Implementation of a LRU/ FIFO Cache

Computer Architecture: Lab #2

Due on May 17, 2020 at 23:59pm

魏剑宇
PB17111586

May 17, 2020

Contents

1 实验环境和工具

- **Host OS**: macOS Mojave 10.14.6
- **Guest OS**: Ubuntu Bionic Beaver
- **VSCode**: 1.44.2
- **Vivado**: HL WebPack Edition
- **Git**: 2.20.1 (Apple Git-117)

本次实验 Simulation 部分在 Ubuntu 虚拟机中完成 (Vivado 没有提供 macOS 版本)，代码编写在 mac 上使用 VSCode 完成.

我的做法是在虚拟机 vivado 创建项目时，将源码的文件夹添加进去的时候，不拷贝，而是映射到项目中。然后在 mac 上通过共享文件夹将源码映射到 ubuntu 上，这样，源码编辑和 Git 版本控制都可以在 mac 上完成，只需在仿真的时候切换到 Ubuntu。整个一套 workflow 还是非常流畅舒适的。

2 Cache 设计

cache 部分的主要更改是替换策略的更改。其他的除了需要给一些变量添加一个 WAY 的维度外，还有就是需要确定是否有命中明确定命中的是哪一路. 这一点我通过一个 for 语句要实现：

```
always @ (*) begin
    cache_hit = 1'b0;
    way_select = 0;
    for (integer i = 0; i < WAY_CNT; i++) begin
        if (valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr) begin
            cache_hit = 1'b1;
            way_select = i;
        end
    end
end
```

2.1 LRU & FIFO

我这里重点描述一下我认为自己设计得非常不错的地方.

像 LRU 或者 FIFO，一种实现思路是用一个位数比较大的寄存器，存储上一次访问到当前的时间. FIFO 的实现还可以用一个队列来存储每次访问.

不过像这样的实现方式事实上有一些问题。姑且不谈上一次的时间每个周期都需要更新，并且可能溢出了位数上限，还有问题就是，每次选出到当前时间最大的时候，需要用到很多的加法器和比较器，会使用很多的资源，实现起来也比较复杂。

而我则使用了一种使用寄存器资源很小，电路资源也很少的实现方式————用一个矩阵来存储历史：

```
reg [WAY_CNT-1:0] history_matrix [SET_SIZE][WAY_CNT];
```

对于每个 cache 组，`history_matrix` 是一个 `WAY_CNT x WAY_CNT bit` 大小的矩阵.（这个矩阵非常小，在 `WAY_CNT = 4` 的情况下，只有使用历史时间存储的 1/8 大小）. 这个矩阵中，$history[i][j] = 1$ 表示的意思是，第 i 路 cache 比第 j 路 cache 更早使用.

这样我们就可以发现，判断某个 cache 是否被替换就非常容易了：

```
&history_matrix[set_addr][i] == 1
```

也就是说，这一路 i 对应的行全为 1，通过一个位与操作就可以实现了. 相对于一大堆比较器的版本，可以说是很简洁并且高效了.

更新也比较简单，只需要在每次访问的时候，将该行全设为 0（除了对角线），该列全设为 1.

```
for (integer i = 0; i < WAY_CNT; i++)
    history_matrix[set_addr][i][way_select] <= 1'b1;
// set row to 0
for (integer j = 0; j < WAY_CNT; j++)
    if (j != way_select)
        history_matrix[set_addr][way_select][j] <= 1'b0;
end
```

这种方式下，LRU 和 FIFO 唯一的区别是 FIFO 只在调入缓存的时候需要更新，LRU 在访问和调入的时候都需要更新.

2.2 接入 CPU

为了接入 CPU，主要要做的有亮点：产生两个新的控制信号 `rd_req` 和 `wr_req`. 这两个信号就是在 cache 中用到的两个信号的意思。其实现也很简单：

```
assign rd_req = opcode == `OP_LOAD;
assign wr_req = opcode == `OP_STORE;
```

剩下这需要把它们放到流水段寄存器中传过来即可。

主要是需要更改 Hazard 模块。在 Hazard 模块中，需要在有请求且 miss 的时候，产生流水线气泡.

```
always @(*) begin
    if ((rd_req | wr_req) & miss) begin
        mem_bubbleF = 1;
        mem_bubbleD = 1;
        mem_bubbleE = 1;
        mem_bubbleM = 1;
        mem_flushW = 1;
    end else begin
        mem_bubbleF = 0;
        mem_bubbleD = 0;
        mem_bubbleE = 0;
        mem_bubbleM = 0;
        mem_flushW = 0;
    end
end
```

这里气泡需要插入一直到 MEM 的信号，所以通过 bubble MEM 之前的所有段，并 flush WB 来实现. 之后将这些信号或上原来的信号即可.

3 资源占用 & 性能

截图较多，关于资源利用的截图可以在 report 文件夹中查看.

更大的 cache 在综合的时候不知道为啥综合到一半就会失败，然后 vivado 会闪退，所以没法进行试验了.

以下是统计的资源利用的情况，我的 MATRIX 和 SORT 两个程序都使用默认的大小.

	Cache Size (WORD)	LUT	FF	MATRIX (ns, hitrate)	SORT (ns, hitrate)
LINE_ADDR_LEN = 2 SET_ADDR_LEN = 3 WAY_CNT = 4	128	2360	5042	1335980 45.4%	113260 86.9%
LINE_ADDR_LEN = 3 SET_ADDR_LEN = 3 WAY_CNT = 2	128	2006	5189	1320748 45.9%	128308 89.1%
LINE_ADDR_LEN = 3 SET_ADDR_LEN = 3 WAY_CNT = 3	192	3018	7332	1317292 46.4%	127244 89.2%
LINE_ADDR_LEN = 3 SET_ADDR_LEN = 3 WAY_CNT = 4	256	4355	9494	645380 82.2%	87544 96.4%
LINE_ADDR_LEN = 4 SET_ADDR_LEN = 3 WAY_CNT = 2	256	4035	10044	499628 90.0%	77032 98.2%
LINE_ADDR_LEN = 4 SET_ADDR_LEN = 3 WAY_CNT = 3	384	6355	14225	486456 90.7%	70336 99.3%
LINE_ADDR_LEN = 4 SET_ADDR_LEN = 3 WAY_CNT = 4	512	9578	18430	269669 99.4%	70336 99.3%
LINE_ADDR_LEN = 3 SET_ADDR_LEN = 4 WAY_CNT = 2	256	3918	9400	561572 86.6%	82616 97.2%
LINE_ADDR_LEN = 3 SET_ADDR_LEN = 4 WAY_CNT = 3	384	6118	13676	502028 89.8%	79512 98.6%
LINE_ADDR_LEN = 3 SET_ADDR_LEN = 4 WAY_CNT = 4	512	7410	17973	288416 98.6%	79512 98.6%

Table 1: 资源占用和性能表

4 分析

4.1 替换策略

我自己选取了一些情况对替换策略进行实验，LRU 和 FIFO 差别不大，LRU 小优。但正如我在 2.1 中所讲的，对于我的实现方式而言，两者所占用的资源是几乎完全一样的，我没必要针对 LRU 和 FIFO 进行资源上的权衡，所以我直接选取 LRU 作为我的替换策略。

4.2 最佳参数和分析

分析一下表格 1，我们可以发现：

- 对于矩阵程序，我们可以发现，当 LINE_ADDR_LEN 或 WAY_CNT 较小时，其命中率和效率明显低于其它情况，同时，每次 WAY_CNT 从 3 到 4，都会带来性能的爆发.

  分析其原因，应当是矩阵乘法具有较强的空间局部性，因此，若 LINE_ADDR_LEN 较大，每次将内存中的数据替换出来时，都能替换出来很大一批数据，并且这些数据几乎都会在紧接下来的计算中被用到.

  同时，由于矩阵乘法设计到同列的连续读取，这些同列的数据虽然不再同一个 LINE 中，但它们的 SET_ADDR 相同，因此会进入同一个组中. 当 WAY_CNT 较大时，它们能够有效地同时存在于缓存中，而不至于被换来换去.

- 对于排序程序而言，其并不具有很强的局部性，WAY_CNT 增大而带来的性能提高并不明显，读取相对来说更加随机（但也具有一定的局部性），因此其命中率、运行时间随大小的关系比较平滑.

由如上分析，我们得出结论：

- 对于矩阵程序，其 cache 大小增大（具体来说是 LINE 和 WAY 的增大，能大幅度的提高性能，是比较值得的. 权衡资源占用后，我们可以选择 (LINE_ADDR_LEN = 3, SET_ADDR_LEN = 4, WAY_CNT = 4) 的参数，此时命中率比较高，时间比较少（与最佳的一个相差不大），同时所使用的 LUT 少了很多.
- 对于排序程序，其 cache 大小和 WAY 增大并不具有非常强的作用，推荐使用 (LINE_ADDR_LEN = 4, SET_ADDR_LEN = 3, WAY_CNT = 2) 的情况，此时资源占用相对较少，性能也非常不错。