

Cache 设计实验报告

张作柏

17300240035

2019 年 6 月 9 日

目录

1	缓存机制简介	1
1.1	高速缓存原理	1
1.2	高速缓存结构	1
1.3	数据替换策略	2
2	实现细节	3
2.1	缓存组织结构实现	3
2.2	高速缓存读写	4
2.3	访存延时实现	5
2.4	替换策略实现	6
2.5	流水线结构调整	6
3	测试样例与结果	7
3.1	swlw.in	7
4	Cache 性能实验	8
5	注意事项	9
5.1	遇到的问题	9
6	申 A 理由	9
7	致谢	9

1 缓存机制简介

计算机的主存一般由 DRAM 芯片构成。在过去的 30 年中, DRAM 的速度仅以每年 7% 增长, 而处理器的性能则以每年 25%~50% 增长。在 1980 年前后, 处理器和存储器的速度是一样的, 但是性能从那时开始有了差别, 存储器速度开始严重落后。DRAM 访问时间比处理器周期长一到两个数量级。

为了抵消这种趋势, 计算机将最常用的指令和数据存储在更快但更小的存储器中, 这种存储器称为高速缓存 (cache), 其速度与处理器相近, 但是容量较小。如果处理器需要的数据在高速缓存中可用, 那么它可以快速返回, 这成为缓存命中 (hit), 否则, 处理器就需要从主存 (DRAM) 中获得数据, 这称为缓存缺失 (miss)。如果在大部分情况下缓存命中, 那么处理器就基本上不需要等待低速的主存。

1.1 高速缓存原理

高速缓存主要应用了时间局部性和空间局部性的原理。时间局部性意味着最近使用过的数据可能很快就会被再次使用。而空间局部性意味着当使用一个数据时, 很可能在之后使用与其相邻的数据。

高速缓存可以一次从主存中取许多相邻的数据, 以此利用空间局部性。同时, 还可以根据之前的数据使用情况, 选择合适的预取策略。而 cache 对时间局部性的利用主要体现在保存之前使用的数据和数据替换策略的设计。我们将在接下来的两小节分别进行讨论。

1.2 高速缓存结构

高速缓存通常保存存储器数据, 其存放数据字的数量称为容量 (C)。

为了利用空间局部性, 当处理器访问一块数据时, 我们可以从内存中提取多个相邻的字。这样的一组字称为高速缓存块 (cache block), 一个高速缓存块中的字数称为块大小 (b), 容量为 C 的高速缓存包含了 $B = C/b$ 个块。

一个高速缓存可以组织成 S 组, 其中每一组有一个或多个数据块。主存中数据的地址和高速缓存中数据的位置之间的关系称为映射 (mapping)。高速缓存按照组中块的数量进行分类:

- **直接映射:** 每组内只有一块, 所以组数 $S = B$ 。在这种情况下, 数据替换策略是无意义的。
- **多路组相联:** 每组提供 N 块, 此时需满足 $S \times N = B$ 。每个内存地址可以映射到唯一的组中, 但是它可以映射到一组中 N 块的任意一块。
- **全相联:** 只包含一组, 即 $S = 1$ 。这一组中包含了 B 路, 存储器地址可以映射到这些路中的任何一块。

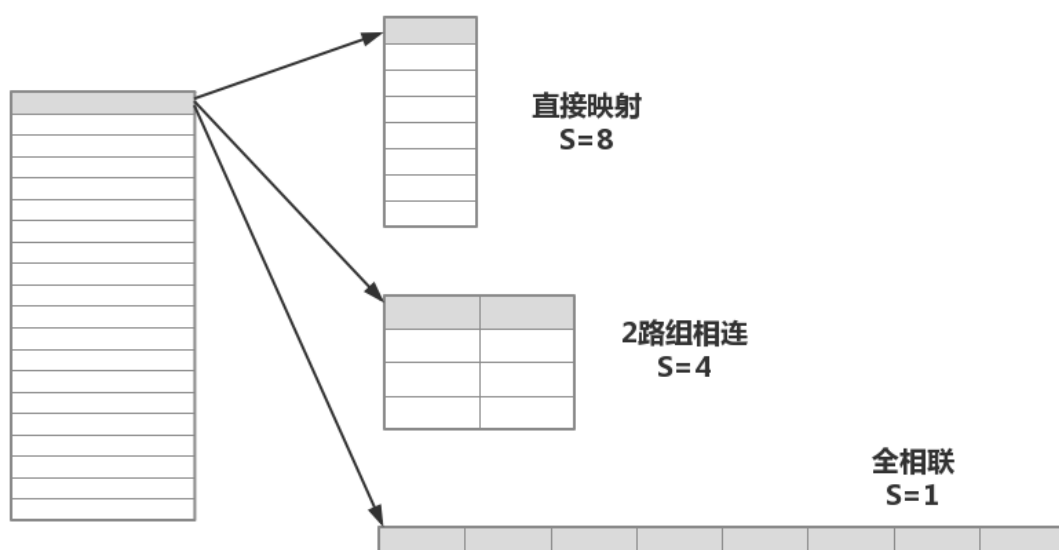


图 1: Cache 组织形式：直接映射、多路组相连和全相联

图 1简单地展示了三种组织形式，其中灰色的部分对应该内存地址可能映射到的块。注意到，以上三种组织形式是本质相同的，只是参数的设置不同而已，所以在书写代码时，可以将组数 S 和路数 N 设置为可调整的参数，以此方便的实现三种组织形式。

1.3 数据替换策略

在直接映射高速缓存中，每个地址映射到唯一的块和组上。如果装入新数据时，其所对应的组满了，那么组中的块就必须用新数据替换。在组相联和全相联的高速缓存中，高速缓存必须在组满时选择哪一个块被替换。常见的替换策略有：

- **随机替换 (RAND):** 随机选择一个块被替换。
- **先进先出 (FIFO):** 选取最先被读入 Cache 的块进行替换。
- **最不常使用 (LFU):** 该算法将一段时间内被访问次数最少的那个块替换出去。
- **最近最少使用 (LRU):** 把 CPU 近期最少使用的块替换出去。

2 实现细节

本次实验是在上次的流水线处理器上直接进行的改进，所以这里主要讨论 cache 模块的实现，对流水线的影响将在第 2.5 节进行讨论。

2.1 缓存组织结构实现

首先，我们以 parameter 的形式设置 cache 的结构：块大小 (b)、组数 (S) 和路数 (N)。

```
module cache #(parameter BLOCK_SIZE=64, SET_CNT=2, LINE_CNT=2) (clk, ...);
```

在之后的实现中，我们将结合这些参数与 for 循环来模拟任意结构的 cache。

紧接着，我们分别计算出确定偏移量 (offset) 与组号所需要位数：

```
parameter OFFSET = $clog2(BLOCK_SIZE/8);
parameter SET_LEN = $clog2(SET_CNT);
```

这里我们使用了 Verilog 中自带的 clog2() 函数，其作用是求以 2 为底的对数。注意，这里因为在我的实现中，内存的每个地址对应 8 个 bit，而这里的 BLOCK_SIZE 代表的是一个块所占的 bit 数，所以要先除以 8。

之后，我们通过 Verilog 中的多维数组来模拟 cache 的结构：

```
reg [BLOCK_SIZE-1:0] data[SET_CNT-1:0][LINE_CNT-1:0];
reg [LINE_CNT-1:0] valid[SET_CNT-1:0];
reg dirty[SET_CNT-1:0][LINE_CNT-1:0];
reg [8 - SET_LEN - OFFSET:0] tag[SET_CNT-1:0][LINE_CNT-1:0];
```

如此，我们将 Cache 的基本结构构建完成，其示意图如下：

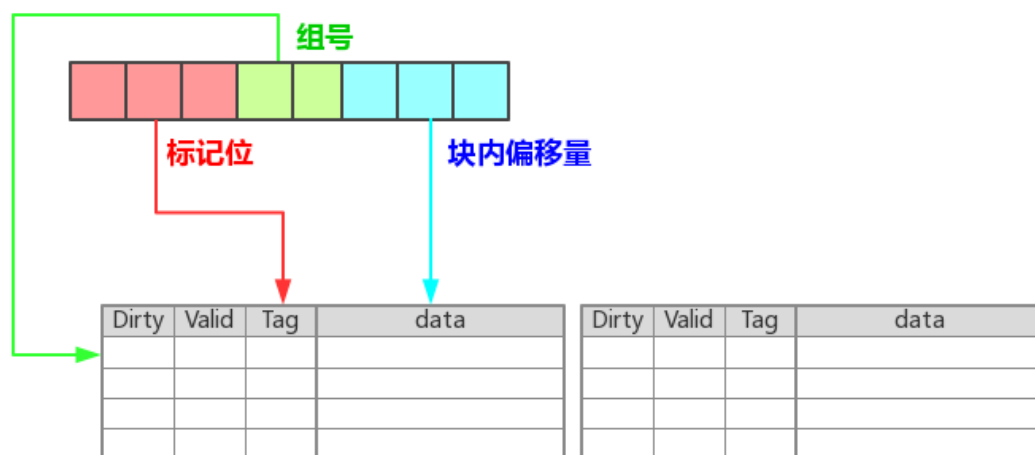


图 2: Cache 基本结构示例：块大小 $b = 64$ 、组数 $S = 4$ 、路数 $N = 2$

2.2 高速缓存读写

(1) 如何判断 Cache 中是否存在我们需要的数据？

首先，我们先根据访存的地址确定其映射到的组号：

```
assign SET_ID = addr[OFFSET + SET_LEN - 1 : OFFSET];
```

之后，我们利用 for 循环，判断组内是否存在有效行的标记位与访存地址相同，若存在，则说明找到了命中的位置。

```
always @(*)
    for (i = 0; i < LINE_CNT; i = i + 1)
        hiti[i] = valid[SET_ID][i] & (tag[SET_ID][i] == addr[8:OFFSET +
SET_LEN]);
assign hit = | hiti;
```

(2) 如何控制 Cache 的读写？

当 Cache 命中时，我们可以根据刚刚计算出的 hiti 数组，对命中的组进行操作：

- 读：

```
for (i = 0; i < LINE_CNT; i = i + 1)
    if (hiti[i]) rdata = data[SET_ID][i][addr[OFFSET-1:2]*32 +: 32];
```

- 写：

```
for (i = 0; i < LINE_CNT; i = i + 1)
    if (hiti[i])
        begin
            dirty[SET_ID][i] = 1;
            data[SET_ID][i][addr[OFFSET-1:2]*32 +: 32] = wdata;
        end
```

(3) 如何与 Memory 交互？

当 Cache 未命中时，我们需要从内存中取数据。这里我们直接套用了原来的 Dmem，通过改变写信号与访存地址来获取我们想要的地址。需要注意，我们这里每次访存取出的块大小不再是 32，而是给定的参数 BLOCK_SIZE。

```
Dmem #(BLOCK_SIZE) dmem(clk, reset, ADDR, WEN, WDATA, RDATA);
```

当发生 miss 时，我们通过改变地址，来从内存中获取数据：

```
ADDR = {addr[8:OFFSET], 'OFFSET'b0};
WEN = 0;
WDATA = 0;
```

注意，这里我们的 Dmem 是组合逻辑，所以不存在延时。而访存的延时，我们会通过状态机来设置，这将在下一小节详细讨论。

2.3 访存延时实现

为了模拟访存时的时间延时，在每次访存时我们需要让流水线等待几个周期，且根据 hit 与 miss 的情况等待不同的周期数。这一点我们可以通过状态机来实现。

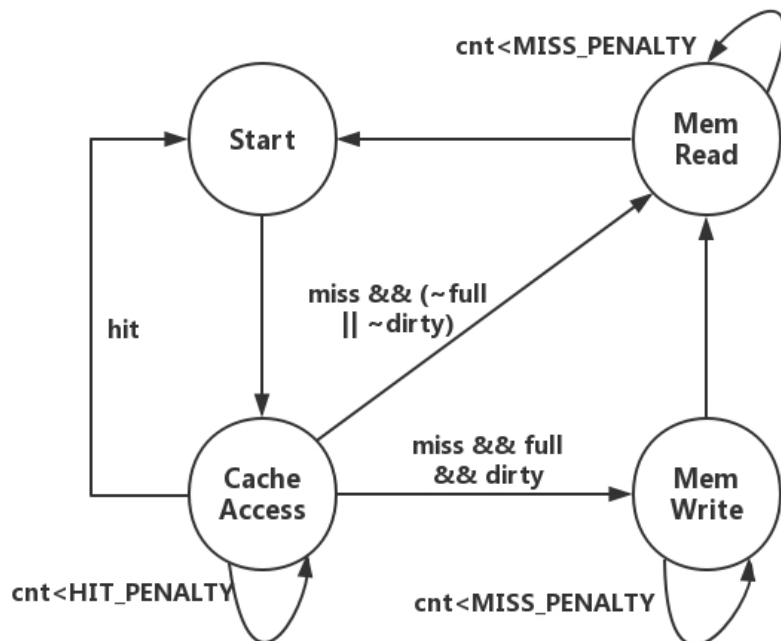


图 3: Cache 延时状态机

首先，我们需要设置两个延时参数：HIT_PENALTY 和 MISS_PENALTY，分别表示从 cache 中取数据和从 memory 中取数据需要的周期数。

之后，设计四个状态，含义如下：

1. **Start:** 起始状态。若指令不需要访存，则始终停留在该状态；若需要访存时，从该状态起始，对计数器清零。
2. **CacheAccess:** 访问 Cache 的状态，自带一个计数器。当计数器累加到 HIT_PENALTY 时，根据是否 hit，选择转移到的状态。
 - (1) 若命中，则从 Cache 中将数据取出，并转移回 Start 状态，标记为取数据完成。
 - (2) 若未命中且当前组已满且需要写入，则下一步需要根据替换策略进行替换，跳至 MemWrite 状态，并将计数器清零。
 - (3) 若未命中且当前组仍有空位，则下一步从内存中取数据，跳至 MemRead 状态，并将计数器清零。
3. **MemWrite:** 写内存状态，当需要进行替换时，进入该状态。在状态起始，更改访存的地址和写数据，设置写使能为 1。当计数器累加到 MISS_PENALTY 时，将其清零，跳至 MemRead 状态。

4. **MemRead**: 读内存状态, 从内存中读取需要的数据。在状态起始, 将访存的地址设置为输入的地址信号。当计数器累加至 MISS_PENALTY 时, 将其清零, 输出读出的数据, 并标记为取数据完成, 跳至 Start 状态。

2.4 替换策略实现

随机替换 (RAND)

先进先出 (FIFO)

最不常使用 (LFU)

最近最少使用 (LRU)

2.5 流水线结构调整

整个流水线的变动并不大, 主要调整在暂停信号。当数据未取完时, 需要将流水线完全暂停住。我们可以通过增加两个信号来控制: LW 信号控制是否为 lw 或 sw 指令、DATA_COMPLETE 信号控制是否完成取数据操作。

在冒险处理 Hazard 模块中, 若当前 Memory 阶段的指令为 lw 或 sw 指令, 且未完成取数据操作, 则令整个流水线暂停, 设置所有状态寄存器的 Stall 信号为 1 即可。

3 测试样例与结果

与单周期相同的测试样例均已通过，这里只列举几个关键的和新加入的测试样例。

3.1 swlw.in

```

0x0 : addi $s0, $0, 0      | 20100000
0x4 : addi $t0, $0, 0      | 20080000
0x8 : addi $v1, $0, 10     | 2003000a
0xc : for:                 |
0xc : sw $t0, 0($s0)       | ae080000
0x10 : addi $t0, $t0, 1    | 21080001
0x14 : addi $s0, $s0, 4    | 22100004
0x18 : bne $t0, $v1, for   | 1468fffc
0x1c : addi $s0, $0, 0     | 20100000
0x20 : addi $t0, $0, 0     | 20080000
0x24 : addi $t1, $0, 0     | 20090000
0x28 : addi $s1, $0, 0     | 20110000
0x2c : for2:               |
0x2c : lw $t0, 0($s0)      | 8e080000
0x30 : add $s1, $s1, $t0   | 02288820
0x34 : addi $s0, $s0, 4    | 22100004
0x38 : addi $t1, $t1, 1    | 21290001
0x3c : slt $v0, $v1, $t1   | 0069102a
0x40 : beq $v0, $0, for2   | 1002fffa
0x44 : nop                 | 00000000
0x48 : nop                 |

```


4 Cache 性能实验

5 注意事项

5.1 遇到的问题

1. 对 Dmem 的初始化如果放在 initial 中好像会有些问题，我把它调整为 reset 为 1 时进行初始化就好了。
2. 仿真的结果和在板子上的结果可能会差异很大，在同一个时钟沿，仿真时两个信号会同时变化，而在板子上就不是了。可以通过将信号分别调整为上升沿和下降沿触发来解决该问题。

6 申 A 理由

- 实现了参数化的高速缓存，复用性强
- 设计检验 Cache 正确性与性能的汇编代买
- 进行了大量实验，对比不同大小与替换策略 Cache 的性能

7 致谢

感谢罗翔同学与我进行深入的讨论，对实验的设计提出了宝贵的建议！