Software Requirements Specification

for

Cache write policies

Version 1.0 approved

Prepared by

Matteo Zini - 533197
Francesco Paladino - 530271
Raffaele Giannessi - 532978
Giulio Rossolini - 530244
Jacopo Malvatani - 533629

University of Pisa

23/05/2019

Table of Contents

Table of Contents ..... **ii**
Revision History ..... **ii**
1. Introduction **1**
1.1 Purpose 1
1.2 Product Scope 1
2. Overall Description **1**
2.1 Product Perspective 1
2.2 Product Functions 1
3. External Interface Requirements **2**
3.1 User Interfaces 2
3.2 Software Interfaces 2
3.3 Required Interfaces 4
4. System Features **5**
4.1 Miss policies 5
4.1.1 Description and Priority 5
4.1.2 Write-allocate functional requirements 5
4.1.3 Write no-allocate functional requirements 6
4.2 Hit policies 6
4.2.1 Description and Priority 6
4.2.2 Write-back functional requirements 6
4.2.3 Write-through functional requirements 6
5. Operations provided **6**
5.1 Set dirty 6
5.2 Check validity dirty 6
5.3 Check data validity 6
5.4 Check dirty 7
5.5 Invalid line 7
5.6 Load 7
5.7 Store 7
5.8 Write with policies 8
6. Test Case **8**

Revision History

Name	Date	Reason For Changes	Version
	23.05.2019	First version	1.0

1. Introduction

1.1 Purpose

This document describes the implementation of the cache write policies module. This is a subsystem of the cache, which is part of the microprocessor architecture design.

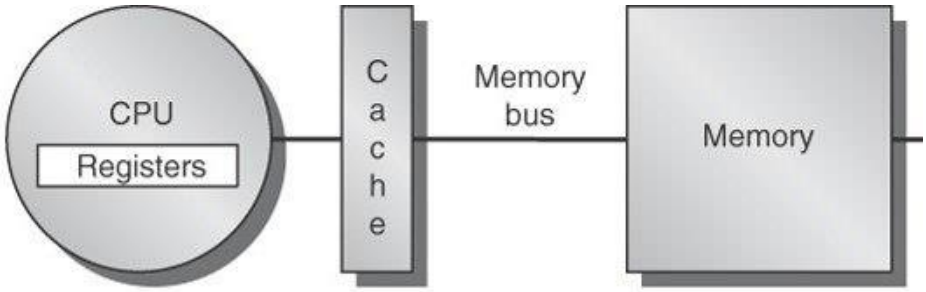
1.2 Product Scope

The software is intended to implement all the possible cache write policies in a microprocessor. The implemented policies are: write-through, write-back, write-allocate and write no-allocate.

2. Overall Description

2.1 Product Perspective

The module is an extension of the basic 1-way cache module of a typical architecture of a microprocessor. The module is a component of the set-associative cache module which is part of a multilevel cache system. The basic scheme representing the position in the whole architecture of the designed module is shown in the following picture.



2.2 Product Functions

This module is instantiated by the higher-level set-associative module for every cache way; the caller must provide to the constructor the desired write policy which must be followed.
In order to apply that policy, every write and read operation must be processed by this module and redelivered to the 1-way cache module.

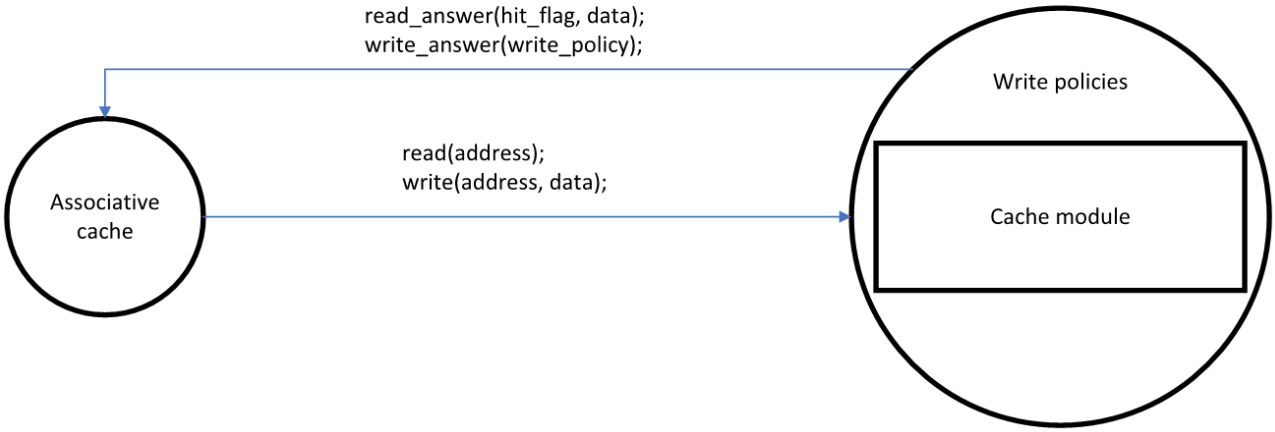
The operations that this module provides are:

- Constructor: decide the write policies
- Read: read a word from the cache
- Write: write a word into the cache

3. External Interface Requirements

3.1 User Interfaces

This module as a C++-class is an extension of the cache module



3.2 Software Interfaces

This module exchanges messages with the higher-level set-associative module in order to execute the requested operation. To implement this communication, we defined a structure for the request messages and another one for the response messages.

The structure of the request has these fields:

- OpType op_type: this is the type of the requested operation
- uint16_t address: this is the address on which the operation has to be performed
- uint16_t* data: this field contains the data to write

The fields “address” and “data” must be set depending on the “op_type”:

- SET_DIRTY: set the address of the cache line to set dirty
- CHECK_DIRTY: set the address of the cache line whose dirty bit must be checked

- CHECK_VALIDITY_DIRTY: set the address to check
- CHECK_DATA_VALIDITY: set the address to check
- INVALID_LINE: set the address to invalid
- LOAD: set the address of the block to load
- STORE: set the address where to store the block and its data
- WRITE_WITH_POLICIES: set the address where to store the word and its data

The structure of the response has these fields:

- In case of read
  - bool hit_flag
  - uint16_t* data
  - uint16_t address
- In case of write
  - WriteResponse wr

All those fields have different meaning depending on the previously requested "op_type":

- SET_DIRTY: address of the dirty cache line and the hit flag: 0 if the line is invalid
- CHECK_DIRTY: address of the checked cache line and hit flag: 1 if the line is dirty
- CHECK_VALIDITY_DIRTY: address of the checked cache line and the hit flag: 1 if the line is dirty, 0 either if the line is invalid or if it's not dirty
- CHECK_DATA_VALIDITY: address of the checked cache line and the hit flag: 1 if the line is valid
- INVALID_LINE: address of the invalidated cache line and the hit flag: 0 if the line was already invalid
- LOAD: address of the requested cache line, hit flag (1 in case of hit) and the read data (null in case of miss)
- STORE: address of the written cache line, hit flag (1 if the block was valid, 0 otherwise)
- WRITE_WITH_POLICIES:

```
if (hit_flag == 1)
        address: the address of the written word
else
        if (the block is invalid)
                address: the address of the word to be written
        else
                address: the address of the block which is occupying the requested
                location
wr: action that the caller must perform, depending on the write policy
```

The response message structure contains the field wr, which is meaningful when a WRITE_WITH_POLICIES operation was requested. It's an enum type and its value represents the action the caller must perform in order to meet the chosen write policies

- NOT_NEEDED: this value is returned when the requested operation is not WRITE_WITH_POLICIES
- PROPAGATE: this value is returned when the write-through policy is selected; so, the caller needs to propagate the write operation to the lower-level
- NO_PROPAGATE: this value is returned when the write-back policy is selected; so, the caller doesn't need to propagate the write operation to the lower-level
- LOAD_RECALL: this value is returned when the write-allocate policy is selected and the block containing the requested word is not in the cache; in order to complete the operation, the caller must load that block before calling the operation again
- CHECK_NEXT: this value is returned when the write-no-allocate policy is selected and the block containing the requested word is not in the cache; in order to complete the operation, the caller must propagate it to the lower-level

3.3 Required Interface

Since the cache write policies module extends the direct-mapped cache module, this interface is required:

- Load(address) : data
- Store(address, data) : result
- Invalid_line(address) : void
- Check_data_validity(address) : result
- Set_dirty(address, value) : void
- Check_validity_dirty(address) : value
- Resolve_index(address) : index
- Get_tag(address) : tag
- Check_dirty(address) : value
- Check_used(address) : value

4. System Features

4.1 Miss policies

4.1.1 Description and Priority

Through a parameter passed to the constructor of this module, it's possible to choose between two different behaviours in case of miss:

- Write-allocate
- Write no-allocate

4.1.2 Write-allocate functional requirements

When a word needs to be written, it can happen that there's not a copy of it in the cache memory. If this policy must be followed, the operation cannot be completed until the word is transferred into the cache. In order to do it, this module replies to the caller with a code which means that before serving the request, it must copy that word into the cache.

4.1.3 Write no-allocate functional requirements

When a word needs to be written, it can happen that there's not a copy of it in the cache memory. If this policy must be followed, the operation must "skip" the cache, and so it must be directly handled by the main memory. That's why this module replies with a code which means that the operation must be "propagated" to the lower level in the memory hierarchy.

4.2 Hit policies

4.2.1 Description and Priority

Through a parameter passed to the constructor of this module, it's possible to choose between two different behaviours in case of hit

- Write-back
- Write-through

4.2.2 Write-back functional requirements

When a word needs to be written, it can happen that there's a copy of it in the cache memory. If this policy must be followed, the word must be written into the correspondent cache location. In this case the word in the main memory is not updated.

4.2.3 Write-through functional requirements

When a word needs to be written, it can happen that there's a copy of it in the cache memory. If this policy must be followed, the word must be written both into the correspondent cache location and into the lower level in the memory hierarchy.

5. Operations provided

The cache write policies module provides the following operations.

5.1 Set dirty

Given an address, this operation consists of setting the *dirty bit* of the corresponding cache block only if it's valid (i.e. the block contains the requested word). In particular, the function *check_data_validity* belonging to the required interface is called to check if the given block is valid; in case of positive response, its *dirty bit* is set to 1. Considering the response structure: the hit flag is set to 1 if the cache block was valid, 0 otherwise; the address is set to the address of the given block; the *write-response* field is set to NOT_NEEDED.

5.2 Check validity dirty

Given an address, this operation consists of checking whether the corresponding cache block is valid (i.e. the block contains the requested word) and dirty. In particular, the function *check_validity_dirty* belonging to the required interface is called to check if the given block is valid and dirty. Considering the response structure: the hit flag is set to 1 if the cache block is both valid and dirty, 0 otherwise; the address is set to the address of the given block; the *write-response* field is set to NOT_NEEDED.

5.3 Check data validity

Given an address, this operation consists of checking whether the corresponding is valid (i.e. the block contains the requested word). In particular, the function *check_data_validity* belonging to the

required interface is called to check if the given block is valid. Considering the response structure: the hit flag is set to 1 if the cache block is valid; the address is set to the address of the given block; the *write-response* field is set to NOT_NEEDED.

5.4 Check dirty

Given an address, this operation consists of checking the *dirty bit* of the corresponding cache block. In particular, the function *check_dirty* belonging to the required interface is called to check if the given block is dirty. Considering the response structure: the hit flag is set to 1 if the cache block is dirty; the address is set to the address of the given block; the *write-response* field is set to NOT_NEEDED.

5.5 Invalid line

Given an address, this operation consists of invalidating the corresponding cache block only if it's valid (i.e. the block contains the requested word). In particular, the function *check_data_validity* belonging to the required interface is called to check if the given block is valid; in case of positive response, the function *invalid_line* belonging to the required interface is called to invalidate the given block. Considering the response structure: the hit flag is set to 1 if the cache block was valid; the address is set to the address of the given block; the *write-response* field is set to NOT_NEEDED.

5.6 Load

Given an address, this operation consists of loading the corresponding cache block only if it's valid (i.e. the block contains the requested word). In particular, the function *check_data_validity* belonging to the required interface is called to check if the given block is valid; in case of positive response, the function *load* belonging to the required interface is called to load the given block into an array. Considering the response structure: the hit flag is set to 1 if the cache block was valid (hit), 0 otherwise (miss); the address is set to the address of the given block; the *write-response* field is set to NOT_NEEDED; the data field is set to the loaded data array only in case of hit, NULL otherwise.

5.7 Store

Given an address and a data block whose size is the same of the size of a cache block, this operation consists of storing the given data block into the corresponding cache block. In particular, the function *check_data_validity* belonging to the required interface is called to check if the corresponding cache block is valid; then, the function *set_dirty* belonging to the required interface is called to set to 1 the *dirty bit* of the cache block; finally, the function *store* belonging to the required

interface is called to store the given data block into the corresponding cache block. Considering the response structure: the hit flag is set to 1 if the cache block was valid (hit), 0 otherwise (miss); the address is set to the address of the given block; the *write-response* field is set to NOT_NEEDED.

5.8 Write with policies

Given an address and a data word, this operation consists of storing the given data word into the corresponding cache block, at the corresponding offset (i.e. it modifies only the word in the whole block). In particular, the function *check_data_validity* belonging to the required interface is called to check if the corresponding cache block is valid.

If the response is positive (i.e. in case of hit), the function *set_dirty* belonging to the required interface is called to set to 1 the *dirty bit* of the cache block; then, the function *load* belonging to the required interface is called to load the given data block into an array; afterwards, the given data word is copied at the right position into the loaded cache block; finally, the function *store* belonging to the required interface is called to store the modified block into the cache. Considering the response structure: the address field is set to the address of the given block; the hit flag is set to true; the the *write-response* field is set to NO_PROPAGATE if the hit policy is WRITE_BACK, PROPAGATE if the hit policy is WRITE_THROUGH.

If the response is negative (i.e. in case of miss), the function *check_used* belonging to the required interface is called to check if the corresponding cache block contains some meaningful data. In case of positive response, the address field of the response structure is set to the tag part of the address of the data currently contained in the block (this is used by the set-associative cache in order to perform the replacement). In case of negative response, the address field of the response structure is set to the address of the given block. Considering the other fields of the response structure: the hit flag is set to 0; the *write-response* field is set to LOAD_RECALL if the miss policy is WRITE_ALLOCATE or to CHECK_NEXT if the policy is WRITE_NO_ALLOCATE.

6. Test case

In order to test the cache write policy module, we simulated in some sense the behaviour of the *orchestrator* module: in particular, we prepared a test file *cwp_test.cpp*, whose *main* function calls the *onNotify* function passing a request message. Also, we modified the function *sendWithDelay* of the class *module*, extended by our module: its original behaviour is to send the response message to the *orchestrator*; our version prints the values of the response message structure in order to understand whether the module works correctly or not (debugging with the output).

To make sure to test every possible behaviour of the module, the *cwp_test.cpp* test file instantiates 4 4096-byte caches by using the possible combinations of policies: *write-back* and *write-allocate*, *write-through* and *write-allocate, write-back* and *write-no-allocate*, *write-through* and *write-no-allocate*.

When the main test file is executed, the output is redirected to a text file. In order to simplify the checking process, we prepared a file with the expected output, supposing the correct behaviour of the module: after the execution the diff command is invoked on the two files to make sure they are exactly equal.