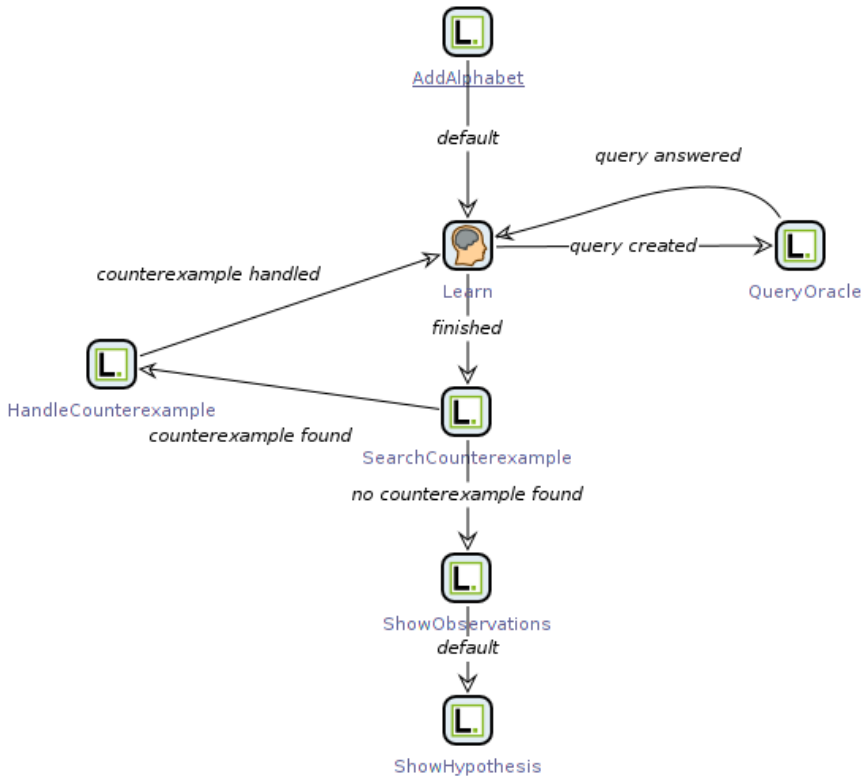


Figure 2.15: Learning Process schema from jABC

Chapter 3

Simulated Caches in LearnLib

If the last chapter explain the permutation vectors, technique to infer which replacement policy is working in the cache memory and the limitations of the model, now is time to explain the alternative, using a learning process and RMM with Learnlib.

For testing the possibilities of this alternative, several replacement policies will be simulated and connected with the LearnLib framework (Figure 3.1). The most known block replacement policies will be simulated by a Java developed classes. Also the LearnLib framework requires a configuration of different classes and parameters to work in conjunction with the cache simulated.

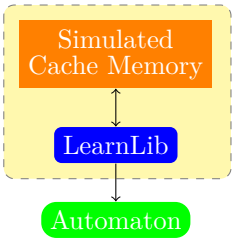


Figure 3.1: Diagram of the interaction between Learnlib and SimCache

The results must be verified and evaluated the processing time required. To verify the RMM given by LearnLib, a comparison with a *handwrite* automaton will be realized.

3.1 Cache Simulation

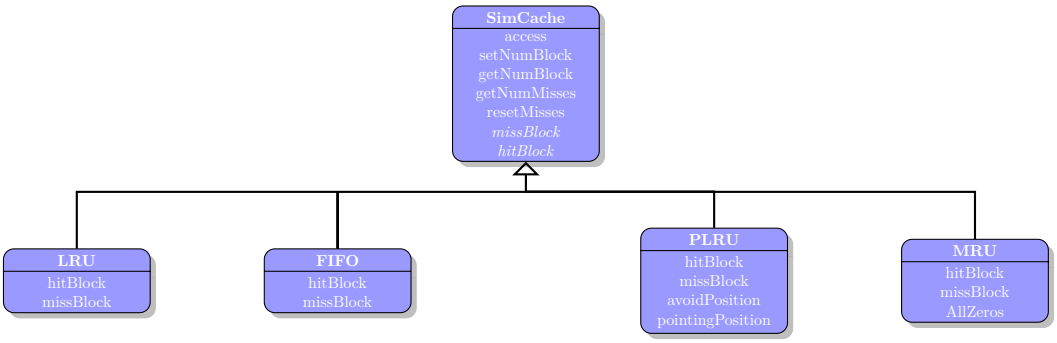
The memory simulation is implemented in JAVA to make easy to interact with LearnLib framework. Implements four different policies: FIFO, LRU, PLRU

and MRU.

All developed policies are using an abstract class implementing the basic access instruction of a cache memory, verifying if the block is in the cache memory or not and then implying the access is a hit or a miss.

The data structure used for implementing the cache saving structure is *ArrayList*, bundled in *java.util* class. This function allows easily adding and removing data of it.

3.1.1 Class Design



3.1.2 Implementation

The main class is an abstract class called *SimCache*. Implements basic functions to get the characteristics of the cache memory and access. This function checks if the requested block is in the cache memory, in case of yes, returns the *hitBlock* function in case of no, returns *missBlock* function.

These functions are abstract in the main class and they are implemented in the different subclasses, remarking the differences between the block replacement policies. In case of a hit, true is returned otherwise false.

The function access will be interfaced with LearnLib framework. The learning process is based on words, this words are a sequence of accesses to cache memory blocks. As an answer of this request, the function will return a sequence of hit and miss.

3.2 LearnLib Configuration

3.2.1 Configuration

Learnlib framework requires to set a configuration of different classes, DSAlphabet and MemCacheTestDriver, to specify how to interact with a external system. In the main class also are defined two important functions with param-

eters: unfolding and RandomWalkEquivalenceOracle, the parameters defined can improve or deteriorate the performance of LearnLib.

DSAlphabet

In this class is defined the alphabet of the learn process. In this case is formed by 3 symbols: access, hit and miss. Access is the function to realize a request to the cache memory. Hit and miss are the possible answers of the function, becoming then a output alphabet and also declared as instances.

MemCacheTestDriver

This class is the link between the LearnLib framework and the cache simulator, defining how to set the simulated cache and how interact with them. The function *step* sets how to do a request to the simulated cache memory and how to handle the answer.

unfoldSize

This variable is defined in the main class of the framework and it is related with the function responsible to compute the register automaton to a mealy machine. Uses the value of unfoldSize to a give a limit to the size to realize the Random Walk to find counterexamples.

RandomWalkEquivalenceOracle

This function is responsible of the internal working of the oracle. The oracle is responsible to create the words to query to the cache memory and for this reason is important to parametrize correctly the next parameters:

- Maximum number of tests
- Minimum length of the word each test
- Maximum length of the word each test
- Random seed

Some considerations about this parameters values.

The first one by report of the developer of LearnLib, the values of length of the word will be the same. This is because this part of the function is developed using exponential decreasing in probability of longer words. For this reason will be tested with longer words: two times the length of the cache memory and shorter words, with just the same length.

The second consideration it's about the number of tests, by default is set in 2000. If increasing this number can degenerate the performance, the decreasing also can distort the final automaton. For this reason, this number will not be modified.

The third consideration is about the random seed, by default it's 60000 but this value is changed to 60001. The choice of an odd value than an even is a good practise in the Pseudo Random Number Generators (PRNG).

3.2.2 Testing

Before effectuate a test of the performance, it is important to set default parameters and standard value in all tests. Decreasing the Unfolding number below than 5 can imply getting a not real replacement policy automaton. As example, the Figure 3.2 compared with 3.7.

Other considerations are the design limitations of the replacement policies. The most important limitation is the number of blocks of PLRU. This block replacement policy must have 2^n blocks for having all blocks in the tree the same level of nodes. Other limitation is the size of MRU, the cache memory must be longer than 2 blocks for having a realistic automaton.

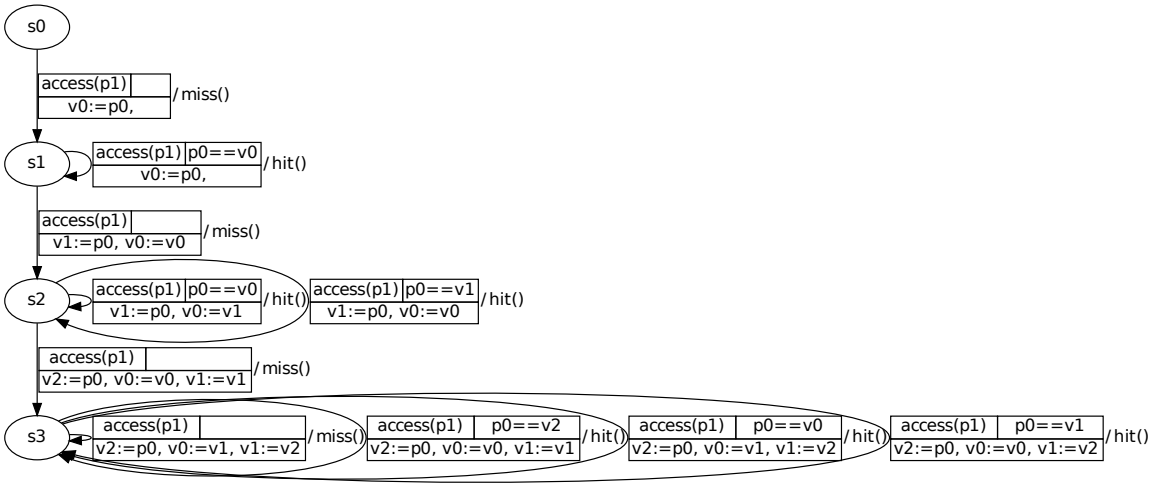


Figure 3.2: MRU of 3 blocks in case of low unfolding value (3)

After tuning the parameters, the runtime required in some sets becomes a problem. When the caches memories are longer than 5 blocks, the required time can be longer than 2 hours. All cache memories longer than 6 blocks in alls tests, after 48 hours were still working.

For this reason the cache memories larger than 6 blocks will be not included in the study of the performance of LearnLib. This limitation can get worse when the replacement policies will be not simulated.

3.3 Results

Two automata based on the simulator of replacement policies will be compared with a hand-made automata. The chosen policies are MRU of 3 blocks and PLRU of 4 blocks. The comparison will be useful to know if the result of LearnLib learning process is successful or not.

In another approach, we will be execute a different testing to measure the runtime required for getting an automaton from LearnLib. This tests are limited for the reasons of time of execution required, longer than 48-72 hours.

3.3.1 Automata

In the next figures, the RMM given by LearnLib Framework is compared with one designed manually, with the same block replacement policy and the same number of blocks. The selected policies are PLRU with a size of 4 blocks and MRU with a size of 3 blocks. The figures result of LearnLib's learning process are reordered to fit in the page.

The result schemas of LearnLib's learning process are raw and must be compared with a hand-written automaton to validate the learning process, if the RMM is representing the block replacement policy or not. The hand-written automaton is result of conceptualise mentally the block replacement policy with independence of the learning process.

Hand-written automaton use colors in order to make easier to read. Red to represent a miss, blue for a hit and grey for a location. Into the locations is represented the state of the cache memory and the state vector for MRU or state three for PLRU. This helps to compare with the results and validating them: after the LearnLib's learning process, the automaton is representing the simulated replacement policy.

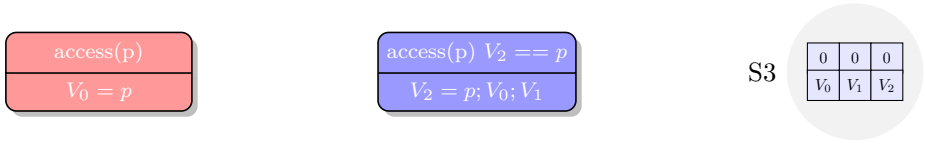


Figure 3.3: Example of miss transition, hit transition and location in the hand-written automaton

In the case of PLRU the size of simulated cache memory is 4 blocks. The figure 3.4 is the hand-written automaton and the figure 3.5 the result from LearnLib. This replacement policy is represented by an automaton of 6 locations and 19 transitions. Each transition implying a operation in the registers will be normalized as happens with the permutation vectors.

This automaton between the location S_0 and S_3 represents the cache memory states filling the empty cache memory. The location S_4, in booth cases, represents when the cache stays empty in the register V_3 instead the location S_5 is when all cache blocks ($[V_0; V_1; V_2; V_3]$) are fulfilled. Once this position is reached, this location is a loop itself, updating the blocks saving the newest block in the register V_3 and moving the older blocks closer to the register V_0.

The other comparison between a hand-written automaton and a result from LearnLib is MRU policy in a cache memory with size of 3 blocks. The figure 3.7 is the result after the learning of the simulated policy and the figure 3.6 is

the hand-written automaton.

The locations between S_0 and S_3 are states representing since is empty until all 3 blocks are fulfilled with data. The location S_3 is also when the state vector is again all 0. In the MRU policy after all blocks are tagged with 1 after a insertion or hit,the state vector returns to default state where all are 0.

After this location, exists three possible vector states: hit or miss of the first block at location S_4 stand for 100 in the state vector, hit of the second block at S_5 representing 010 or hit of the third block at S_6 equivalent to 001. The next miss always will be in the lowest position tagged with 0 in the state vector, that means, in case of S_4 and S_5 will be the location S_7 symbolize the vector state 110 and in the case of S_6 will be S_8 represent 101.

To mentioned before states S_7 and S_8 also represents locations after a second hit: S_7 can be a hit in the second block and S_8 can be a hit in the third block. To this locations also can be a hit in the third block, this case is the location S_9 stand for state vector 101. This three states, S_7, S_8 and S_9 will return to the state S_3 after a hit or miss because the vector state will become all zeros again and S_3 symbolizes this state.

The biggest difference with other policies is the higher number of locations and states compared with the same number of blocks in the cache memory: MRU automaton is 10 locations and 34 transitions. This can be a big problem because the problem can scale quickly of order as increases the number of blocks in the cache memory.

The two versions has the number of locations and transitions with the same characteristics. It is only a small difference: the transitions are normalized in the LearnLib automaton. The hand-written version has not any normalization but in learning process automaton, the locations are normalised to have always the newest block in the highest position (V_2) and the oldest block in the lowest position (V_0) of the cache memory.

PLRU automaton number of locations is lineal, only a k number of locations are required for fulfil the cache and effectuate the permutation of the blocks when is full.

The number of locations for MRU automaton is defined by formula $S = 2^k - 1 + k$, where k is the number of blocks. The first part of the formula, 2^k, is the operation to get the number of combinations of the vector state. The second part, $1 + k$ are the necessary locations to fulfil from an empty cache memory.

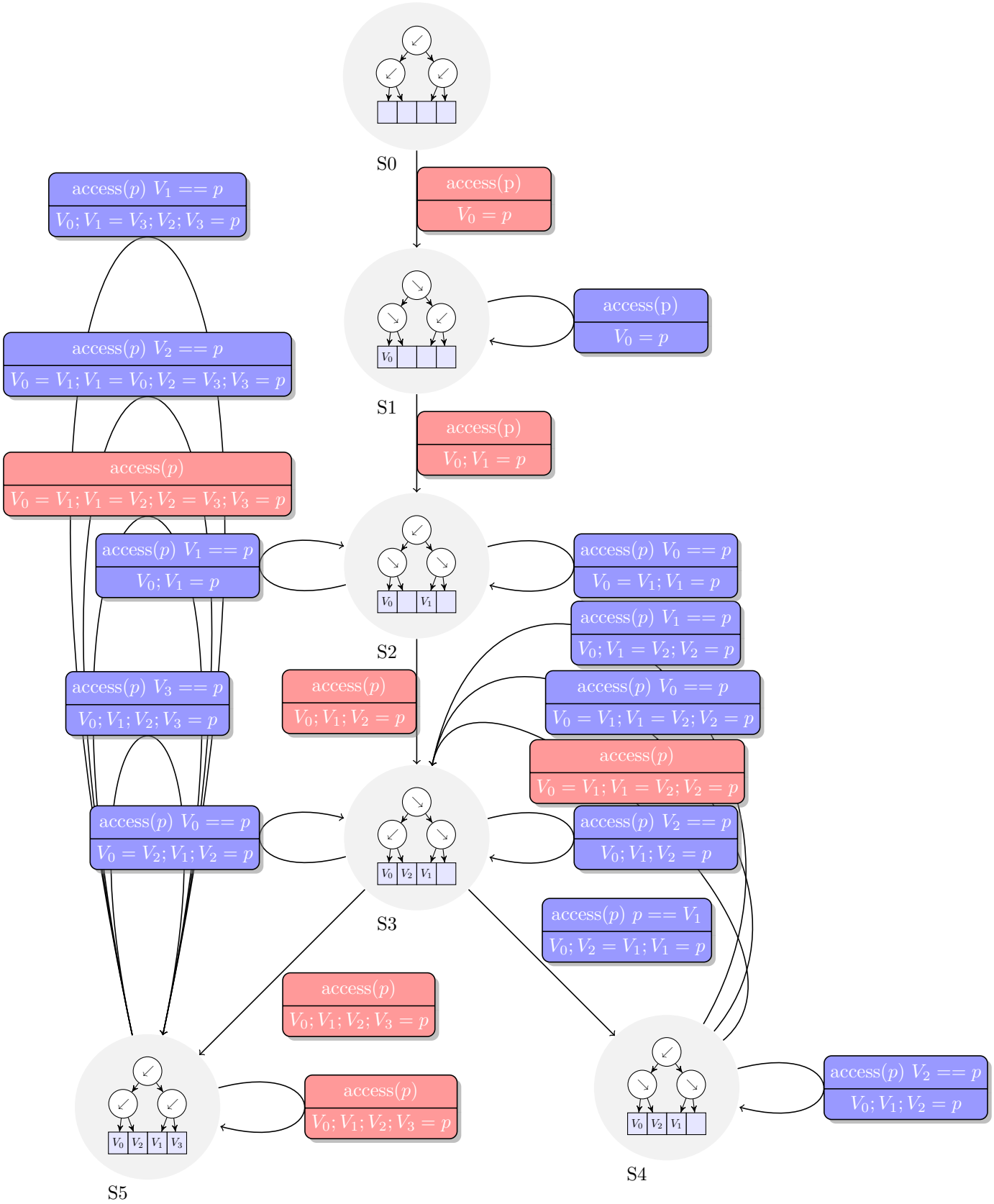


Figure 3.4: RMM of PLRU with size of 4 blocks (hand-written)

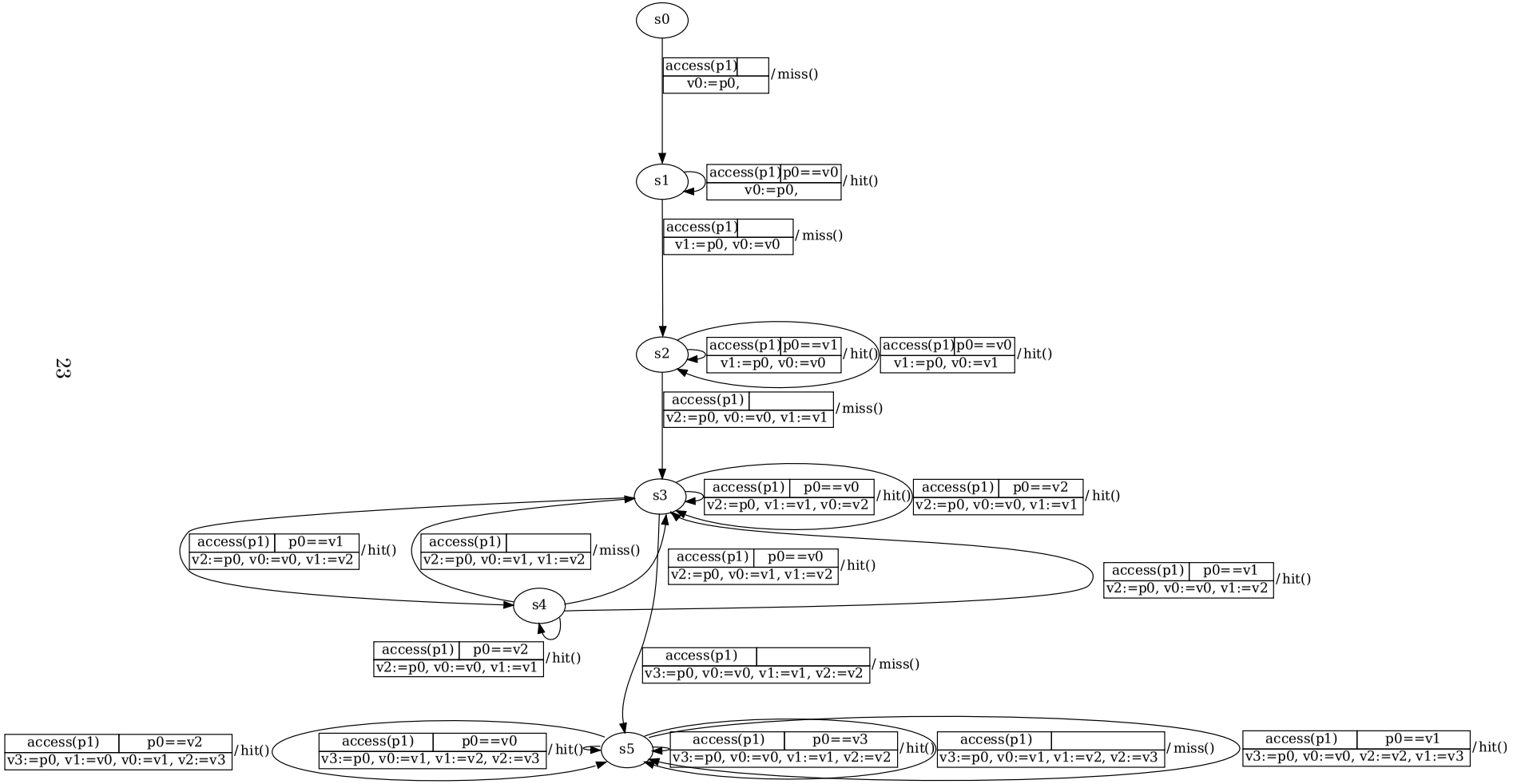


Figure 3.5: Learning Process for PLRU with size of 4 blocks

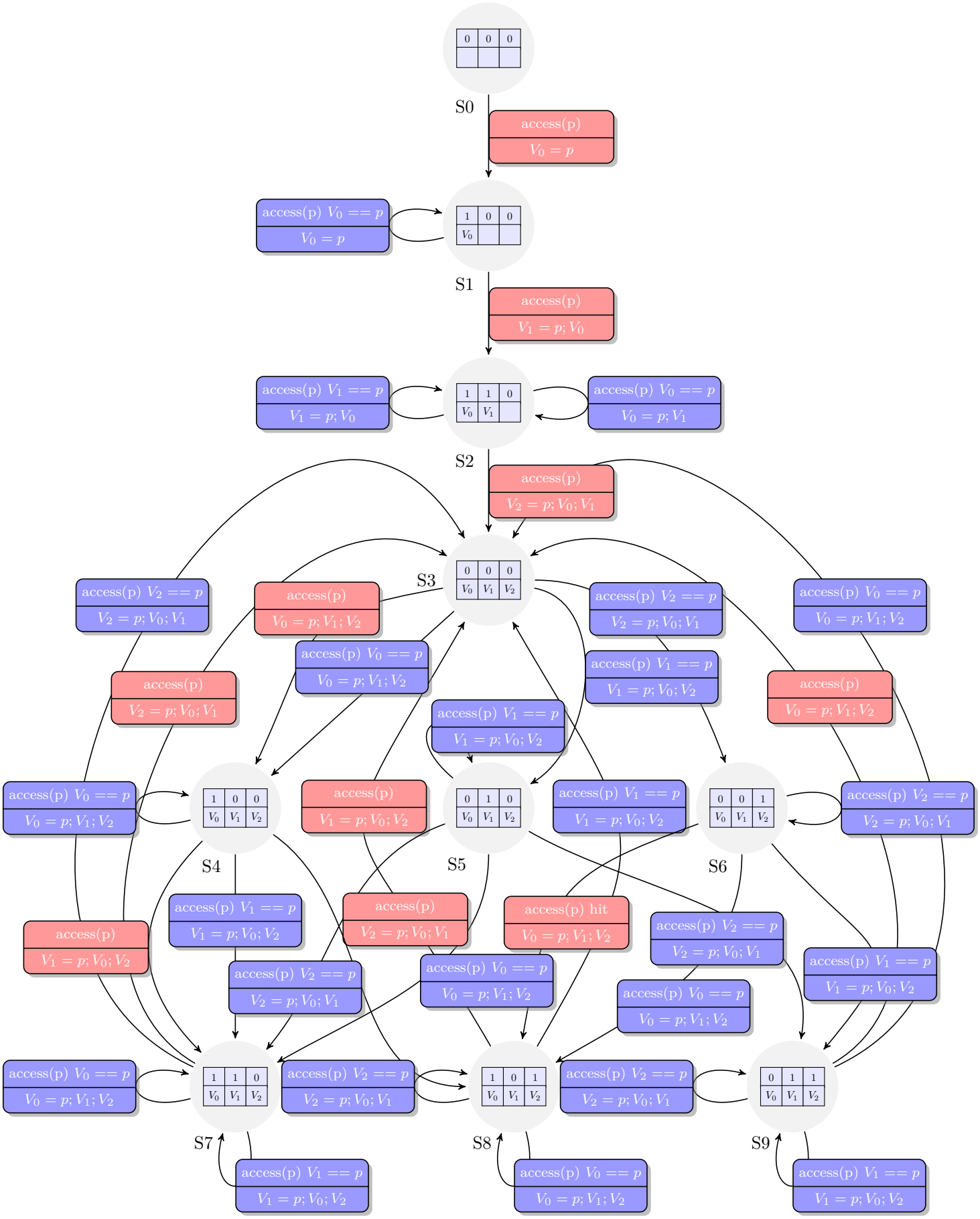


Figure 3.6: RMM of MRU with size of 3 blocks (hand-written)

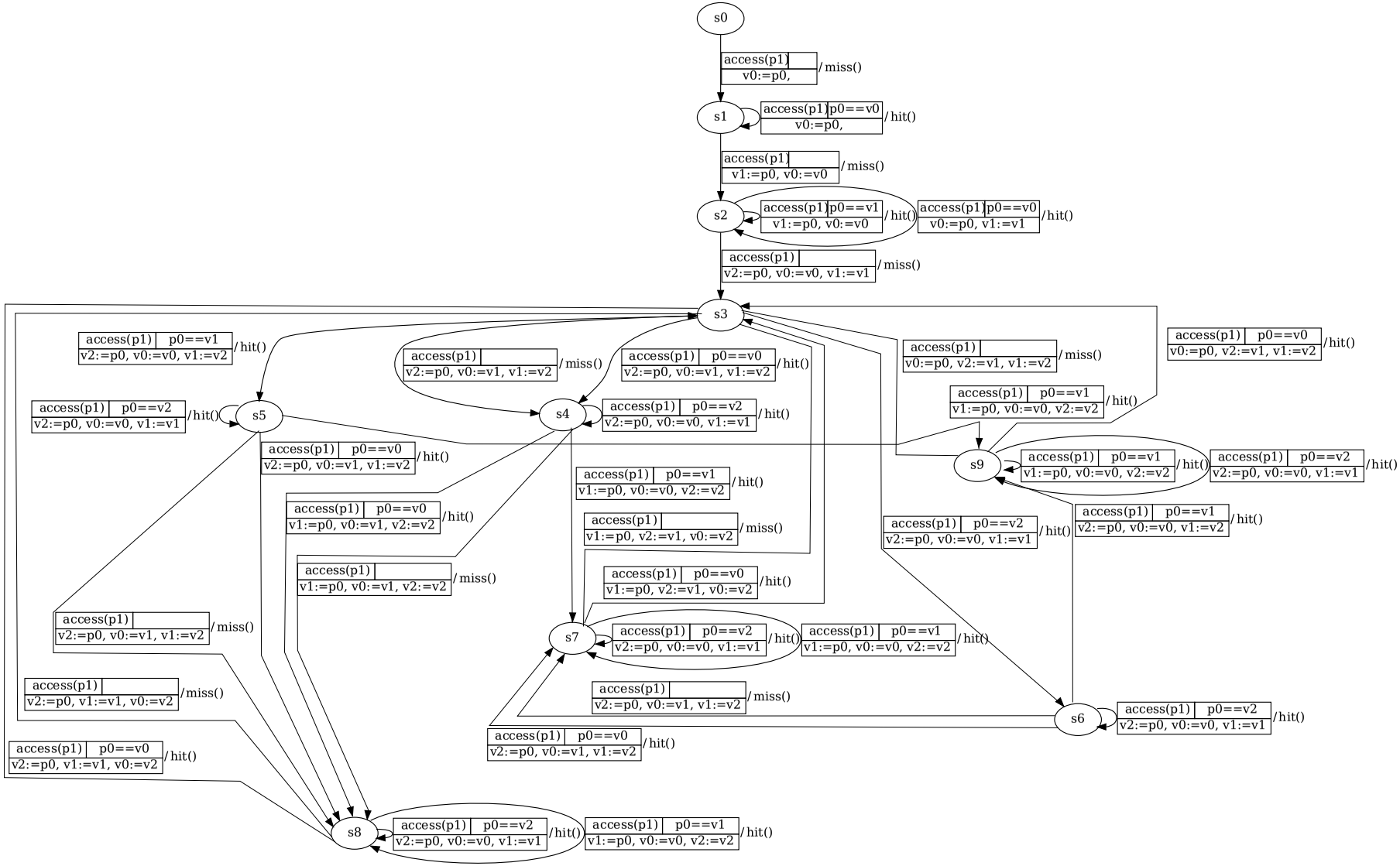


Figure 3.7: Learning Process for MRU with size of 3 blocks

3.3.2 Runtimes

The required LearnLib runtime to elaborate an automaton is a factor it must take into account. The required runtime can be a handicap for infer a block replacement policy. For this reason, a set of test (Tables 3.1, 3.2, 3.3 and 3.4) were executed with each replacement policy simulated (FIFO, LRU, PLRU and MRU) with sizes from 1 to 5 to have a sample of the required runtime for each one.

From each test two values are analysed: total queries and runtime. The first values gives the meaning of the second: who much time is required to effectuate certain number of queries. This is a good indicator to demonstrate the complexity predicted of creating the automaton.

To set out another factor concerning to LearnLib internal working, specially when it is effectuating the random walk of the learning process, it is the number of counterexamples to compare: the unfold number. All test are executed with two unfold numbers, five and seven. Lower than 5, for the selected tests, the LearnLib automaton is incorrect and it is not corresponding with the simulated replacement policy. Higher than 7, the selected tests requires more runtime and almost all tests went out of memory.

FIFO and LRU are the block replacement policies with lower runtime. The complexity of this policies is almost linear, the automata are always number of block of the cache memory plus one. The number of queries between the two policies is increasing at the same way, except the case of FIFO of 4 blocks with unfolding 5: It is part of the randomness in the LearnLib internal working because for unfolding 7 the increasing has not this artefact.

In the case of PLRU, the particularities of this replacement policy limits the number of blocks to a power of two numbers, limiting the runtime test to size of 2 and 4 blocks. The complexity of this problem is linear, $O(k)$, because after all blocks are fulfilled only one location more is required to permutate the blocks. PLRU takes more time than LRU and FIFO because the management of the block is more complex.

The formula to know the number of locations of MRU automaton also It will define the order of MRU with LearnLib: $O(2^k)$. It is an exponential time complexity problem, because all combinations of 0 and 1 in the state vector MRU will be represented. For this reason is required more resources and runtime than the other studied block replacement policies.

The complexity of MRU automaton also is remarkable in the number of total queries, more than 100 times higher than the other block replacement policies. Even the strategies to optimize the computer to run the tests, for a size larger than 4 went out of memory with the lowest unfold number possible.

Unfold Size equal to 5

Replacement Policy	2 Blocks	3 Blocks	4 Blocks	5 Blocks
FIFO	8	7	14	31
LRU	4	8	13	29
PLRU	8	X[1]	17	X[1]
MRU	8	86	176	Out of Memory

Table 3.1: Time to get RMM (times in seconds)

Replacement Policy	2 Blocks	3 Blocks	4 Blocks	5 Blocks
FIFO	30	85	309	109
LRU	18	36	13	109
PLRU	18	X[1]	923	X[1]
MRU	18	2042	57466	Out of Memory

Table 3.2: Number of total queries

Unfold Size equal to 7

Replacment Policy	2 Blocks	3 Blocks	4 Blocks	5 Blocks
FIFO	10	14	25	195
LRU	9	11	17	236
PLRU	9	X[1]	25	X[1]
MRU	9	50	426	Out of Memory

Table 3.3: Time to get RMM (times in seconds)

Replacment Policy	2 Blocks	3 Blocks	4 Blocks	5 Blocks
FIFO	30	85	341	670
LRU	30	61	107	178
PLRU	30	X[1]	198	X[1]
MRU	30	6990	40606	Out of Memory

Table 3.4: Number of total queries

[1]To have a balanced tree, the size of PLRU is limited to power of two numbers

Chapter 4

Hwquery

After simulating the block replacement policies, now is time to design a solution to interface a real systems cache into LearnLib framework. The program creating the queries to the cache will have as input a word integrated by a series of queries and it will generate as output the answer of the queries, hit or miss.

Unfortunately, Java works under a Virtual Machine environment, which is not possible to realize direct hardware queries. The programming language offering more control of low level instructions is C.

Developing applications with this propose with C is not usual and this also requires to take some considerations because GCC compiler can have a lot of optimizations compromising the functionality of the program after compiling. In this chapter is developed a possible design to interact real cache requests with LearnLib (Figure 4.1).

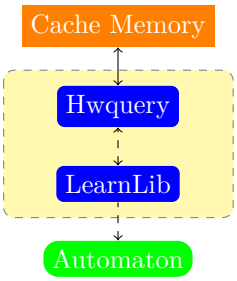


Figure 4.1: Diagram of the interaction between Learnlib and hwquery

4.1 Design

This request program must be designed fault tolerant for measuring the number of misses. To evaluate if an access to the cache memory is an hit or a miss in a real system can not be a request of only one block, must be a sequence of accesses to evaluate if is a miss or an hit comparing the statics between them.

This sequence of accesses will be called word. For an input word, this word will be fragmented request by request from the second value until completing the testing.

For this shorter word in the iteration, ϕ_{test}, two additional words will be generated. The first one, ϕ_{hit} , changes the last value of the word for the same value of the before last position in the word. The second one, ϕ_{miss} , the last value of the word will be the largest value in all word plus one (Figure 4.3).

This three words will be tested individually. Each one will have a number of misses. To chose if was a miss or a hit, ϕ_{test} will be compared between ϕ_{hit} and ϕ_{miss}. If the value ϕ_{test} is near to ϕ_{hit}, the last query will be saved as a hit and otherwise, as a miss. This process will be repeated N times (N is set to 10000), to minimize the interference of mechanisms of prefetching in the cache memory as well of other mechanisms involving the memory management in the processor architecture (Figure 4.2).

Between each repetition of the test, the cache memory will be cleaned using the word ϕ_{clean}. This word is integrated by as long are the tests word by a numbers higher than the highest number in the test word. Without the chance of repeating a number existing in the testing words, we remove the possibility of get false misses.

4.2 Implementation

The most important function in the program is the request to the cache memory, *test_access* (Figure 4.4). The variables required in the inner loop of this function are specified to use in a registers of the processor to avoid requests to the cache memory.

It is not possible to get the number of misses with a basic C instruction or with a call to the Operative System. The Performance Application Programming Interface (PAPI), a hardware monitoring library, interfaces the Performance Counter where is possible to get the number of misses.

PAPI have implemented a system to read the status of the Performance Counter (PC) of the CPU, making easy for the applications to read the statistics saved in the PC like number of misses of the cache memory.

This application will require to read the number of misses from the PC. With PAPI this request must be used as an event. The query to the cache memory will be realized between the callbacks (opening and closing) of the event.

4.2.1 Main memory reservation

Under any operative system and programming code, It is not possible to do queries directly to the cache memory. This is one of the most important considerations for developing a query to the cache memory, only it is possible to do

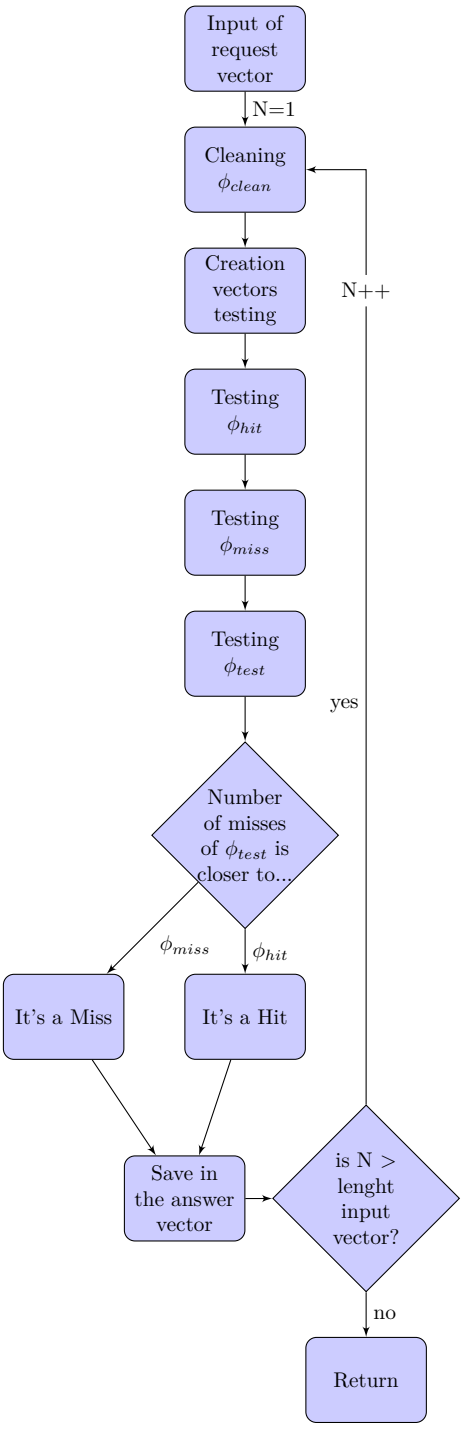
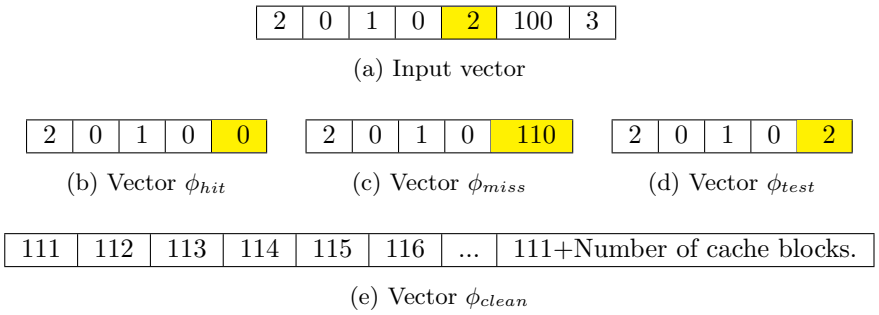


Figure 4.2: Flow diagram of *hwquery*

queries to the main memory.



(a) Input vector

(b) Vector ϕ_{hit} (c) Vector ϕ_{miss} (d) Vector ϕ_{test}

(e) Vector ϕ_{clean}

Figure 4.3: Different vectors *hwquery* for the fifth iteration of the request vector

The chosen solution is mapping the cache memory into the main memory (Figure 4.5). This mapping will require to know some characteristics of the memory structure, in essence: total size, size of block and number of blocks. With this information is possible to construct an address for the main memory pointing to a known address in the cache memory.

In C language, the instruction to map the main memory is *malloc*. This instruction will reserve the allocation for a variable in the C program the space specified by the user. In this case the instruction will be used to map all the main memory.

4.2.2 Cache memory access

The requests to the cache memory use the variable with *malloc*. Another design consideration is about the block replacement policies, all of them works in the block level. All query to the cache memory must point to the same set and word in the block.

The cache memory address (Figure 4.6) is formed by the next fields: Index, Offset and Tag. Tag is the address part pointing to the cache block, Index points to number of set in the cache memory and Offset specifies the word in the block.

The field Tag is in the most significant bits of the address. The user sends a query for a block, the number of block will be displaced as many bits long are the index and offset fields.

In C is possible to move the required bits with the symbol <<, in the code inside is the line responsible to create an address specifying a cache block in the same set, to the same word.

To know the bits required for the Offset and the Index it is necessary to know the cache specifications. In the case of the set-associative memory: the associativity, the total cache size and the cache block size. This data is used to calculate the size of:

```
int test_access(int phi[], size_t size, int n_samples, int numsets,
     int blocksz, int extra_value){

        initialize_papi(PAPI_L1_DCM);

        int num_miss = 0;
        register unsigned int x = 0;
        register int i;
        register int j;
        register int k;
        int num_sets = numsets;
        int block_size = blocksz;
        register int offset = log2(block_size);
        register int index = log2(num_sets);

        for ( j = 0; j < n_samples; j++){

                CLEAN_CACHE_N;

                start_papi();   // start miss counter

                for (i = 0; i < size; i++){
                                x += mem[phi[i]<<(offset+index)];
                }
                num_miss = num_miss + stop_papi();      // stop
                    miss counter
        }

        printf(" ",x);
        return num_miss;

        }
```

Figure 4.4: Function *test_access*

- **Index**: $\log_2(\frac{Number\,of\,Sets}{Block\,Size * Associativity})$
- **Offset**: $\log_2(Block\,Size)$

4.3 Testing output

The word tested (Figure 4.7) is a word mixing miss and hit. The input word is $[1\,2\,3\,1\,3\,2\,4\,5\,6\,7\,6\,8\,1\,2\,3\,6]$, this vector indicates in a sequence which blocks are requested. The expected output word is $[0\,0\,0\,1\,1\,1\,0\,0\,0\,0\,1\,0\,1\,1\,1\,1]$ where "0" means miss and "1" hit.

In this debugging output is possible to see the evolution of ϕ_{test} (labelled as Phi_T,T), ϕ_{miss} (Phi_M, M) and ϕ_{hit} (Phi_H, H) in each iteration. D is the difference between $(\phi_{test} - \phi_{miss})$ and $(\phi_{test} - \phi_{miss})$. This numbers shows the noise of measuring miss from the Performance Counter. This noise is important and remarks the importance of developing the design of ϕ words.

The chosen word do not infer any replacement policy, this will be task of other program wit a set of different words. The objective is evaluate if the program

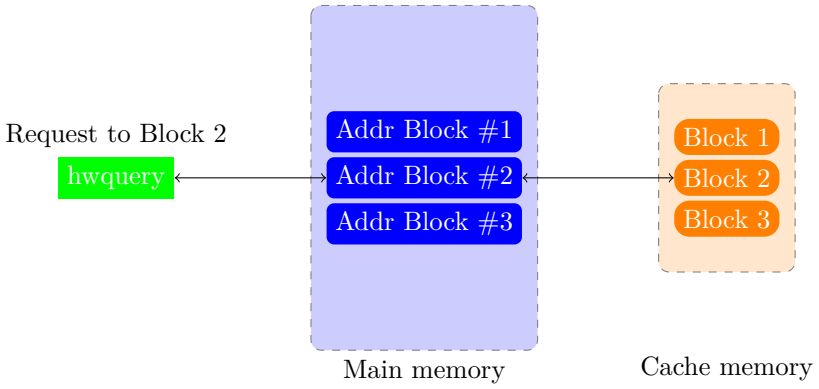


Figure 4.5: Interaction between hwquery and the cache memory

← (a) →	← (b) →	← (c) →

(a) Tag: bits pointing for a block in a set

(b) Index: required bits for addressing the total number of sets

(c) Offset: required bits for addressing a word in the block

Figure 4.6: Structure of main memory address

can detect when an element of the word is hit or miss.

The output word must be equal after all executions but the records of the Performance Counter of each ϕ word may differ between two executions. One problem detected is hwquery can give a wrong output word.

This problem happens when between two successive executions of hwquery. One explanation of this problem is the ϕ_{clean} word is not enough to cleaning the cache and the system of previous executions of hwquery.

```
1 => MISS:
Phi_H:(step: 2): 1  1
Phi_M:(step: 2): 1  9
Phi_T:(step: 2): 1  2
2 => MISS: H:3008528, M:4009832, T:4004324, D:995796 > 5508
Phi_H:(step: 3): 1  2  2
Phi_M:(step: 3): 1  2  9
Phi_T:(step: 3): 1  2  3
3 => MISS: H:4011764, M:5010076, T:5006556, D:994792 > 3520
Phi_H:(step: 4): 1  2  3  3
Phi_M:(step: 4): 1  2  3  9
Phi_T:(step: 4): 1  2  3  1
4 => HIT: H:5006746, M:6010758, T:5008693, D:1947 < 1002065
Phi_H:(step: 5): 1  2  3  1  1
Phi_M:(step: 5): 1  2  3  1  9
Phi_T:(step: 5): 1  2  3  1  3
5 => HIT: H:6000757, M:6006471, T:5013131, D:987626 < 993340
Phi_H:(step: 6): 1  2  3  1  3  3
Phi_M:(step: 6): 1  2  3  1  3  9
Phi_T:(step: 6): 1  2  3  1  3  2
6 => HIT: H:5027104, M:7016396, T:5026443, D:661 < 1989953
Phi_H:(step: 7): 1  2  3  1  3  2  2
Phi_M:(step: 7): 1  2  3  1  3  2  9
Phi_T:(step: 7): 1  2  3  1  3  2  4
7 => MISS: H:5025227, M:7020108, T:6025571, D:1000344 > 994537
Phi_H:(step: 8): 1  2  3  1  3  2  4  4
Phi_M:(step: 8): 1  2  3  1  3  2  4  9
Phi_T:(step: 8): 1  2  3  1  3  2  4  5
8 => MISS: H:6020347, M:7037679, T:8019903, D:1999556 > 982224
Phi_H:(step: 9): 1  2  3  1  3  2  4  5  5
Phi_M:(step: 9): 1  2  3  1  3  2  4  5  9
Phi_T:(step: 9): 1  2  3  1  3  2  4  5  6
9 => MISS: H:7072664, M:9022127, T:8904364, D:1831700 > 117763
Phi_H:(step: 10): 1  2  3  1  3  2  4  5  6  6
Phi_M:(step: 10): 1  2  3  1  3  2  4  5  6  9
Phi_T:(step: 10): 1  2  3  1  3  2  4  5  6  7
10 => MISS: H:9021449, M:10018417, T:10021040, D:999591 > 2623
Phi_H:(step: 11): 1  2  3  1  3  2  4  5  6  7  7
Phi_M:(step: 11): 1  2  3  1  3  2  4  5  6  7  9
Phi_T:(step: 11): 1  2  3  1  3  2  4  5  6  7  6
11 => HIT: H:10021581, M:11069234, T:10019808, D:1773 < 1049426
Phi_H:(step: 12): 1  2  3  1  3  2  4  5  6  7  6  6
Phi_M:(step: 12): 1  2  3  1  3  2  4  5  6  7  6  9
Phi_T:(step: 12): 1  2  3  1  3  2  4  5  6  7  6  8
12 => MISS: H:10004916, M:11002333, T:11003163, D:998247 > 830
Phi_H:(step: 13): 1  2  3  1  3  2  4  5  6  7  6  8  8
Phi_M:(step: 13): 1  2  3  1  3  2  4  5  6  7  6  8  9
Phi_T:(step: 13): 1  2  3  1  3  2  4  5  6  7  6  8  1
13 => HIT: H:11002177, M:12002945, T:11290161, D:287984 < 712784
Phi_H:(step: 14): 1  2  3  1  3  2  4  5  6  7  6  8  1  1
Phi_M:(step: 14): 1  2  3  1  3  2  4  5  6  7  6  8  1  9
Phi_T:(step: 14): 1  2  3  1  3  2  4  5  6  7  6  8  1  2
14 => HIT: H:11348853, M:12319956, T:11342899, D:5954 < 977057
Phi_H:(step: 15): 1  2  3  1  3  2  4  5  6  7  6  8  1  2  2
Phi_M:(step: 15): 1  2  3  1  3  2  4  5  6  7  6  8  1  2  9
Phi_T:(step: 15): 1  2  3  1  3  2  4  5  6  7  6  8  1  2  3
15 => HIT: H:11352035, M:12355746, T:11772652, D:420617 < 583094
Phi_H:(step: 16): 1  2  3  1  3  2  4  5  6  7  6  8  1  2  3  3
Phi_M:(step: 16): 1  2  3  1  3  2  4  5  6  7  6  8  1  2  3  9
Phi_T:(step: 16): 1  2  3  1  3  2  4  5  6  7  6  8  1  2  3  6
16 => HIT: H:11681205, M:12712036, T:11725371, D:44166 < 986665
```

Figure 4.7: Debug output for the word [1 2 3 1 3 2 4 5 6 7 6 8 1 2 3 6]

Chapter 5

Conclusions

The simulation of cache memories in junction with LearnLib Framework shows as a good alternative to Permutation Vectors to infer the block replacement policy used by the cache memory. This project was only focused with the hardware cache memories.

The different replacement policies (FIFO, LRU, PLRU and MRU) were tested with different sizes, from size equal 2 to size equal to 5 blocks. The result of each test is an actual working automaton. This LearnLib automaton is compared with an automaton with the same specifications designed manually. In case of unknown policy, this approach offers a way to characterize the replacement policy with an automaton.

The function for interfacing the cache request, hwquery, passed successfully several tests. Each test contains different words, from all miss to almost all hit, showing in all of them the expected result. These tests were executed in two different processor architectures with the same result.

Computers are nowadays multitasking systems. Infering the replacement policy used by the system with the number of misses, is hardly useful itself. During the run of the hwquery there are also running other tasks related with the operative system, but the use of ϕ_{words} and the number of samples removes the noise caused by the other tasks running in the system.

Besides the good results, this alternative also has some weak points. The improvement of Learnlib's performance is a requirement to be able to infer real block replacement policies from a cache memories of commercial systems.

The required runtime number is an important weakness. Some replacement policies, as the case of MRU, can have a exponential growth of all possible status. This can increases the time necessary to get an automaton.

The hwquery function must be improved and also have independence of the processor architecture. The errors related with the output word can penalize the performance of interfacing the cache memory for LearnLib.

About the architecture dependence, the developer must know some of the cache characteristics, the associativity and the block size, to run the program. This design also can be a problem to interface the cache memories without associativity design.

The approach explained in this project; the use of register automata to represent replacement policies, it is successful as a concept but it has the limitations explained along the chapters: the limitation of the learning process, the complexity of some policies and the difficulty of universalising the request to a real cache memory.

Those are the problems of the approach in this project, but there is another problem: the manufacturer. The ethics on discovering how a piece of hardware works are always motive of dispute. When a developer uses back-engineering to discover how the hardware works, he may have two future problems: a legal dispute between both parts, or the most common, the manufacturer changes some specifications in the next update of the hardware, making the developed software inoperative.

But the clever evolution on this subject would have been a less restrictive strategy. Nowadays, the developers of software are independent from the hardware designers, consequently the lack of communication between producers penalizes the full usage of the hardware by the software. Releasing the specifications of the hardware, would make possible its full-usage by the developers of software.

The learning process used in this project could be cheated by a manufacturer. Some examples of how the thesis approach could turn useless: Using non-deterministic policies, adding some additional interfacing to the cache memory, adding a trigger to randomize the block replacement or removing the cache requests, converting all of them to main memory requests. Still, after this inconveniences, the approach explained in this thesis is valid in all CPU generations, since the first one with Performance Counter to the last generation.

5.1 Future work

Interconnect hwquery with LearnLib

By the side of LearnLib Framework, the use of Java Native Interface (JNI) is part of the main solution and adding some modifications of the *MemOracleDriver* to create words of certain size.

In the other hand, about hwquery, It will be required to know the parameters of the cache memory as the block size, number of blocks, level of associativity. This can be fixed by using the parameters inferred by ChiPC. The hardware parameters saved by the operative system, as example, at the /proc/ files in Linux can be inaccurate because are based in the information provided by the hardware itself.

Improve the performance of LearnLib

In case of big cache memories, the performance of LearnLib can be poor and must be enhanced. The framework required more than 72 hours of runtime before finishing the learning process in the simulations of a caches longer than 6 blocks.

The accurate testing of the parameter values show us that problem is not only about tuning the parameters to the most optimal value. One possible strategy is to reduce the space explored by the LearnLib Framework. It can be positive to reduce the runtime required to obtain a result.

In fact, in [4] proposes the use of *symmetry reduction* in order to reduce the number of counterexamples to compute. This is not a universal solution. It must be tested with all block replacement policies and also studied how can deal with a real cache queries.

Match the replacement policy with a known policies

The system actually don't inform about which policy is managing the cache blocks. A small data base containing the automata related with a known block replacement policies, can be a solution to match it.

This also requires a canonical representation of all automata. This can be a disadvantage of using LearnLib, because the parameters can modify the results of the resulting automata. One possible way is to define a standard value for all parameter variables and keep it even if affects execution times.

Bibliography

[1] A. Abel, "Measurement-based inference of the cache hierarchy," Master's thesis, Saarland University, 2012. [Online]. Available: http://embedded.cs.uni-saarland.de/literature/AndreasAbelMastersThesis.pdf

[2] A. Abel and J. Reineke, "Measurement-based modeling of the cache replacement policy," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013, pp. 65–74.

[3] M. Merten, F. Howar, B. Steffen, S. Cassel, and B. Jonsson, "Demonstrating learning of register automata," in *TACAS*, 2012, pp. 466–471.

[4] F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson, "Inferring semantic interfaces of data structures," in *ISoLA (1)*, 2012, pp. 554–571.

[5] V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*. McGraw-Hill, 1996, pp. 224 – 238.

[6] M. L. Ginsberg, *Essentials of Artificial Intelligence*. Morgan Kaufman Publishers Inc., 1993, pp. 300 – 322.

[7] N. Guan, M. Lv, W. Yi, and G. Yu, "WCET Analysis with MRU Caches: Challenging LRU for Predictability," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2012, pp. 55–64.

[8] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.

[9] G. H. Mealy, "A Method to Synthesizing Sequential Circuits," 1955.

[10] B. Wilkinson, *Computer Architecture*. Prentice Hall, 1991, pp. 64–99.

[11] D. A. Peterson and J. L. Hennessy, *Computer Organization and Design*. ElSevier, 2007, pp. 468–510.

[12] J. J. DeMent, R. Hall, P. P. Liu, and T. Q. Truong, "Pseudo-LRU for a locking cache," Patent US 7 055 004, 05 40, 2006. [Online]. Available: http://www.google.com/patents/US7055004

[13] A. Junier, D. Hardy, and I. Puaut, "Impact of instruction cache replacement policy on the tightness of wcet estimation," in *Proc. of the 2nd Junior Researcher Workshop on Real-Time Computing, in conjunction to RTNS*, 2008, pp. 5–8.

Appendix A

Code of SimCache

```
abstract boolean missBlock(Object block);

abstract boolean hitBlock(Object block);

public boolean access(Object block){


    if (CacheBlocks.contains(block)==true) {

        return hitBlock(block);

    }else{

        numberMises++;
        return missBlock(block);

    }
```

Figure A.1: Code of Hit and Miss in main class

```
boolean hitBlock(Object block) {

    return true;

}

boolean missBlock(Object block) {

    if (this.CacheBlocks.size()<this.getNumBlock()){

        this.CacheBlocks.add(block);

    }else{

        this.CacheBlocks.remove(0);
        this.CacheBlocks.add(block);

    }

    this.numberMises++;
    return false;
}
```

Figure A.2: Code of Hit and Miss for FIFO

```
boolean hitBlock(Object block){

    this.CacheBlocks.remove(block);
    this.CacheBlocks.add(block);

    return true;
}

boolean missBlock(Object block) {

    if (this.CacheBlocks.size()<this.getNumBlock()){

        this.CacheBlocks.add(block);

    }else{

        this.CacheBlocks.remove(0);
        this.CacheBlocks.add(block);

    }

    this.numberMises++;
    return false;
}
```

Figure A.3: Code of Hit and Miss for LRU

```
    private void allZeros(){

        int allElementsZeroFlag = 0;

        for(int i=0; i < this.memBlocks; i++){

            if(state.get(i) == true){

                allElementsZeroFlag++;

              }

        }

        if(allElementsZeroFlag == this.memBlocks){

                        for(int i=0;i<memBlocks;i++){

                state.set(i,false);

            }
            allElementsZeroFlag = 0;

        }

    }

    boolean missBlock(Object block) {

       for(int i=0;i<this.memBlocks;i++){

            if(state.get(i) == false){
            this.CacheBlocks.set(i, block);
            state.set(i,true);
            allZeros();
            break;
            }

        }

       return false;

    }

    boolean hitBlock(Object block) {

        state.set(this.CacheBlocks.indexOf(block),true);

        allZeros();

        return true;
    }
```

Figure A.4: Code of Hit and Miss for MRU

```
boolean missBlock(Object block) {

    int missPos = pointingPosition();
    this.CacheBlocks.set(missPos, block);
    avoidPosition(missPos);
    return false;
}

boolean hitBlock(Object block) {

    int hitPos = this.CacheBlocks.indexOf(block);

    avoidPosition(hitPos);

    return true;
}

void avoidPosition(int pos){

    int path = pos + (this.memBlocks-1);

    while (path > 0){

        int parent = (path - 1)/2;
        tree.set(parent, path%2);
        path = parent;

    }

}


int pointingPosition(){

    int targetPosition = 0;
    int path = 0;

    while (path < this.memBlocks-1){

        if(tree.get(path) == 0) path = 2*path + 1;
        else path = 2*path + 2;

    }

    targetPosition = path - (this.memBlocks-1);

    return targetPosition;
}
```

Figure A.5: Code of Hit and Miss for PLRU

Curiously enough, the only thing that went through the mind of the bowl of petunias as it fell was "Oh no, not again". Many people have speculated that if we knew exactly why the bowl of petunias had thought that we would know a lot more about the nature of the Universe than we do now.

The Hitchhiker's Guide to the Galaxy
Douglas Adams