

关于 MESI 一致性缓存的简单 cache 模拟器实验

S2223 张超群 3122151007

目录

| | |
|--------------------------|----|
| 1. 实验介绍 | 1 |
| 2. 实验分析 | 1 |
| 2.1. cache 模拟器架构..... | 1 |
| 2.2. cache 模拟器配置..... | 1 |
| 2.3. trace 格式..... | 2 |
| 2.4. 输出格式..... | 2 |
| 3. 背景介绍 | 3 |
| 3.1. MESI 分析 | 3 |
| 3.2. 命令解释器 | 4 |
| 3.3. 指令解释器 | 4 |
| 4. 模拟器设计 | 4 |
| 4.1. 公共头文件 | 4 |
| 4.2. 命令解释器 | 4 |
| 4.3. CacheLine 模拟器 | 6 |
| 4.4. 缓存协议模拟器 | 6 |
| 4.5. 指令解释器 | 9 |
| 4.6. trace 打印机..... | 12 |
| 4.7. cachesim 主函数 | 12 |
| 5. 模拟结果 | 13 |
| 6. 实验感悟 | 14 |
| 参考文献..... | 15 |

1. 实验介绍

实现 MESI 策略的一致缓存 cache 模拟器。该模拟器仅有 2 个核，每个核只有一级 cache，cache line 为 64B，其余设定可自行发挥。要求在给定 trace 文件下，模拟器表现各个核在不同读写操作后的 MESI 状态。

trace 文件格式举例如下：

trace0.txt

```
-----  
0 0X00007C01  
1 0X00007C01  
0 0X00007C02  
-----
```

trace0 表示对核 0 的读写，文件中 0 表示读，1 表示写，后面为地址。

2. 实验分析

2.1. cache 模拟器架构

根据实验给出的参考文献[1]中的分析，cache 模拟器可从类型、层级、驱动模式等方面考虑。模拟类型可分为时序型和功能型。本实验仅要求展现 MESI 策略，并无带宽、延迟等时序方面要求，而且，时序型模拟器也远比功能型复杂。因此，模拟类型选择功能型。模拟层级可分为 Application 和 Full System，这两者因其过于复杂显然不在考虑范围内。因此，模拟层级能够达到运行 trace 文件即可。驱动模式可分为 Execution 驱动、Emulation 驱动、Trace 驱动，这里毫无疑问选择 trace 文件驱动。

尽管上述分析在简单的 cache 模拟器设计中显得多余，但在没有明确要求的程序设计中仍是必要的。

2.2. cache 模拟器配置

实验中明确给定的配置有 2 核、cache line 为 64B、一级 cache。

第一，仅考虑 2 核也不能完备地显示所有 MESI 的状态转移，例如，有 2 个 cache 为 S，1 个 cache 为 I，I 发生读或者写操作。然而，只需要 3 个核便可以展示所有的情况。因为在 M 或 E 状态 cache 存在的情况下，其余 cache 必是 I 状态，或者 S 状态 cache 存在，其余 cache 只能是 S 或 I 状态。那么 3 核及以上的配置会是更好的选择，一般处理器内的核数都是偶数个，所以选择 4 核配置。尽管可以把核数作为一个参数输入，但是 4 核以上的配置反而会显得多余。当然如果为了展示编程能力的话，可以把核数作为参数输入。

第二，cache line 大小决定了地址映射中行内偏移位数。但在 MESI 协议中，地址映射部分显然是有些多余的。因为通常为了模拟单一数据的一致缓存，总是简单地将每个核内的 cache 视为一个整体而忽略其中组或者组内 cache line 的细节，所以 cache 是否有对应内存块都是通过表示 cache 整体的 MESI 状态判断。MES 状态表示数据块在 cache 内，I 状态其表示不在 cache 内或者数据无效。那么一系列地址映射后，只是为了确定内存块是否位于发起读写请求的 cache 中而显得舍近求远，毕竟每个 cache 的 MESI 状态对编写这个模拟器的程

序员而言是可见的。除非真的编写一个隐去 MESI 状态而只能通过地址得知这些信息的模拟器。此外，考虑如果考虑地址映射，便不得不的考虑组内行数，替换策略等配置信息，而这些显然是对 MESI 模拟是无益并且不明晰的。

第三，通过第二点分析可知 cache 内的细节是多余的，那么 cache 的层次和相应容量也是没有必要的。

第四，既然上述分析得出地址是无效的，那么关于地址部分也不必考虑，所以也需要对 trace 格式进行修改，这会在 1.2.3 节中解答。

综上，需要考虑的模拟器配置仅有 4 核和 MESI 协议。

2.3. trace 格式

实验中给出 trace 例子有两个问题：1.trace 的文件名表示对核的输入，运行单一 trace 文件只能对单一核进行读写；2.trace 格式中的地址参数显得多余，一系列地址映射后只为了判断数据行是否在 cache 中以及处于何种状态，而这完全可由 MESI 协议中的 I 状态判断。因此，选择重新设计 trace 文件格式。即 trace 文件格式为：

```
-----
0 R0
1 W0
2 R1
3 W2
4 W3
-----
```

第一个参数为读写周期序数，第二个参数表示读写操作，第三个参数表示核号。如 0 R0 表示首先对核 0 读。

2.4. 输出格式

实验中没有给出输出格式，因此本格式参考了维基百科中关于 MESI 的介绍，如表 1：

表 1. 模拟器输出格式

| | Request | P0 | P1 | P2 | P3 | Data Supplier |
|---|-----------|----|----|----|----|---------------|
| | Initially | - | - | - | - | - |
| 0 | R0 | E | - | - | - | Mem |
| 1 | W0 | M | - | - | - | - |
| 2 | R1 | S | S | - | - | P0 |
| 3 | W2 | I | I | M | - | P0/P1 |
| 4 | W3 | I | I | S | S | P2 |

3. 背景介绍

3.1 MESI 分析

MESI 协议有许多介绍，本节不再赘述，但是仍有一些细节问题需要处理。

第一，主存是否有 MESI 状态？

尽管在介绍 MESI 时，很常见地把主存也标为 ESI 中的某些状态，但是在[2]中，表示主存不需要状态标志位。那么，这应该是从硬件上考虑的，但在实际的科普介绍中，给主存标明状态有助于理解。

第二，MESI 是哪种写策略？

[2]表示 MESI 使用 Write-back，而且在多数情况下这是默认的。但在某些情况，MESI 采用了 Write-once。Write-once 和 Write-back 在 MESI 中有着不同的作用机制，这将导致它们在状态迁移和写回主存两方面有很大不同。

在 Write-once 中，不写入主存只发生在 cache 由 E 变为 M 时。例如，假定 3 个内核 CPU0、CPU1、CPU2 状态为 I、S、S，CPU0 请求写，则状态变为 E、I、I，CPU0 进行一次内存写；当 3 个内核状态为 I、I、M 时，CPU0 请求写，则状态变为 E、I、I。首先 CPU2 接受到总线写请求，便把数据写回内存并广播数据到总线，CPU0 收到后，三者状态变为 S、I、S，之后 CPU0 再对数据写，并且写回内存，此时变为 E、I、I，一共发生两次内存写。网站[3]可以很好地展现上述 MESI 在 Write-once 下的变化，以及包括 cache line 替换的情况。

在 Write-back 中写入主存只有 2 种情况[4]。处于 M 状态的 cache 由于 I 状态 cache 发生 read miss 变为 S，或者 write miss 变为 I。其中，M 状态 cache 变为 S，它的 cache line 并没有被替换，但仍旧进行了写回，这应该是为了满足“S”状态的定义，不然主存中就没有对应的数据块了。令人困惑的是当 M 状态 cache 变为 I 时，它已经失效的副本仍旧被写回，而最新的数据块在别的 cache 中。那么失效副本有必要写回内存吗？在维基百科[5]和相关论文中都没有回答这个问题。我个人猜测，这是出于一种维持数据稳定的目的，内存至少保有一个最近有效的数据，那么即使最新的数据块出错（例如分支预测），也不必回滚到上次有效的状态，而是可以直接丢弃掉。MOESI 中定义的 O 状态，可以很好地解决上述第一种需要写回的情况。在 MOESI 中，M 状态的 cache 由于 I 状态 cache 发生 read miss，前者会变 O，后者变 S，此时不发生写回。但是第二种情况，仍需要写回。而少了一种写回情况，MOESI 的表现几乎处处好于 MESI。

Write-once 和 Write-back 在两种情况下的状态迁移有。cache 为 I 或 S 发生本地写，Write-once 下变为 E，Write-back 下变为 M。本模拟器采用 Write-back，状态转移如表 2：

表 2. MESI 状态转移

| State | 本地读 | 本地写 |
|-------|--|---|
| I | <ul style="list-style-type: none">• 其他 cache 都为 I 或空，不变。本地变 E• 有 cache 为 S，不变。本地变 S• 有 cache 为 E，变 S。本地变 S• 有 cache 为 M，变 S 并写回。本地变 S | <ul style="list-style-type: none">• 其他 cache 都为 I 或空，不变。本地变 M• 有 cache 为 S，变 I。本地变 M• 有 cache 为 E，变 I。本地变 M• 有 cache 为 M，变 I 并写回。本地变 M |
| S | <ul style="list-style-type: none">• 本地不变 | <ul style="list-style-type: none">• 其余 S 变 I。本地变 M |
| E | <ul style="list-style-type: none">• 本地不变 | <ul style="list-style-type: none">• 本地变 M |
| M | <ul style="list-style-type: none">• 本地不变 | <ul style="list-style-type: none">• 本地不变 |

3.2 命令解释器

命令解释器是基本所有模拟器都会有的功能，其形式如下：

Usage: ./sim [-hv] -s <s> -n <n> -t <tracefile>

其中 h, v 一般为无参选项，表示帮助信息和版本。后面一般为有参选项，即使没有输入也会有默认值，而 tracefile 这种输入文件类参数一般必须要有给定文件。

而为了读取命令，一般使用 getopt() 函数，例如上述命令表示为：

```
while( (opt = getopt(argc,argv,"hvs::n:t:")) != -1)
{
    if (opt == 't') tracefile = optarg;
}
```

具体细节使用可以查询网上信息。

3.3 指令解释器

指令解释器读取输入文件中的信息，一般使用 fscanf() 函数。举例使用如下：

```
while(fscanf(fp, " %c %xu,%d\n", &operation, &address, &size) > 0)
```

fscanf 读取每一行的 OP、地址、尺寸参数，直到文件尾。具体使用细节可以查询网上信息。

4. 模拟器设计

4.1. 公共头文件

common.h 存放一些可能需要用到的 C++ 头文件。

```
#ifndef _COMMON_H
#define _COMMON_H
#include <iostream>
using namespace std;
#endif
```

4.2. 命令解释器

由于核数固定为 4 核，也不涉及其他参数输入，故选项参数仅有输入文件名一项。从 c 命令解释器 argParser.h 如下：

```
#ifndef _ARGPARSER_H_
#define _ARGPARSER_H_
#include "Common.h"
#include <getopt.h>
class argParser
{
```

```

private:
string traceFilename;    //输入文件名

public:
argParser(int argc, char* argv[]);
~argParser();
void printHelp();
string getTracefile();
};
#endif

```

argParser.cpp 如下:

```

#include "argParser.h"
argParser::argParser(int argc, char* argv[])
{
    char opt;
    //使用 getopt()函数读取选项参数,optarg 为选项参数
    while( ( opt = getopt(argc,argv,"ht:") ) != -1)
    {
        switch (opt)
        {
            case 'h':
                printHelp();    //打印帮助信息
                break;
            case 't':
                traceFilename = optarg;
                break;
            default :
                break;
        }
    }
}

argParser::~argParser(){}
void argParser::printHelp()
{
    cout<<"Usage:"<< " ./my.exe [-h] -t <file>\n";
    cout<<"Options:\n";
    cout<<"-h          Print this help message.\n";
    /*
    cout<<"-p          Cache Coherency Protocol option.\n";
    cout<<"-n          Number of cores.\n";
    */
    cout<<"-t          Trace file.\n";
    cout<<"Examples:\n";
    cout<<"./my.exe -t mytrace.txt\n";
}

```

```

}
string argParser::getTracefile()
{
    return traceFilename;
}

```

4.3. CacheLine 模拟器

用一个缓存行来代替整个核内的缓存。因此只有核号和 MESI 状态（用一个字符表示）2 个成员数据。CacheLine.h 如下：

```

#ifndef _CACHELINE_H_
#define _CACHELINE_H_
class CacheLine
{
public:
    //这里的核号和状态都是公开的
    int core_id;
    char ccp_state; //缓存行的一致状态,显式使用
    /*
    uint8_t *c_data = new uint8_t[cacheline_size];
    还可以添加有效位、地址、数据等成员数据.....
    */
    CacheLine ();
    ~CacheLine();
    void init ( int );
};
#endif

```

用 ‘-’ 表示该行未被分配，CacheLine.cpp 如下：

```

#include "CacheLine.h"
CacheLine::CacheLine ()
{
    core_id = 0;
    ccp_state = '-';
}
void CacheLine::init( int ID = 0 )
{
    core_id = ID;
    ccp_state = '-';
}
CacheLine::~~CacheLine(){}

```

4.4. 缓存协议模拟器

缓存协议模拟器里应该有状态和状态转移函数,但是前者如果单独枚举为一个新的类型会比较麻烦,因此直接用字符表示表状态。Coherency.h 如下:

```
#ifndef _COHERENCY_H_
#define _COHERENCY_H_
#include "CacheLine.h"
#include "Common.h"
class Coherency
{
    /*
    enum state_t
    {
        '-',
        'I',
        'S',
        'E',
        'M'
    }
    */
public:
    void MESI_transition (CacheLine c[], string request);
};
#endif
```

其中 request 表示读写请求,例如 R0, W1。

根据表 2 可以得到 Coherency.cpp 文件如下:

```
#include "Coherency.h"
void Coherency::MESI_transition (CacheLine c[4], string request)
{
    const char* req = request.c_str();
    char req_op = req[0];
    int req_core = req[1] - '0'; //如果是两位数以上的核数,需要重新考虑
    if ( req_op != 'R' && req_op != 'W') return;
    if ( req_op == 'R')
    {
        switch( c[req_core].ccp_state )
        {
            case '-':
            case 'I':
            {
                int other_I = 0; //判断是否全为 INVALID 或未分配
                for(int i = 0; i < 4; i++)
                {
                    if( i == req_core ) continue;
                    switch ( c[i].ccp_state )
                    {
```



```

        case '-':
        case 'I':
            other_I++;
            break;
        case 'S':
            //有 cache 为 S, 不变
            break;
        case 'E':
            c[i].ccp_state = 'S'; //有 cache 为 E, 变 S
            break;
        case 'M':
            c[i].ccp_state = 'S'; //有 cache 为 M, 变 S
            break;
    }
}
if( other_I == 3)
    c[req_core].ccp_state = 'E';
else
    c[req_core].ccp_state = 'S';
break;
}
case 'S':
    break;
case 'E':
    break;
case 'M':
    break;
}
}
if ( req_op == 'W')
{
    switch( c[req_core].ccp_state )
    {
        case '-':
        case 'I':
        {
            for(int i=0; i<4 ; i++)
            {
                if( i == req_core ) continue;
                switch ( c[i].ccp_state )
                {
                    case '-':
                    case 'I':
                        break;

```

```

        case 'S':
            c[i].ccp_state = 'I'; //有 cache 为 S, 变 I
            break;
        case 'E':
            c[i].ccp_state = 'I'; //有 cache 为 E, 变 I
            break;
        case 'M':
            c[i].ccp_state = 'I'; //有 cache 为 M, 变 I
            break;
    }
}
c[req_core].ccp_state = 'M';
break;
}
case 'S':
{
    for(int i=0; i<4 ; i++)
    {
        if( i != req_core && c[i].ccp_state == 'S')
            c[i].ccp_state = 'I';
    }
    c[req_core].ccp_state = 'M';
    break;
}
case 'E':
    c[req_core].ccp_state = 'M';
    break;
case 'M':
    break;
}
}
}
}

```

这里主要要注意 I 状态 cache 发生读写的情况。

4.5. 指令解释器

从 trace 文件中要读取在某个周期的读写请求，得到读写请求数组 request[100]，以及指令总数 instnum，方便为循环使用 MESI 转移函数提供终止条件。traceReader.h 文件如下：

```

#ifndef _TRACEREADER_H_
#define _TRACEREADER_H_
#include "Common.h"
#include <fstream>
#include <sstream>
class traceReader

```

```

{
    private:
        string traceFile;
        ifstream trace;
        string request[100];    //存储指令流的数组
        int instnum;           //统计一共多少行指令
    public:
        traceReader();
        traceReader( string );
        ~traceReader();
        void setInst();
        string* getInst();
        int getInstnum();
};
#endif

```

traceReader.cpp 文件如下:

```

#include "traceReader.h"
traceReader::traceReader()
{
    traceFile = "";
    instnum = 0;
}
traceReader::traceReader( string file )
{
    traceFile = file;
    instnum = 0;
}
traceReader::~~traceReader()
{
    if ( trace.is_open() ) trace.close();
}
void traceReader::setInst()
{
    trace.open( traceFile, ios::in );
    if ( !trace.is_open() )
    {
        cout<<"Can not open the file: "<< traceFile <<endl;
        return;
    }
    string traceLine;    //读取 trace 的每一行
    string field;        //读取每一行的各个参数
    int fieldID;
    int cycle;

```

```

string req;
/*
trace 格式为
0 R0
1 W1
表示第一个指令周期里对 CPU0 读，第二个指令周期表示对 CPU1 写
*/
while( getline (trace, traceLine) )
{
    istringstream lineStream (traceLine);
    fieldID = 0;
    //不同参数之间以空格分开
    while( getline (lineStream, field, ' ') )
    {
        if(fieldID == 0)
        {
            cycle = atoi(field.c_str());
        }
        if(fieldID == 1)
        {
            req = field;
        }
        fieldID++;
    }
    request[cycle] = req;
    instnum++;
}
}

string* traceReader::getInst()
{
    return request;
}

int traceReader::getInstnum()
{
    return instnum;
}

```

这里并没有使用 `fscanf` 函数，而是采取一个比较笨的方法。第一个 `while()` 循环每次得到 `trace` 的一行数据，然后第二个 `while()` 循环开始读取一行中的 2 个参数，通过 `fieldID` 来区分第几个参数。

4.6. trace 打印器

这个比较简单，参考表 1 中的格式。`traceWrite` 头文件和源文件如下。

```
#ifndef _TRACEWRITER_H_
```

```

#define _TRACEWRITER_H_
#include "Common.h"
#include "CacheLine.h"
class traceWriter
{
public:
    void printState( CacheLine [], string );
};
#endif

```

```

#include "traceWriter.h"
void traceWriter::printState( CacheLine c[4], string request )
{
    cout<< request <<" ";
    for(int i=0; i<4; i++)
    {
        cout<< c[i].ccp_state <<" ";
    }
    cout<<endl;
}

```

4.7. cachesim 主函数

将前面的文件整合。cachesim.cpp 如下：

```

#include "Common.h"
#include "argParser.h"
#include "CacheLine.h"
#include "Coherency.h"
#include "traceReader.h"
#include "traceWriter.h"
int main(int argc, char *argv[])
{
    argParser argparser( argc, argv );
    string traceFile = argparser.getTracefile();
    traceReader tracereader( traceFile );
    tracereader.setInst();
    string* request = tracereader.getInst();
    int instnum = tracereader.getInstnum();

    CacheLine *c = new CacheLine[4];
    for (int i=0; i<4; i++)
    {
        c[i].init(i);
    }
}

```

```

    traceWriter print;
    Coherency protocol;
    cout<<"      P0  "<<"P1  "<<"P2  "<<"P3  \n";
    print.printState( c, "      ");
    for (int i=0; i<instnum; i++)
    {
        protocol.MESI_transition( c, request[i] );
        cout<<i<<"  ";
        print.printState( c, request[i]);
    }
    delete[] c;
    return 0;
}

```

5. 模拟结果

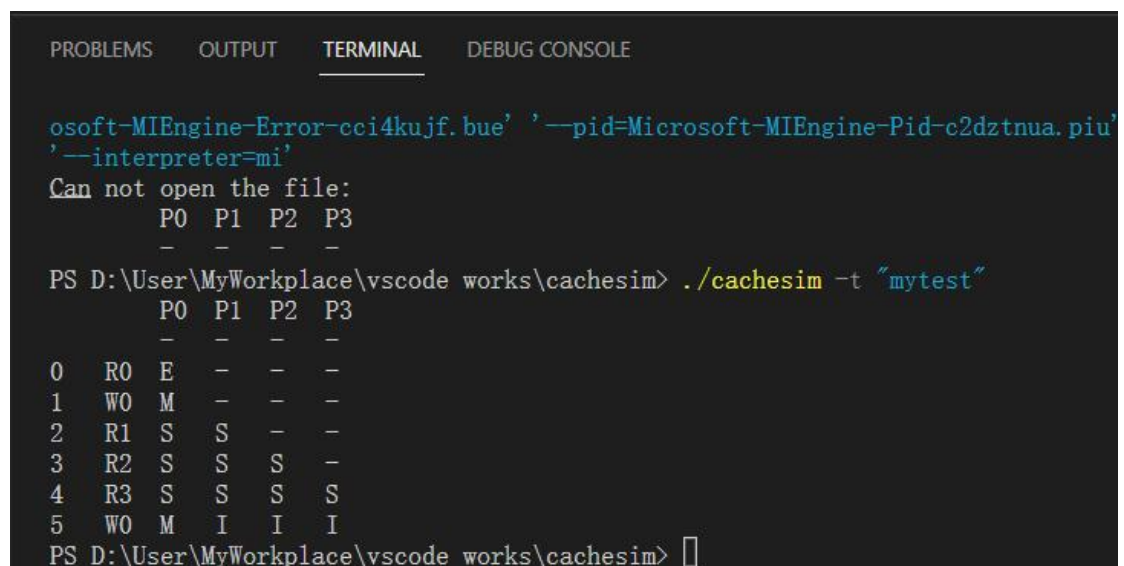
由于上面项目都是在 vscode 中编写的，在编译时选择将所有文件编译即可，避免了编写 makefile 文件。

trace 文件在如下情况时，输出如图 1、图 2 所示：

```

-----
0 R0
1 W0
2 R1
3 R2
4 R3
5 W0
-----

```



```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

osoft-MIEngine-Error-cci4kujf.bue' '--pid=Microsoft-MIEngine-Pid-c2dztnua.piu'
'--interpreter=mi'
Can not open the file:
      P0  P1  P2  P3
      -  -  -  -
PS D:\User\MyWorkplace\vscode works\cachesim> ./cachesim -t "mytest"
      P0  P1  P2  P3
      -  -  -  -
0   R0  E  -  -  -
1   W0  M  -  -  -
2   R1  S  S  -  -
3   R2  S  S  S  -
4   R3  S  S  S  S
5   W0  M  I  I  I
PS D:\User\MyWorkplace\vscode works\cachesim> 

```

图 1.

0. P0 读，读不命中，其他 cache 中都没有数据，因此从内存中写入 P0，P0 变 E。

1. P0 写，写命中，当前状态为 E，直接变为 M。
2. P1 读，读不命中，而 P0 中有数据，P0 发送数据至 P1，P0 由 M 变 S，P1 变 S。
3. P2 读，读不命中，而其他 cache 中有数据，故发送至 P2，P2 变 S。
4. P3 读，读不命中，而其他 cache 中有数据，故发送至 P3，P3 变 S。
5. P0 写，写命中，由于 P0 为 S，故还要将失效信号发送给其余 S 状态 cache，P1、P2、P3 变 I。

```

-----
0 R0
1 R1
2 R2
3 W0
4 W3
5 R2
-----

```

```

PS D:\User\MyWorkplace\vscode works\cachesim> ./cachesim -t "mytest"
      P0  P1  P2  P3
      -  -  -  -
0  R0  E  -  -  -
1  R1  S  S  -  -
2  R2  S  S  S  -
3  W0  M  I  I  -
4  W3  I  I  I  M
5  R2  I  I  S  S
PS D:\User\MyWorkplace\vscode works\cachesim> 

```

图 2.

0. P0 读，读不命中，其他 cache 中都没有数据，因此从内存中写入 P0，P0 变 E。
1. P1 读，读不命中，而 P0 中有数据，P0 发送至 P1，P0 由 E 变 S，P1 变 S。
2. P2 读，读不命中，而其他 cache 有数据，故发送至 P2，P2 变 S。
3. P0 写，写命中，P0 由 S 变 M，其余 S 状态 cache 变 I。
4. P3 写，写不命中，而 P0 中有数据，P0 将数据写回内存，并发送至 P3，P0 由 M 变 S，P3 接受数据后变 S，写完后 P3 由 S 变 M，并将失效信号发送其余 S 状态 cache，P0 由 M 变 I。（实际中，可能没有中间两个 S 状态，而是一步到位）
5. P2 读，读不命中，而 P3 中有数据，P3 将数据写回内存，并发送至 P2，P3 由 M 变 S，P2 接受数据后变 S。

以上两个测试展示了 MESI 中需要特别注意的 3 种情况：I 状态 cache 读时，若其他 cache 无数据，变 E，为此需要遍历所有核；I 状态 cache 发生读写，并且只有 M 状态 cache 有可用数据，M 状态 cache 需要写回；S 状态 cache 写时，需要遍历所有其余 S 状态 cache 使其变为 I。

6. 实验感悟

此次实验的主要难点有二：

- 一、确定 MESI 中的细节。由于课程中介绍 MESI 的篇幅短，且其中有些错误和不明晰

的部分。如没有考虑 I 状态 cache 发生本地读时，当其他 cache 没有该数据时，会变 E 而非 S 的情况。以及对 Write-once 中第一次的定义并不清楚。现在来看，只有对于 M 和 E 状态 cache，才会认为它不是第一次写。不过这些问题已经在 3.1 节分析清楚了。

二、对 cache 层次的抽象。

Cache line 是 cache 的子类，因此 cache line 中不能有 cacheline size 的信息，这个信息应当是从 cache 类继承的。尽管本模拟器并没有考虑这个，但合理地分配信息所属类是很关键的。例如，我曾在 Coherency 类中定义了 MESI 枚举状态，并且引用了 CacheLine 类作为 Coherency 类中 MESI 转移函数的一个变量，又在 CacheLine 类中引用了 Coherency 类，使用其中的枚举状态作为成员数据类型。这导致了编译失败。并且因为状态变成了枚举，所以输出枚举量都麻烦起来。其次，就是处理好各个类之间的接口，好在本模拟器的交换通路较少，即使不做规划也可以记住参数。但如果层次一复杂，可能需要实现画类似于 Verilog 线路图具现化。

实际编写起来，MESI 状态转移反而是很好编写的。

使用 C++ 编程主要原因是以前曾经反复研究过一个主存模拟器，所以会相对熟悉一些。但我也感受到 C++ 有太多细节需要注意。它很低级，数据在内存中如何分配可以自己决定，但是也要注意数据的释放，方方面面的考虑也有些过分耗费精力。

源代码均已上传 GitHub。

https://github.com/deagman/MESI_cache-simulator

参考文献

- [1] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. 2020. A Survey of Cache Simulators. ACM Comput. Surv. 53, 1, Article 19 (January 2021), 32 pages. <https://doi.org/10.1145/3372393>
- [2] Mark S. Papamarcos and Janak H. Patel. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In Proceedings of the 11th annual international symposium on Computer architecture (ISCA '84). Association for Computing Machinery, New York, NY, USA, 348–354. <https://doi.org/10.1145/800015.808204>
- [3] <https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/caches/MESIHelp.htm>
- [4] Dimitris Kehagias and Ioannis Raptis. 2016. An Interactive MESI Cache Coherence Simulator for Educational Purposes. In Proceedings of the 20th Pan-Hellenic Conference on Informatics (PCI '16). Association for Computing Machinery, New York, NY, USA, Article 61, 1 – 4. <https://doi.org/10.1145/3003733.3003765>

[5] https://en.wikipedia.org/wiki/MESI_protocol.