2024 年春计算机体系结构

Project 03: Cache 结构和行为的描述

1. Cache 的结构

指令 Cache 相当于一个“只读”的数据 Cache，因此接下来我们都以数据 Cache 为例进行说明。

代码 1 是数据 Cache 的结构定义。可以看出，数据 Cache 被定义为一个二维数组 dCache[][]，其中 DC_NUM_SETS 和 DC_SET_SIZE 都是常量，分别代表组数和每组的块数（相联度）。显然，当 DC_NUM_SETS 为 1 时，映射策略为全相联，而当 DC_SET_SIZE 为 1 时，映射策略为直接映像。

代码 1　数据 Cache 的结构定义

```
struct cacheBlk {
        int tag;
        int status;
        int trdy;
} dCache[DC_NUM_SETS][DC_SET_SIZE];
```

数组 dCache 的每个元素就是一个 Cache 块的标识部分，类型都是 struct cacheBlk。其中，tag 记录了 Cache 块的标识；status 为 Cache 块的状态，为 0 表示无效（invalid），1 表示有效（valid），2 表示“脏”（dirty）；trdy 记录了 Cache 最近一次被访问的时间，可用于实现 LRU 替换算法。

上面的 C 语言描述并没有定义 Cache 的数据存储部分。这是用 C 语言描述 Cache 的方便之处。这样用 C 语言描述的 Cache，在程序的执行过程中，数据依然保留在存储器中，不必进入 Cache。

类似地，还可以用 C 语言定义写缓冲的结构，如代码 2 所示。写缓冲是个一维数组，其元素个数为常量 DC_WR_BUFF_SIZE。它的每个元素记录了标识（tag）和 trdy 两个信息，其含义与 Cache 块中对应的信息相同。

代码 2　写缓冲的结构定义

```
struct writeBuffer{
        int tag;
        int trdy;
} dcWrBuff[DC_WR_BUFF_SIZE];
```

2. 数据 Cache 的访问过程

访问数据 Cache 时，首先要将地址划分为三个字段，标识、索引和块内偏移。但由于这里没有必要描述 Cache 的数据存储部分，因此块内偏移字段也不需要了。代码 3 描述了如何获得这两个字段的值。

代码 3　将存储地址划分为两个字段

```
1       blkOffsetBits = log2( DC_BLOCK_SIZE);
2       indexMask = (unsigned)(DC_NUM_SETS - 1);
3       tagMask = ~indexMask;


4       blk = ((unsigned)addr) >> blkOffsetBits;
5       index = (int)(blk & indexMask);
6       tag = (int)(blk & tagMask);
```

代码 3 的前三条语句定义了三个掩码：blkOffsetBits 为块内偏移掩码，log2 是对数函数（底数为 2）；indexMask 是索引掩码，它与 Cache 的组数有关；tag 是标识掩码。

代码 3 的第 5 条语句获得索引字段的值。第 6 条语句获得标识。至于第 4 条语句，显然它是为后面的计算做准备的。

另一个需要注意的地方是，引入 Cache 后必须引入时间，**为什么？**为此，我们定义全局时间变量 time，如代码 4 所示。

代码 4　引入全局时间变量后的程序执行过程

```
int time;
… …
void Execution(void)
{
    pc = 0x1000;              // 将程序的入口地址设为 0x1000
    time = 1;                 // 全局时间初值为 1
    for ( ; pc; ) {
        ….                    // 与之前 Project 中 Execution 函数对应部分相同
        time ++;              // 单周期 CPU，每执行 1 条指令需要 1 个 Cycle
    }
```

```
}
```

2.1 判断访问 Cache 是否命中

代码 4 描述了一个顺序的数据 Cache 访问过程。之所以说它是一个顺序的过程，是因为它每次仅将访问地址的标识字段 tag 与一个候选 Cache 块的标识进行比较，一共需要比较 DC_SET_SIZE 次。

代码 5　Cache 访问命中还是不命中

```
for( i = 0; i < DC_SET_SIZE; i++) {
    if ( (dCache[index][i].tag == tag) && (dCache[index][i].status != DC_INVALID) ) {
        *slot = i;
        return( TRUE);
    } else   /* Find a possible replacement line */
    if ( dCache[index][i].trdy < lruTime) {
        lruTime = dCache[index][i].trdy;
        lruSlot = i;
    }
}
```

代码 5 中的第一条 if 语句说明了访问 Cache 命中的条件，即 tag 匹配且 Cache 块不是无效（invalid）的。Else 部分的语句说明，如果不命中，还应该找出一个候选块，为接下来可能要进行的替换做准备。确定候选块的方法很简单，将该组中 trdy 最小的 Cache 块作为候选即可，表示这一个 Cache 块的上一次使用时间据当前时间最久。

2.2 命中的处理

如果是读访问，且访问 Cache 命中，只需要更新 Cache 块的 trdy 项，将其置为当前时间即可。如果是写访问，且访问 Cache 命中，除了更新 trdy，还需要将 Cache 块的状态（status）设为 2，表示“dirty”状态。对应的 C 代码如下，假设其中的 dcBlock 表示被访问的 Cache 块，time 表示当前时间：

```
// 读命中
dcBlock -> trdy = time;
// 写命中
```

```
dcBlock -> trdy = time;
dcBlock -> status = 2;
```

2.3 不命中的处理

与命中时相比，访问 Cache 不命中时的处理要复杂很多。我们仍然按照读和写两种情况来分析。

当读访问不命中时，应将对应的数据块从主存中调入 Cache，并更新对应的标识和状态记录。如果发生了替换，还需要将被换出的块的信息放入写缓冲。由于 Cache 中的数据存储部分已被忽略（即数据一直保存在主存中），这里仅需更新标识和状态记录，用 C 语言描述如代码 6 所示：

代码 6　读不命中时的处理

```
1    int trdy = MemRdLatency;
2    if ( dcBlock -> status == 2 )  // 如果被换出的块为脏块
3        trdy += wrBack( tag, time );
4    dcBlock -> tag = tag;
5    dcBlock -> trdy = time + trdy;
6    dcBlock -> status = 1;
```

代码 5 中 MemRdLatency 是个常数，表示从主存中读出一个 Cache 块所需的时间，即访问 Cache 不命中时的失效开销。它首先判断块 dcBlock 是否为脏块。如果是，则将其放入写缓冲，函数 wrBack 描述了这一过程，2.4 节将介绍该函数的具体实现；否则什么也不做。接下来，修改块 dcBlock 的状态。代码 5 的倒数第 2 句将 dcBlock->trdy 置为 time+trdy，表示当前时刻（time）之后的 trdy 时刻 Cache 块的数据才有效。如果发生了替换，数据有效的时刻还要往后推，因为在完成读之前，必须先把被换出的块写回主存。从这里也可以看出，代码 5 描述了一个阻塞 Cache，只有等写缓冲清空，读操作才能继续。

代码 7 描述了写不命中时的处理。它首先将变量 trdy 的值初始化为 0，然后判断被换出的 Cache 块是否为“脏”。若是，则将该块写回（wrBack）；否则，将 dcBlock->status 置为 2。请注意，代码 7 的第 2 行和第 5 行中都使用的变量 dcBlock->status，但第 2 行中的表示被换出 Cache 的状态，第 4 行中的表示被写访问（修改）的 Cache 块的状态。代码 7 的最后 3 行用于更新 Cache 块的状态。

代码 7　写不命中时的处理

```
int trdy = 0;
if ( dcBlock->status == 2) /* Must remote write-back old data */
    trdy = wrBack( tag, time);
else
    dcBlock->status = 2;
/* Read in cache line we wish to update */
trdy += MemRdLatency;
dcBlock->tag = tag;
dcBlock->trdy = time + trdy;
```

2.4 正确性测试

假设访问数据 Cache 的函数接口为

int accessDCache(int opcode, int addr, int time)

这里 opcode 表示访存指令的操作码，addr 表示访存地址，time 表示当前时刻。则可以通过下面的代码 8 测试所编写的数据 Cache 代码的正确性。

代码 8　数据 Cache 的正确性测试

```
1    int opcode = 41;    // 只进行LW 操作，LW 指令的操作码为 41
2    int time = 1;       // 起始时刻为 1
3    time += accessDCache( 41, 0x1000, time );
4    time += accessDCache( 41, 0x1004, time );
5    …
```

代码 9 的前两行表示进行正确性测试时，只执行 LW 指令，且起始时刻为 1。从第 3 行开始，每一行都表示一次数据 Cache 读访问（LW），并根据这次访问是否命中 Cache 修改当前时刻 time。根据这段代码结束后 time 的值，即可以判断所描述的 Cache 行为是否正确。

3. 作业

请在完成以下作业。每位同学独立完成。

1）按照第 2 节的介绍，完成数据 Cache 的 C 语言描述。假设访问数据 Cache 的接口函

数如 2.4 节所述。

2）测试作业 1）的正确性。

要求：提交源代码和必要的文档。