Lab 3 Report

实验内容

- 将直接映射DCache改写为组相连DCache，使用FIFO和LRU两种替换算法进行改写
- 将改写好的DCache接入流水线
- 调整DCache各项参数，统计不同配置的DCache的性能

DCache设计

组相连DCache设计

组相连DCache的基本元件和直接相连无异，但需要修改：

- 命中逻辑：需要对每一路Tag进行并行比较：

```
wire cache_hit;
reg [WAY_CNT-1:0] hit;
always @(*) begin                       // 判断 输入的address 是否在 cache 中命中
    hit = 0;
    for(integer i = 0; i < WAY_CNT; i++) begin
        if(valid[i][set_addr] && cache_tags[i][set_addr] == tag_addr) begin
            hit[i] = 1'b1;
        end
    end
end
assign cache_hit = |hit;                // 如果有任意一路命中，则cache_hit=1
```

- FIFO替换算法：使用移位寄存器实现一个队列来维护**换入路的编号**，在每次缺失时，取出最低位（队头）的路编号，将数据换入这一路后，将寄存器向右移位，并将路编号插入最高位（队尾）：

```
reg [WAY_CNT * WAY_WIDTH - 1 : 0] fifo [SET_SIZE];
assign select_out = fifo[set_addr][WAY_WIDTH - 1 -: WAY_WIDTH];
.... // 无关代码
// 在SWAP_IN_OK状态下
fifo[mem_rd_set_addr] <= {select_out, fifo[mem_rd_set_addr][WAY_CNT * WAY_WIDTH
- 1 : WAY_WIDTH]};
```

  值得注意的是，我们需要对这个队列赋予一个初值。当rst时：

```
for(integer i = 0; i < SET_SIZE; i++) begin
    for(integer j = 0; j < WAY_CNT; j++) begin
        fifo[i][j * WAY_WIDTH +: WAY_WIDTH] <= j[WAY_WIDTH-1:0];
    end
end
```

- LRU替换算法：类似于FIFO，使用移位寄存器为每一行维护所有二进制路号，之后进行如下操作：

- 当命中某一路时，在这个移位寄存器中找到对应路号的位置，将这个路号移动到最高位，同时高位路号依次右移（例如：4路组相连初始状态为11100100，即二进制的3210。当某次访存命中第一路，那么这个寄存器变为01111000）。
- 当没有命中时，取出最低的路号，进行替换，替换完成后将寄存器右移，并将最低位的路号写到最高位

实现方法如下：

```
reg [WAY_CNT * WAY_WIDTH - 1 : 0] lru [SET_SIZE];
assign select_out =  lru[set_addr][WAY_WIDTH - 1 -: WAY_WIDTH];

// update LRU
reg [WAY_CNT-1:0] lru_hit_loc;
always @(*) begin
    lru_hit_loc = 0;
    if(cache_stat == IDLE && cache_hit) begin
        for(integer i = 0; i < WAY_CNT; i++) begin
            lru_hit_loc[i] = (lru[set_addr][i * WAY_WIDTH +: WAY_WIDTH] ==
hit_way);
        end
    end
    else if(cache_stat == SWAP_IN_OK) begin
        for(integer i = 0; i < WAY_CNT; i++) begin
            lru_hit_loc[i] = (lru[mem_rd_set_addr][i * WAY_WIDTH +: WAY_WIDTH]
== select_out);
        end
    end
end
wire [WAY_WIDTH-1 : 0] lru_hit;
assign lru_hit = encoder[lru_hit_loc];
```

其中，为了适配可变长度的编码，我不得不维护了一个查表式encoder

在命中时，需要对LRU表中每一个编码进行并行比较，找到对应命中路编码所在位置，之后修改表项：

```
// 更新LRU：找到命中点，提到最高位，剩下的顺序补位
for(integer i = 0; i < (WAY_CNT - 1) * WAY_WIDTH; i = i + WAY_WIDTH) begin
    if(i >= lru_hit * WAY_WIDTH)begin
        lru[set_addr][i +: WAY_WIDTH] <= lru[set_addr][(i + WAY_WIDTH) +:
WAY_WIDTH];
    end
end
lru[set_addr][WAY_CNT * WAY_WIDTH - 1 -: WAY_WIDTH] <= hit_way;
```

在没有命中的填充结束时，也需要对LRU表进行更新：

```
lru[mem_rd_set_addr] <= {select_out, lru[mem_rd_set_addr][WAY_CNT * WAY_WIDTH -
1 : WAY_WIDTH]};
```

DCache与流水线的适配

DCache接入流水线后，需要使流水线能够在Cache Miss时能停下。由此，我们需要做如下修改：

- 将miss信号接入Hazard，并在miss时，停顿所有段间寄存器
- 由于ICache是同步读，因为同步读的那个寄存器并没有被停顿，如果贸然停顿流水线，就会出现丢失指令的情况。因此在ID阶段需要做特殊处理：

```
// IR.v
always@(posedge clk)
begin
    bubble_ff <= bubbleD;
    flush_ff <= flushD;
    if(!bubble_ff) inst_ID_old <= inst_ID_raw;
end
```

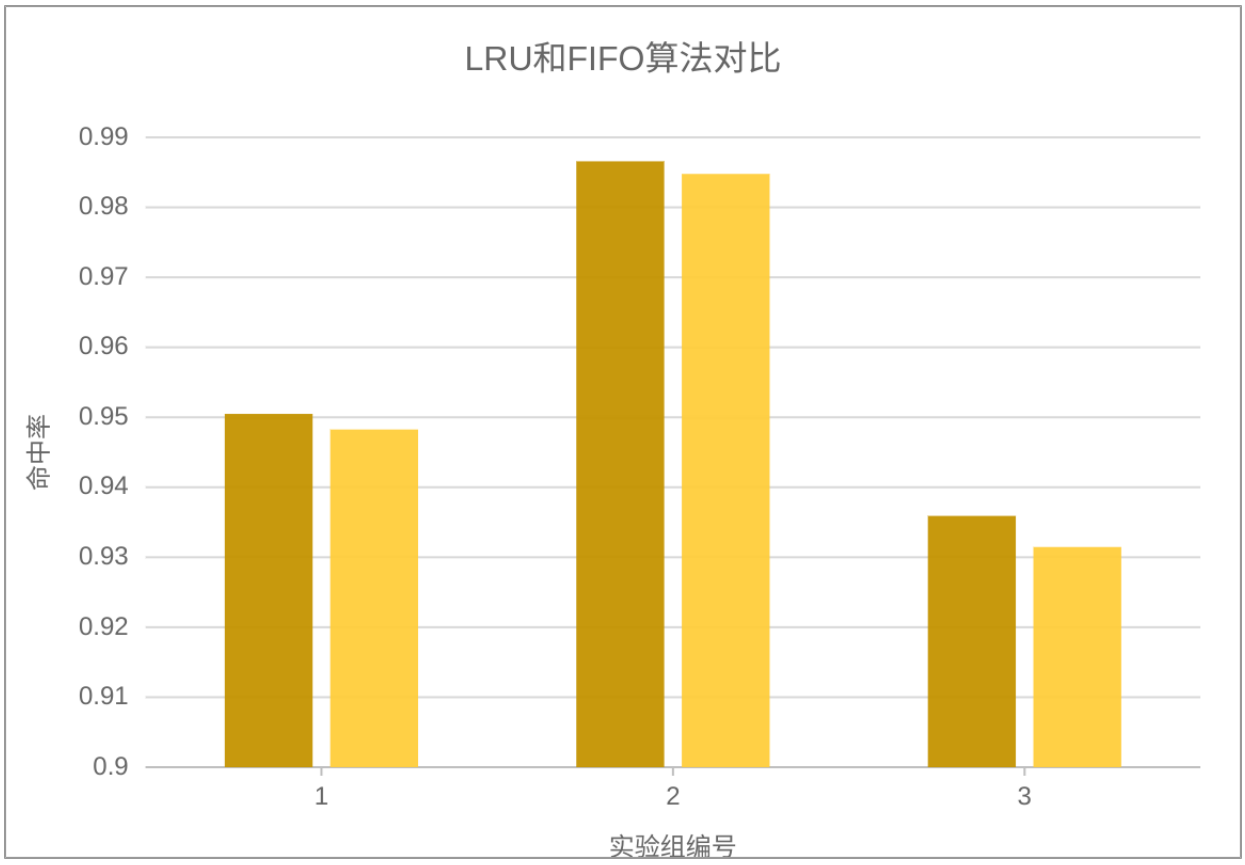
DCache性能测试

QuickSort

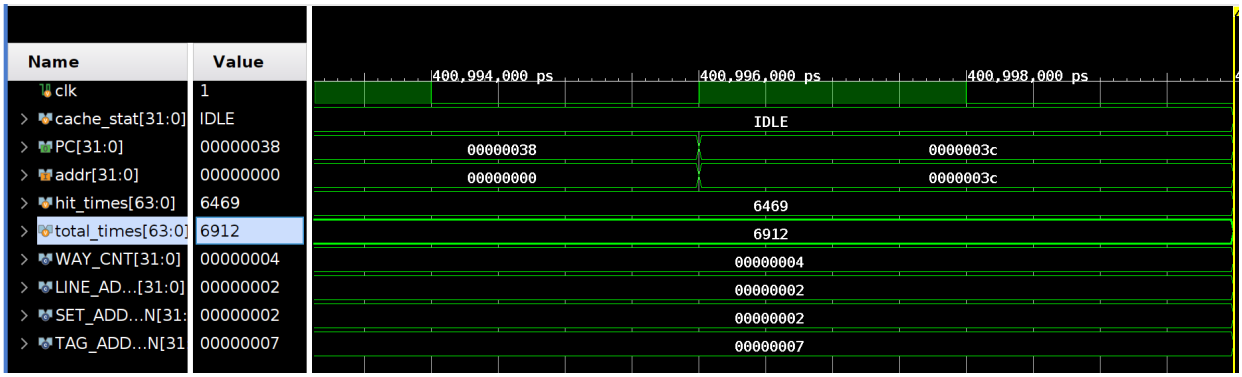
算法对命中率的影响

测试替换算法对于Cache性能的影响：（使用256个数字的快速排序）

No	替换算法	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	HIT_TIMES	TOTAL_TIMES	HIT_RATE
1	LRU	1	4	2	5	6469	6806	0.9505
1	FIFO	1	4	2	5	6469	6822	0.9483
2	LRU	2	4	2	5	6469	6557	0.9866
2	FIFO	2	4	2	5	6469	6569	0.9848
3	LRU	4	2	2	7	6469	6912	0.9359
3	FIFO	4	2	2	7	6469	6945	0.9315



可以基本得出结论：LRU算法略强于FIFO算法，以下是LRU算法在4路每路2组每组2字的配置下运行快速排序的性能截图：

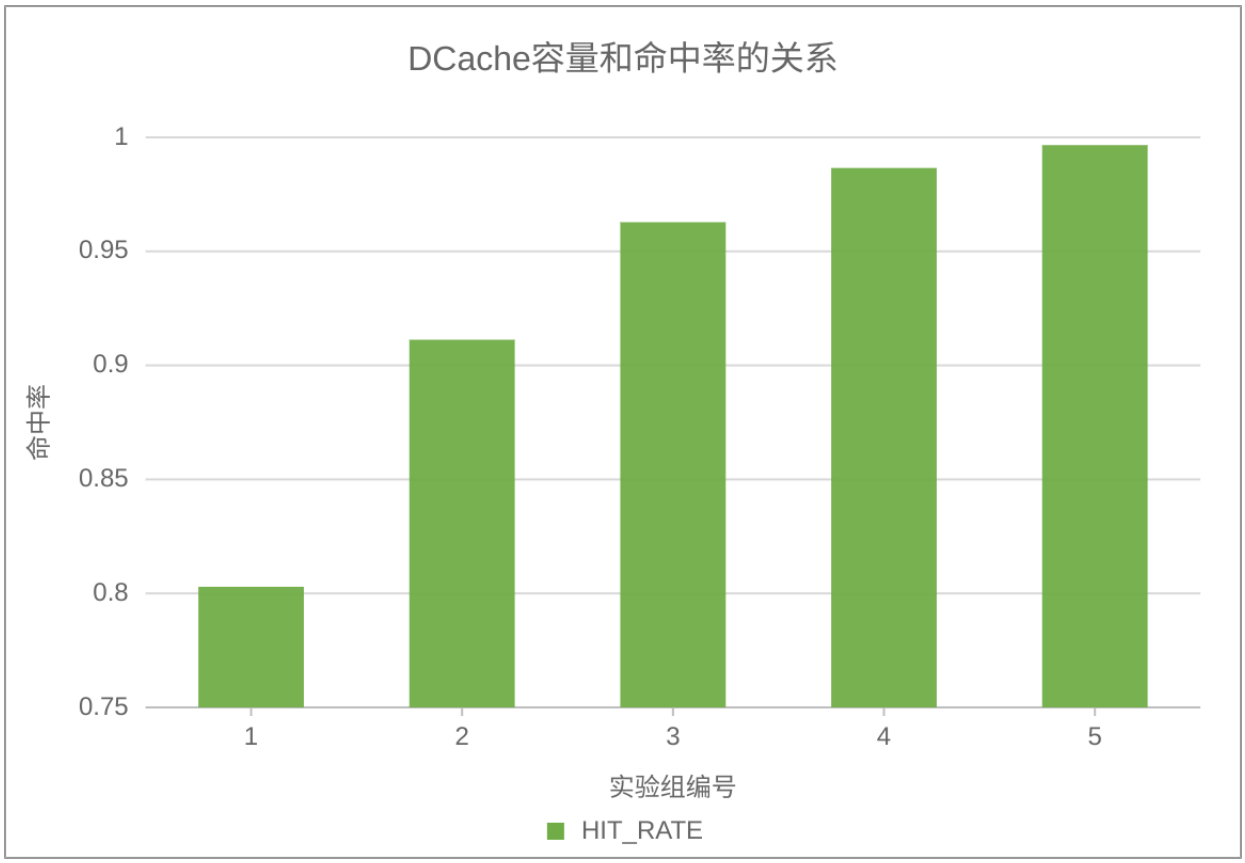


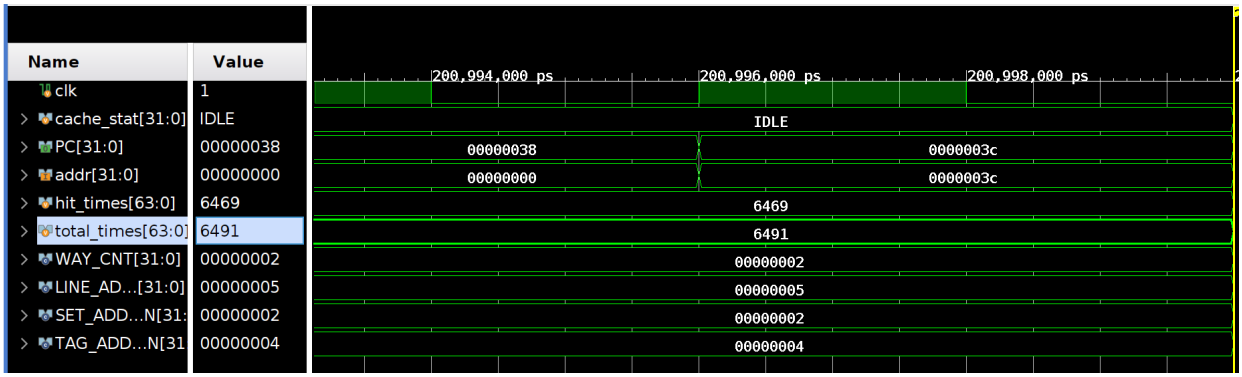
容量对命中率的影响

下面的实验中，我们都使用比较好的LRU算法来进行探索：

改变LINE_LEN来统计：

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	HIT_TIMES	TOTAL_TIMES	HIT_RATE
1	2	1	2	8	6469	8057	0.8029
2	2	2	2	7	6469	7099	0.9113
3	2	3	2	6	6469	6719	0.9628
4	2	4	2	5	6469	6557	0.9866
5	2	5	2	4	6469	6491	0.9966





由于我们设置了最大的访存延迟为50个周期，因此不能再扩大行容量了。可以看到，当行地址长度为5时，基本已经将所有mem中的数据全部取入高速缓存了，因此这个时候cache基本全是强制缺失和冲突缺失。

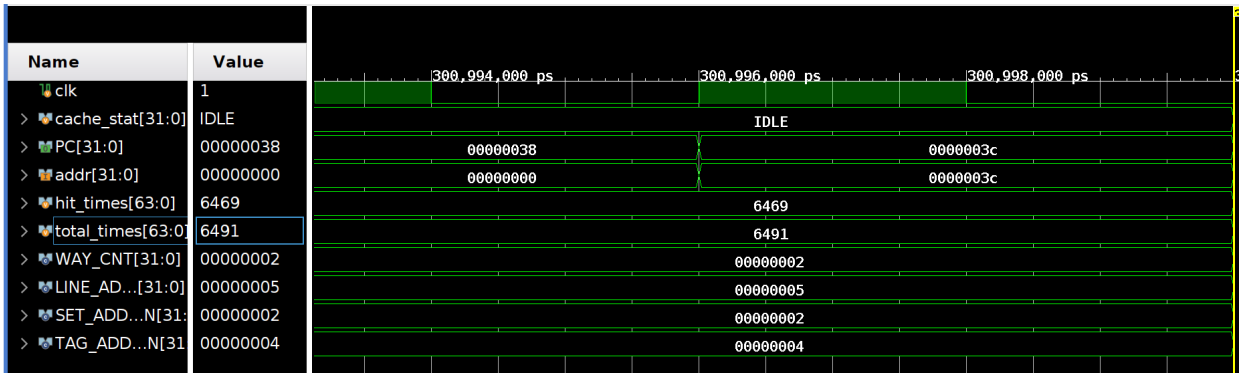
这时可以得出结论：当容量逐渐增大时，命中率逐渐提升，但提升速度逐渐变得缓慢。当容量大到可以囊括内存中大部分数的时候，这时候命中率是最高的。

组相连度和组数、行长度对命中率的影响

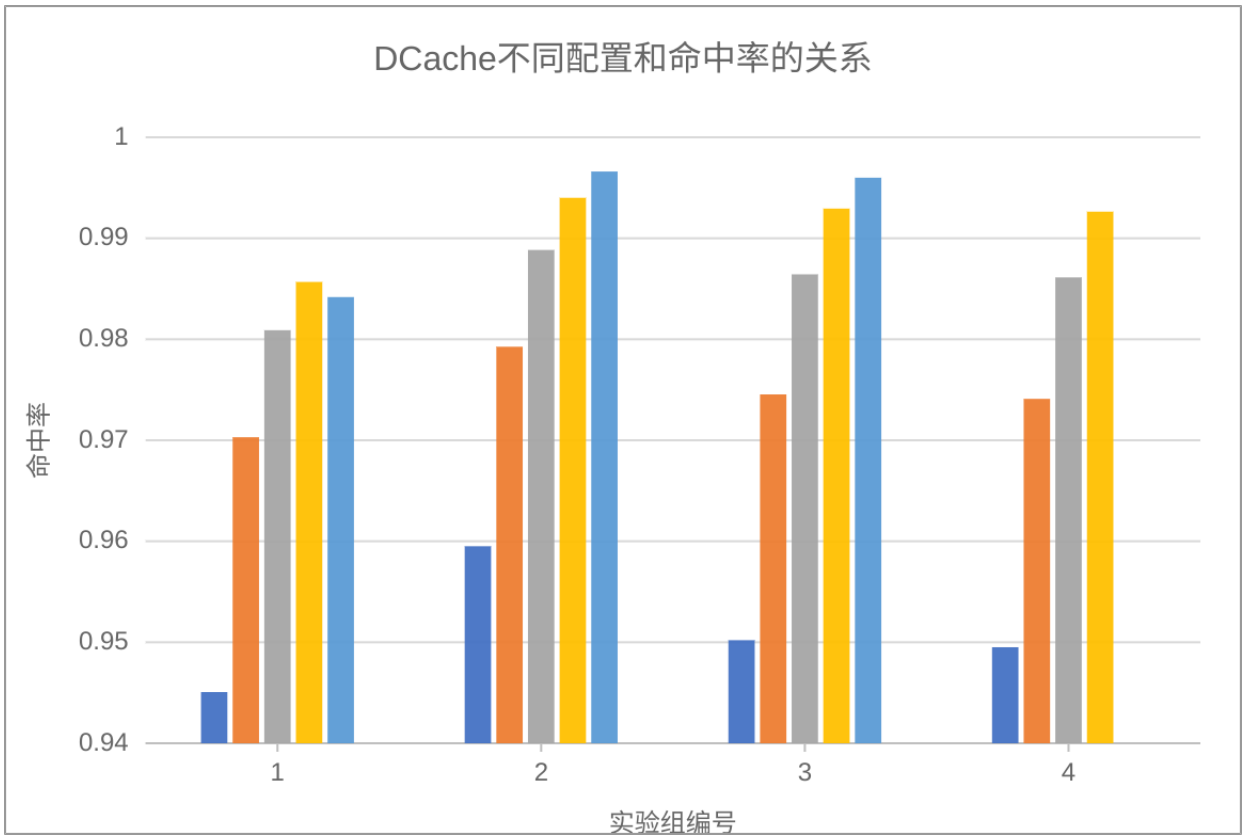
我们将Cache容量设置为上面的最优值，改变其余几个参数进行实验：

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	HIT_TIMES	TOTAL_TIMES	HIT_RATE
1	1	1	7	3	6469	6845	0.9451
1	1	2	6	3	6469	6667	0.9703
1	1	3	5	3	6469	6595	0.9809
1	1	4	4	3	6469	6563	0.9857
1	1	5	3	3	6469	6573	0.9842

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	HIT_TIMES	TOTAL_TIMES	HIT_RATE
2	2	1	6	4	6469	6742	0.9595
2	2	2	5	4	6469	6606	0.9793
2	2	3	4	4	6469	6542	0.9888
2	2	4	3	4	6469	6508	0.9940
2	2	5	2	4	6469	6491	0.9966
3	4	1	5	5	6469	6808	0.9502
3	4	2	4	5	6469	6638	0.9745
3	4	3	3	5	6469	6558	0.9864
3	4	4	2	5	6469	6515	0.9929
3	4	5	1	5	6469	6495	0.9960
4	8	1	4	6	6469	6813	0.9495
4	8	2	3	6	6469	6641	0.9741
4	8	3	2	6	6469	6560	0.9861
4	8	4	1	6	6469	6517	0.9926



可以看出，**2路+5位行地址+2位组地址**构成了QuickSort程序的最佳Cache配置。同时我们也可以发现：**当Cache容量一定时，行越宽，命中率就会越低。**不过行越宽代表着缺失处理所用的时间更长，但本实验中所有缺失填入时间均为50周期，因此只需要关注命中率即可。上图为最优配置下的运行结果截图。



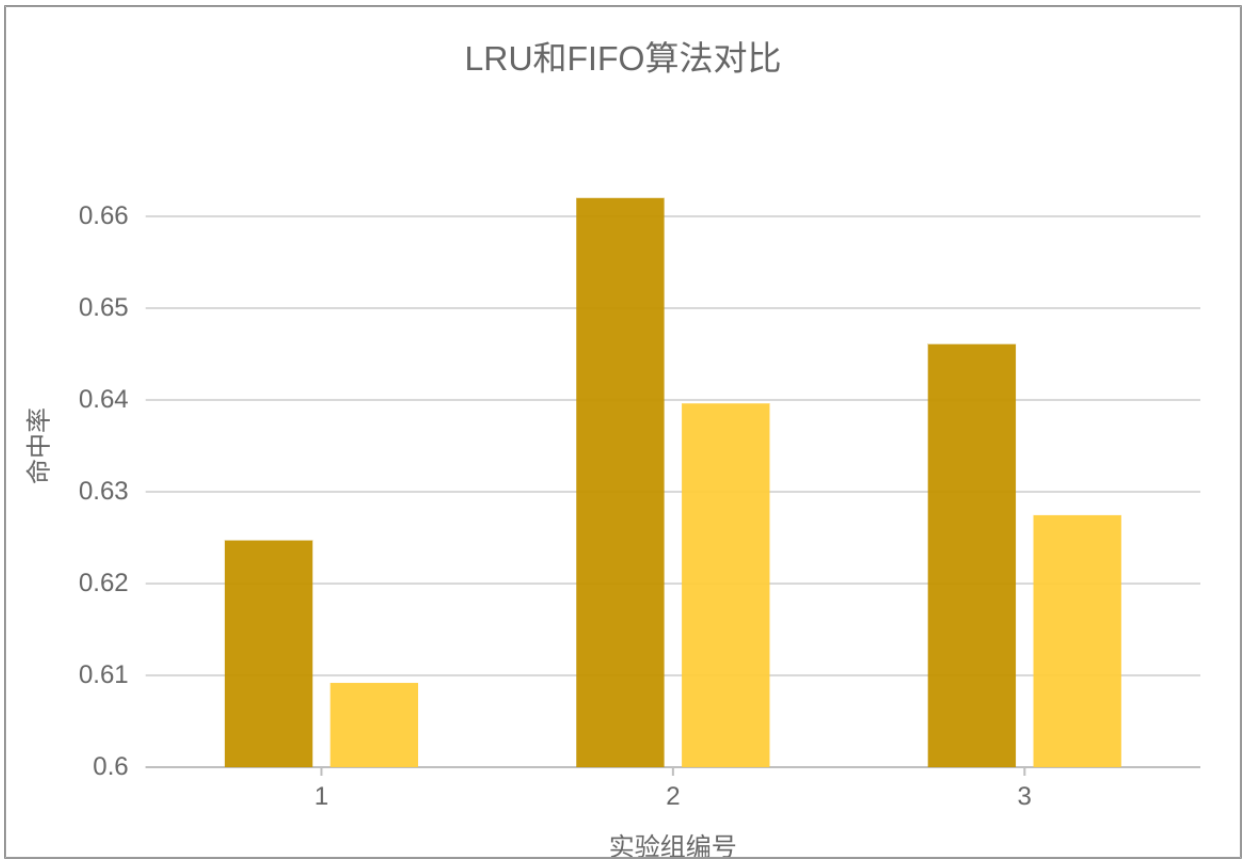
仔细分析可以发现，由于程序访问的地址空间比较小，组相连的优势并没有完全体现，因为想要触发冲突缺失，在单路的情况下，字地址至少相差256——这已经是快速排序的规模了。如果牺牲了单路大小换取更多的路，反而会导致更多的替换发生。

MatMul

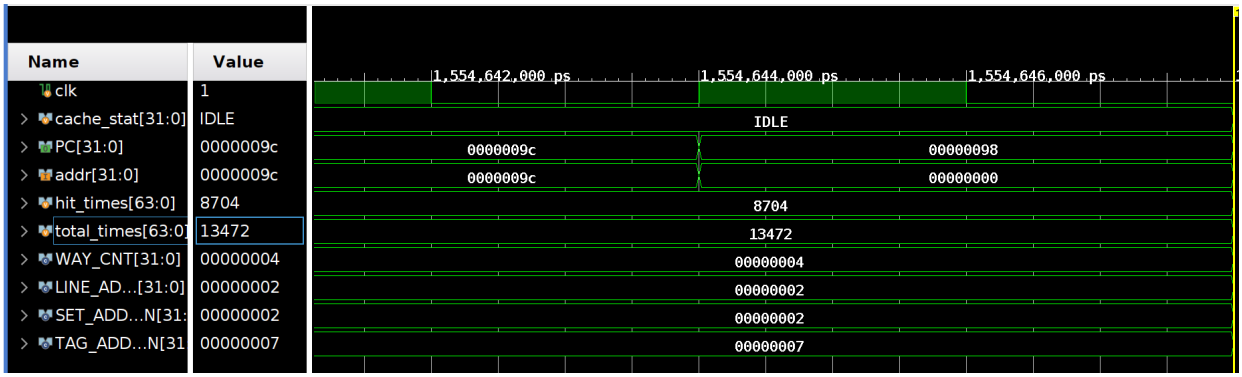
算法对命中率的影响

测试替换算法对于Cache性能的影响：（使用规模为16的矩阵乘法）

No	替换算法	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	HIT_TIMES	TOTAL_TIMES	HIT_RATE
1	LRU	1	4	2	5	8704	13933	0.6247
1	FIFO	1	4	2	5	8704	14288	0.6092
2	LRU	2	4	2	5	8704	13148	0.6620
2	FIFO	2	4	2	5	8704	13608	0.6396
3	LRU	4	2	2	7	8704	13472	0.6461
3	FIFO	4	2	2	7	8704	13872	0.6275



可以基本得出结论：LRU算法略强于FIFO算法，以下是LRU算法在4路每路2组每组2字的配置下运行快速排序的性能截图：

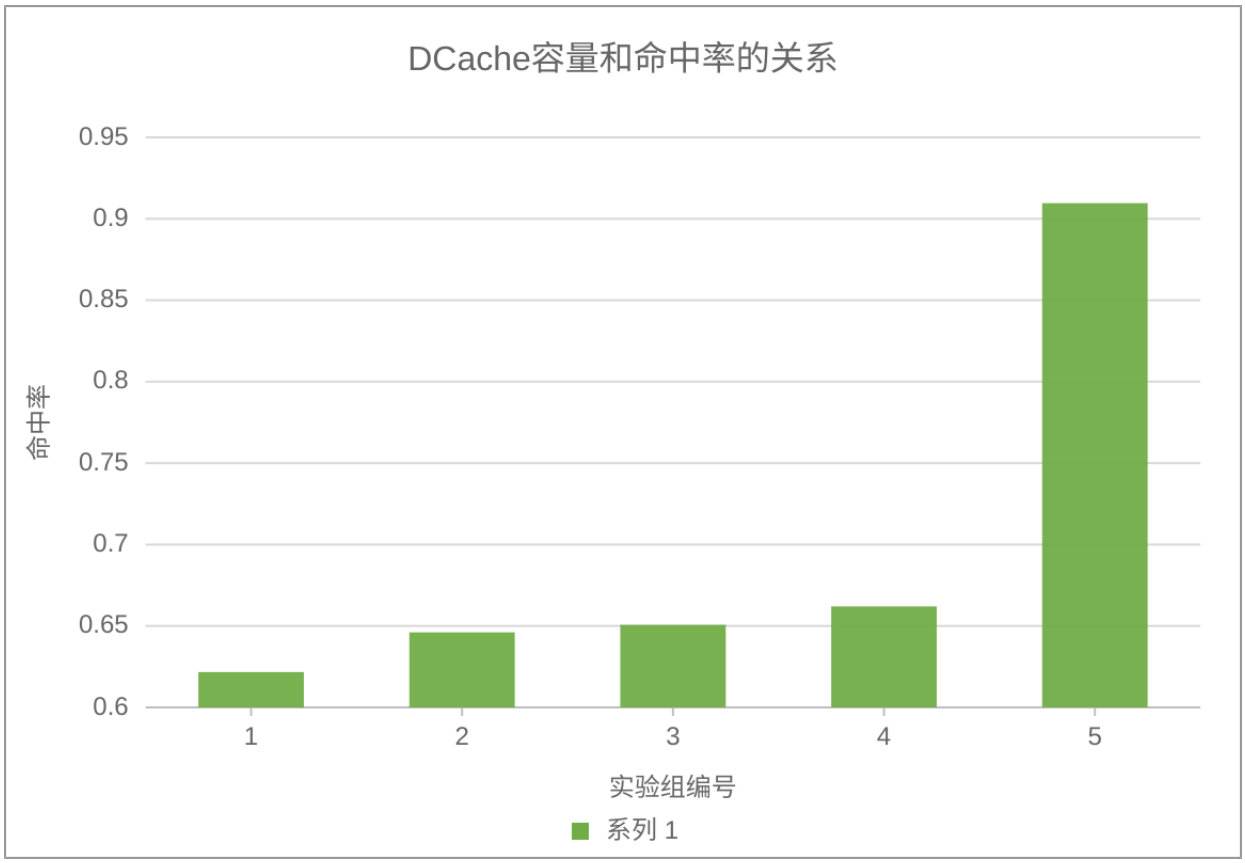


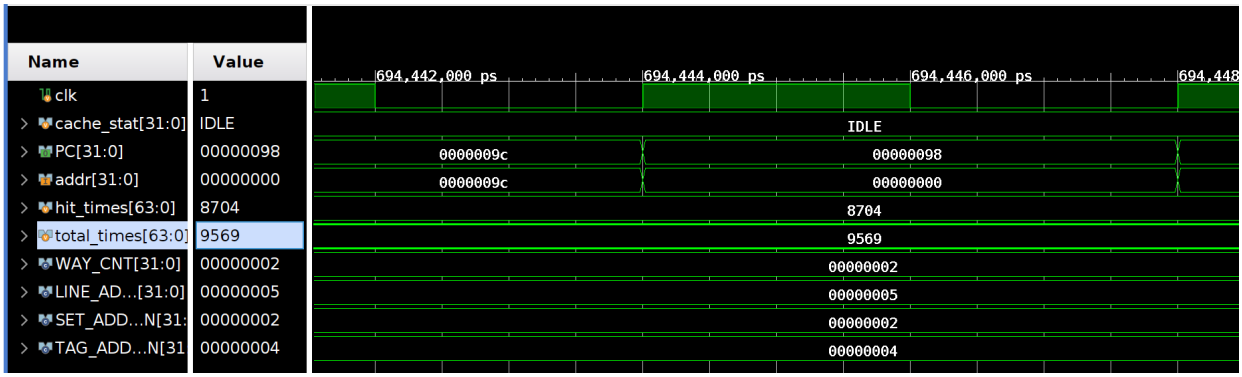
容量对命中率的影响

我们使用比较好的LRU算法来进行探索：

改变LINE_LEN来统计：

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	HIT_TIMES	TOTAL_TIMES	HIT_RATE
1	2	1	2	8	8704	14000	0.6217
2	2	2	2	7	8704	13472	0.6461
3	2	3	2	6	8704	13376	0.6507
4	2	4	2	5	8704	13148	0.6620
5	2	5	2	4	8704	9569	0.9096





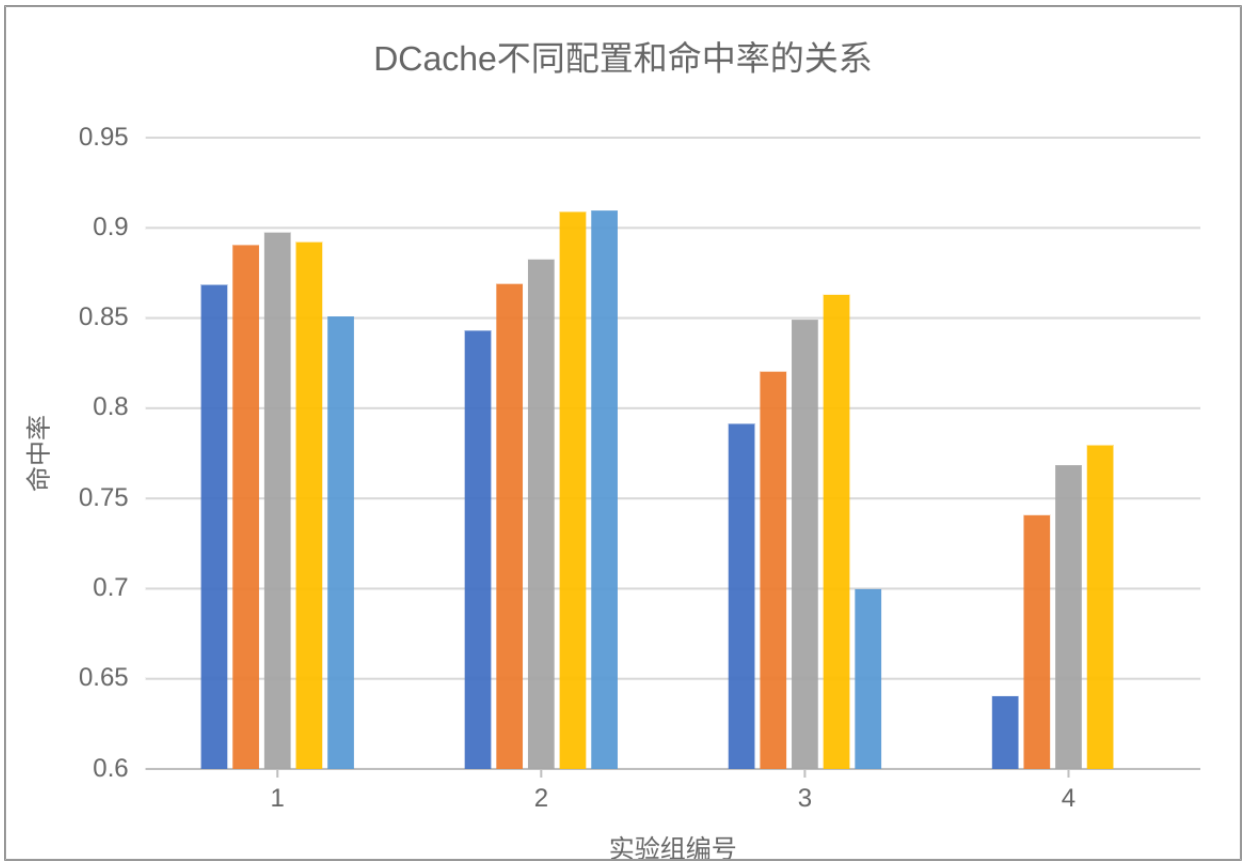
与快速排序类似：当容量逐渐增大时，命中率逐渐提升。当一行可以容纳32个字的时候，命中率有了显著的提升，从理论上分析，这可能是和矩阵乘法的性质有关：每一行都有16个32位数，Cache的一行可以容纳完整的矩阵一行，可以对乘法有显著的加速。

组相连度和组数、行长度对命中率的影响

我们将Cache容量设置为上面的最优值，改变其余几个参数进行实验：

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	HIT_TIMES	TOTAL_TIMES	HIT_RATE
1	1	1	7	3	8704	10023	0.8684
1	1	2	6	3	8704	9775	0.8904
1	1	3	5	3	8704	9699	0.8974
1	1	4	4	3	8704	9757	0.8921
1	1	5	3	3	8704	10229	0.8509
2	2	1	6	4	8704	10325	0.8430

No	WAY_CNT	LINE_LEN	SET_LEN	TAG_LEN	HIT_TIMES	TOTAL_TIMES	HIT_RATE
2	2	2	5	4	8704	10017	0.8689
2	2	3	4	4	8704	9863	0.8825
2	2	4	3	4	8704	9577	0.9088
2	2	5	2	4	8704	9569	0.9096
3	4	1	5	5	8704	10999	0.7913
3	4	2	4	5	8704	10611	0.8203
3	4	3	3	5	8704	10251	0.8491
3	4	4	2	5	8704	10087	0.8629
3	4	5	1	5	8704	12439	0.6997
4	8	1	4	6	8704	13592	0.6404
4	8	2	3	6	8704	11751	0.7407
4	8	3	2	6	8704	11327	0.7684
4	8	4	1	6	8704	11167	0.7794



可以看出，**2路+5位行地址+2位组地址**构成了MatMul程序的最佳Cache配置。

对于矩阵乘法，虽然输入数据规模已经达到了512，但依然和QuickSort一样，对于这样访问空间较小的程序，更多的组导致了更多的冲突，随着组数的增加，每一路的组数也不得不缩小，这很有可能导致一个频繁被使用的块因为地址在组相连情况下和另一个块冲突，而不停被换入换出。而在较低相连度的情况下，能够引发冲突的地址差更大，更不容易发生换入换出。

资源使用情况

在这里，我们比较一下LRU算法和FIFO算法在最佳配置下的资源使用情况（即2路行地址5为组地址2位的情况）

- LRU

```
1. Slice Logic
--------------

+-------------------------+-------+-------+------------+-----------+-------+
|        Site Type        |  Used | Fixed | Prohibited | Available | Util% |
+-------------------------+-------+-------+------------+-----------+-------+
| Slice LUTs*             | 11645 |     0 |          0 |     63400 | 18.37 |
|   LUT as Logic          | 11645 |     0 |          0 |     63400 | 18.37 |
|   LUT as Memory         |     0 |     0 |          0 |     19000 |  0.00 |
| Slice Registers         | 11512 |     0 |          0 |    126800 |  9.08 |
|   Register as Flip Flop | 11512 |     0 |          0 |    126800 |  9.08 |
|   Register as Latch     |     0 |     0 |          0 |    126800 |  0.00 |
| F7 Muxes                |   288 |     0 |          0 |     31700 |  0.91 |
| F8 Muxes                |    96 |     0 |          0 |     15850 |  0.61 |
+-------------------------+-------+-------+------------+-----------+-------+
```

- FIFO

```
1. Slice Logic
--------------

+-------------------------+-------+-------+------------+-----------+-------+
|        Site Type        |  Used | Fixed | Prohibited | Available | Util% |
+-------------------------+-------+-------+------------+-----------+-------+
| Slice LUTs*             | 10861 |     0 |          0 |     63400 | 17.13 |
|   LUT as Logic          | 10861 |     0 |          0 |     63400 | 17.13 |
|   LUT as Memory         |     0 |     0 |          0 |     19000 |  0.00 |
| Slice Registers         | 11505 |     0 |          0 |    126800 |  9.07 |
|   Register as Flip Flop | 11505 |     0 |          0 |    126800 |  9.07 |
|   Register as Latch     |     0 |     0 |          0 |    126800 |  0.00 |
| F7 Muxes                |   288 |     0 |          0 |     31700 |  0.91 |
| F8 Muxes                |    96 |     0 |          0 |     15850 |  0.61 |
+-------------------------+-------+-------+------------+-----------+-------+
```

  可以看到LRU算法的资源消耗量要比FIFO算法的资源消耗量大，这主要是由于LRU的LRU表的更新策略相对于FIFO的更新策略更加复杂。另外，由于LRU需要在每次命中时更新LRU表，其功耗自然也是更大的。

- 在资源消耗方面，比较大的差异还是在于查找表。虽然LRU算法为了查找命中位置使用了更多的寄存器，但毫无疑问，LRU的查找逻辑电路（特别是对于所有位置的并行比较）消耗的资源必定更大。

实验问题记录

- 接入DCache后，出现丢指令问题

这个问题主要来自于一个很隐蔽的问题：这个流水线看似是五级流水线，但ICache却是同步读，故在ICache和IF-ID寄存器中还有一层寄存器。这个寄存器是同步读取的特殊隐藏寄存器，无法接入Stall信号，所以经常会丢指令。好在段间寄存器中留了inst_old接口，只需要把在bubble_ff是不更新inst_old即可

- 解决上述问题后，出现Cache重复写问题

  这里事实上是由于我错误的将一个bubbleE接成了flushE造成的，修改之后流水线成功运行了benchmark

实验总结

- 本次实验让我加深了Cache的理解，又想起了计算机组成原理的期末考题：相连度高不代表命中率高
- 本次实验还让我理解了Cache和流水线的适配方法，以及对于流水线冲突的处理方法
- 通过若干数据的测试，我了解了Cache性能变化和各项参数的关系