计算机体系结构 实验三 Cache 设计和实现

PB18151866 龚小航

【实验目的】

1、权衡 cache size 增大带来的命中率提升收益和存储资源电路面积的开销

2、权衡选择合适的组相连度（相连度增大 cache size 也会增大，但是冲突 miss 会减低）

3、体会使用复杂电路实现复杂替换策略带来的收益和简单替换策略的优势（有时候简单策略比复杂策略效果不差很多甚至可能更好）

4、理解写回法的优劣

【实验环境】

Xilinx Vivado 2019.1

【实验要求】

1、理解已经提供的直接映射策略的 cache，将它修改为 N 路组相连的 cache，并通过提供的 cache 读写测试。

2、使用阶段一编写的 N 路组相连 cache，正确运行我们提供的几个程序。

3、对不同 cache 策略和参数进行性能和资源的测试评估，编写实验报告。

【实验过程及具体实现】

一、将提供的直接映射 Cache 改为 N 路组相联的 Cache

相比于原来的定义，需要把 cache_mem 等加上 WAY_SIZE 域，用于选路。如图所示：

```
localparam LINE_SIZE       = 1 << LINE_ADDR_LEN  ;        // 计算 line 中 word 的数量，即 2^LINE_ADDR_LEN 个word 每 line
localparam SET_SIZE        = 1 << SET_ADDR_LEN   ;        // 计算一共有多少组，即 2^SET_ADDR_LEN 个组
localparam WAY_SIZE        = 1 << WAY_CNT   ;

reg [           31:0] cache_mem    [SET_SIZE][WAY_SIZE][LINE_SIZE]; // SET_SIZE个line，每个line有LINE_SIZE个word
reg [TAG_ADDR_LEN-1:0] cache_tags   [SET_SIZE][WAY_SIZE];           // SET_SIZE个TAG
reg                    valid        [SET_SIZE][WAY_SIZE];           // SET_SIZE个valid(有效位)
reg                    dirty        [SET_SIZE][WAY_SIZE];           // SET_SIZE个dirty(脏位)
```

再在各处将这些变量对应的实例都加上 WAY_SIZE 域与选路信号即可。

改成 N 路组相联后需要实现两种替换算法，FIFO 与 LRU，使用条件编译选择替换策略。

首先定义 cache 命中信号与命中时的选路信息。用 cache_hit 信号表示当前请求数据在 cache 中已命中，利用 for 循环匹配实现；用 way_hit 记录命中时选择的 way 的序号。

```
reg cache_hit = 1'b0;
integer way_hit = 0;
always @ (*) begin              // 判断 输入的address 是否在 cache 中命中
    cache_hit = 1'b0;
    for(integer i=0; i<SET_SIZE; i++) begin
        if( valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr ) begin
            cache_hit = 1'b1;
            way_hit = i;
            break;
        end
    end
end
```

另一方面，用 reg[WAY_CNT-1:0] wayout_choice;//换出way选择 记录若需要换出时当前选中的换出 way 的序号

1、 FIFO 替换策略的实现

对于每一个 set，用 reg [31:0] FIFO_Choice [SET_SIZE]; 记录换出时应该选中的 way 序号。对全局来说，wayout_choice 记录将要换出的 way 号，因此在 cache_state == IDLE 时将 wayout_choice 赋以 FIFO_Choice[set_addr]，即对于当前 set 应该换出的 way 号。而 FIFO_Choice[set_addr]在 cache_state == SWAP_IN_OK 时进行更新，每次选择下一个序号的 way。更新过程如下所示：

$$FIFO\_Choice[set\_addr] = (FIFO\_Choice[set\_addr] + 1) \% WAY\_SIZE;$$

举例来说，当 WAY_CNT=2 时，way 选择顺序为 $0 \to 1 \to 2 \to 3 \to 0 \to 1 \to 2 \to 3 \to 0 \to \cdots\cdots$

完整实现如下所示：

```
/*置换算法实现*/
`ifdef FIFO
reg [31:0] FIFO_Choice [SET_SIZE]; //只需要对于每一个set，分别用一个变量记录该set应该换出的way号即可
always@(*) begin
    if (rst) begin
        wayout_choice = 0;
        for(integer i = 0; i < SET_SIZE; i++) begin
            FIFO_Choice[i] = 0;
        end
    end
    else begin
        ;
    end
    if(cache_stat == IDLE && ~cache_hit)begin
        wayout_choice = FIFO_Choice[set_addr]; //选路时只需要在需要换出时，根据不同的set编号，选择相应的way即可
    end
    if(cache_stat == SWAP_IN_OK)
        FIFO_Choice[set_addr] = (FIFO_Choice[set_addr] + 1) % WAY_SIZE;  //选路完成后，需要对每一组的换出变量进行更新
                                //这样实现的原因是，FIFO策略是先进先出，实际上就是一个循环队列，按照顺序进行更新即可。
    else ;
end
`endif
```

2、 LRU 替换策略的实现

LRU 替换策略在换出块时选择最长时间未被使用过的 way。利用`reg [WAY_CNT - 1 : 0] set_way_select[SET_SIZE][WAY_SIZE];`记录每一个 set 中所有 way 换出时的相对优先级。set_way_select[set_addr][0]换出时优先级最低（最近刚使用过）。Rst 初始化时，需要设置 set_way_select[i][k] = k 以保证每一个 way 序号正确写入。

举例来说，对每一个 set，例如 set_way_select[1][1~4] = {4 2 3 1}代表 4 号 way 刚被使用过，1 号 way 最久未使用。

Cache 命中之后需要对当前 set 中重新设定每个 way 的换出优先级，将刚访问到的 way 优先级置为最低，并将原本在它之前的 way 序号向后顺移一位。用`integer exchange_flag;`表示当前在 set_way_select 中访问的当前元素是否需要右移一位。这里需要注意的是命中时 way_hit 表示的是命中的 way 的序号本身，而在结构 set_way_select[set_addr][]中存储的内容才是 way 的序号。

另一方面，在换入成功时，即 cache_stat == SWAP_IN_OK 就需要将 set_way_select[set_addr][]的最后一个元素换到第 0 号位置，且其余元素向后顺移一位。

完整代码如下所示：

```
`ifdef LRU  //LRU策略换出最长时间未被使用的块
reg [WAY_CNT - 1 : 0] set_way_select[SET_SIZE][WAY_SIZE];
    //对每一个set，例如set_way_select[1][1~4] = {4 2 3 1}代表4号way刚被使用过，1号way最久未使用
reg [WAY_CNT - 1 : 0] set_way_select_temp;      //循环右移后最后插入[SET_SIZE][0]号的交换temp寄存器
integer exchange_flag;//用于记录开始每一元素右移一位的标志信号
always@(*) begin
    if (rst) begin
        for (integer i=0; i < SET_SIZE; i++) begin
            for (integer k=0; k < WAY_SIZE; k++) begin
                set_way_select[i][k] = k;
            end
        end
    end
    else ;
    if(cache_stat == IDLE && ~cache_hit)begin
        wayout_choice = set_way_select[set_addr][WAY_SIZE - 1];//换出选择
    end
    else begin
        ;
    end

   if(cache_stat == IDLE && cache_hit && (wr_req || rd_req))begin
        exchange_flag = 0;
        for(integer i = WAY_SIZE; i >= 1; i--)begin
            if(exchange_flag)begin //不变部分
                set_way_select[set_addr][i] = set_way_select[set_addr][i-1];
            end
            else if(set_way_select[set_addr][i-1] == way_hit)begin//开始每一位右移一次
                exchange_flag = 1;
            end
        end
        set_way_select[set_addr][0] = way_hit;
    end


    if(cache_stat == SWAP_IN_OK) begin
        set_way_select_temp = set_way_select[set_addr][WAY_SIZE-1];
        for (integer i = WAY_SIZE; i > 0; i--) begin
            set_way_select[set_addr][i] = set_way_select[set_addr][i-1];
        end
        set_way_select[set_addr][0] = set_way_select_temp;
    end
end
`endif
```

二、将 cache 与实验 2 实现的 RISC-V 流水线 CPU 联合仿真。

将 cache 替换实验 2 中的数据存储器 Data_Mem 并在 WBSegReg 文件中例化，同时添加 miss 计数以及 hit 计数。WBSegReg 新增实现如下所示：

```
63  reg [31:0] hit_count = 0, miss_count = 0;  // 命中或缺失计数
64  reg [31:0] last_addr = 0;   //
65  wire cache_rd_wr = (|WE) | MemToRegM;
66
67  reg cache_rd_wr_reg;
68  reg cachemiss_reg;
69  always@(posedge clk)begin
70      cachemiss_reg <= CacheMiss;
71      cache_rd_wr_reg <= cache_rd_wr;
72  end
73
74  always @ (posedge cache_rd_wr_reg or posedge rst) begin
75      if(rst) begin
76          hit_count  = 0;
77          miss_count = 0;
78      end
79      else ;
80      if(cachemiss_reg == 1'b1)
81         miss_count = miss_count+1;
82      else ;
83      if(cachemiss_reg == 1'b0)
84         hit_count  = hit_count +1;
85      else ;
86  end
87
88  cache #(
89      .LINE_ADDR_LEN  ( 3             ),
90      .SET_ADDR_LEN   ( 1             ),
91      .TAG_ADDR_LEN   ( 8            ),
92      .WAY_CNT        ( 1             )
93  ) cache_test_instance (
94      .clk            ( clk           ),
95      .rst            ( rst           ),
96      .miss           ( CacheMiss     ),
97      .addr           ( A             ),
98      .rd_req         ( MemToRegM     ),
99      .rd_data        ( RD_raw        ),
100     .wr_req         ( |WE           ),
101     .wr_data        ( WD            )
102 );
```

Cache_rd_wr 信号是 cache 读写请求信号，当此信号取上升沿时判断此时 cachemiss 是否有效，有效则 miss_count+1;否则 hit_count+1；而为了产生正确的结果，cachemiss 信号与 cache_rd_wr 信号均需要设成寄存器类型。通过时序逻辑电路对它们进行赋值即可。

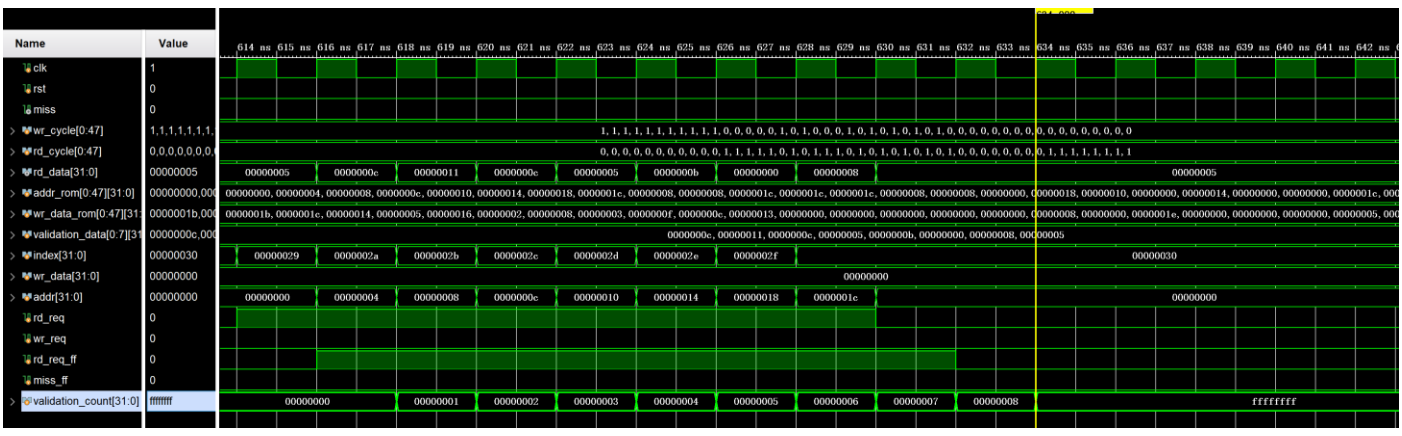
另外，冲突模块也需要在 miss 发生时令流水线产生 50 个周期的停顿。只需要在 CacheMiss 输入信号有效时令所有 stall 信号有效即可。

三、更改 SET_SIZE,WAY_CNT 参数与替换策略，在快速排序与矩阵乘法程序上比较硬件消耗与 cache 缺失率

通过仿真可以知道 cache 缺失情况，而通过综合能得出 cache 消耗的硬件资源数量。

【实验结果】

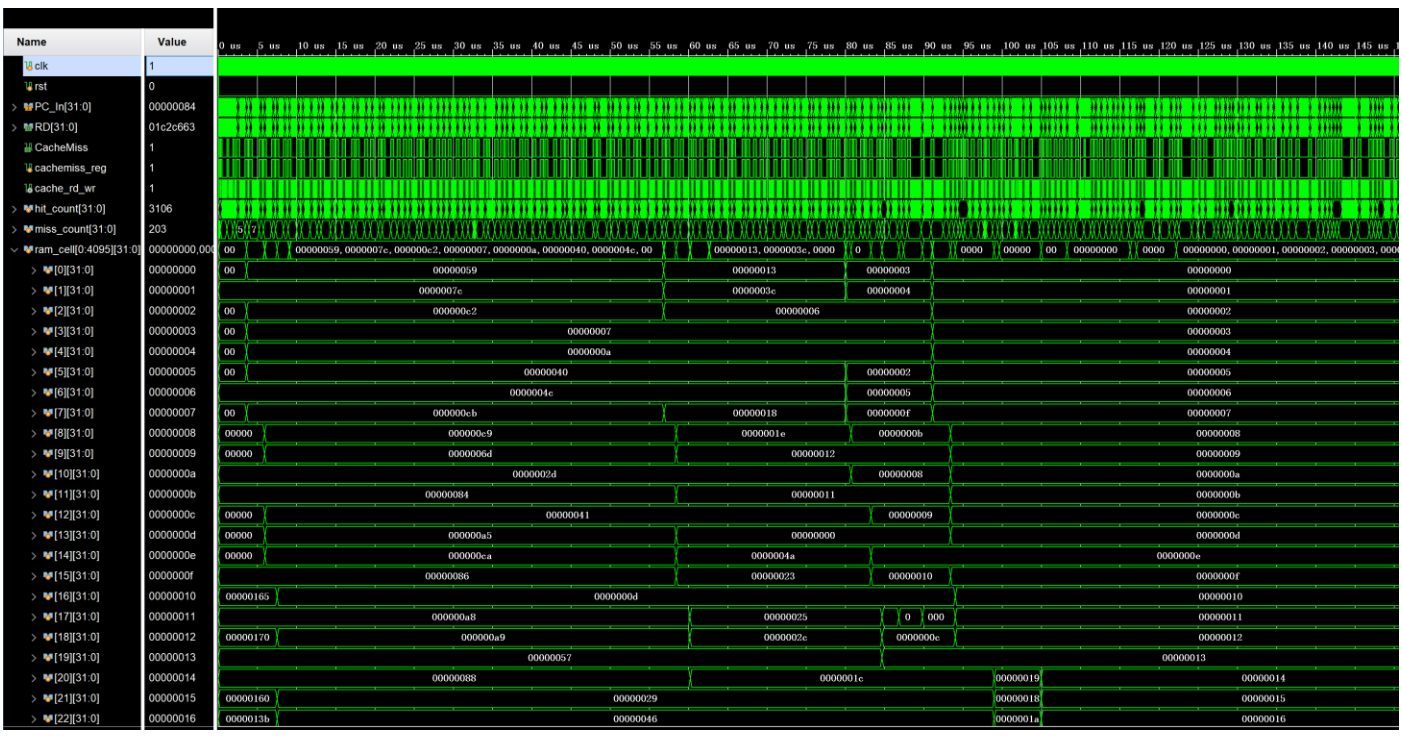
Cache 独立仿真测试：



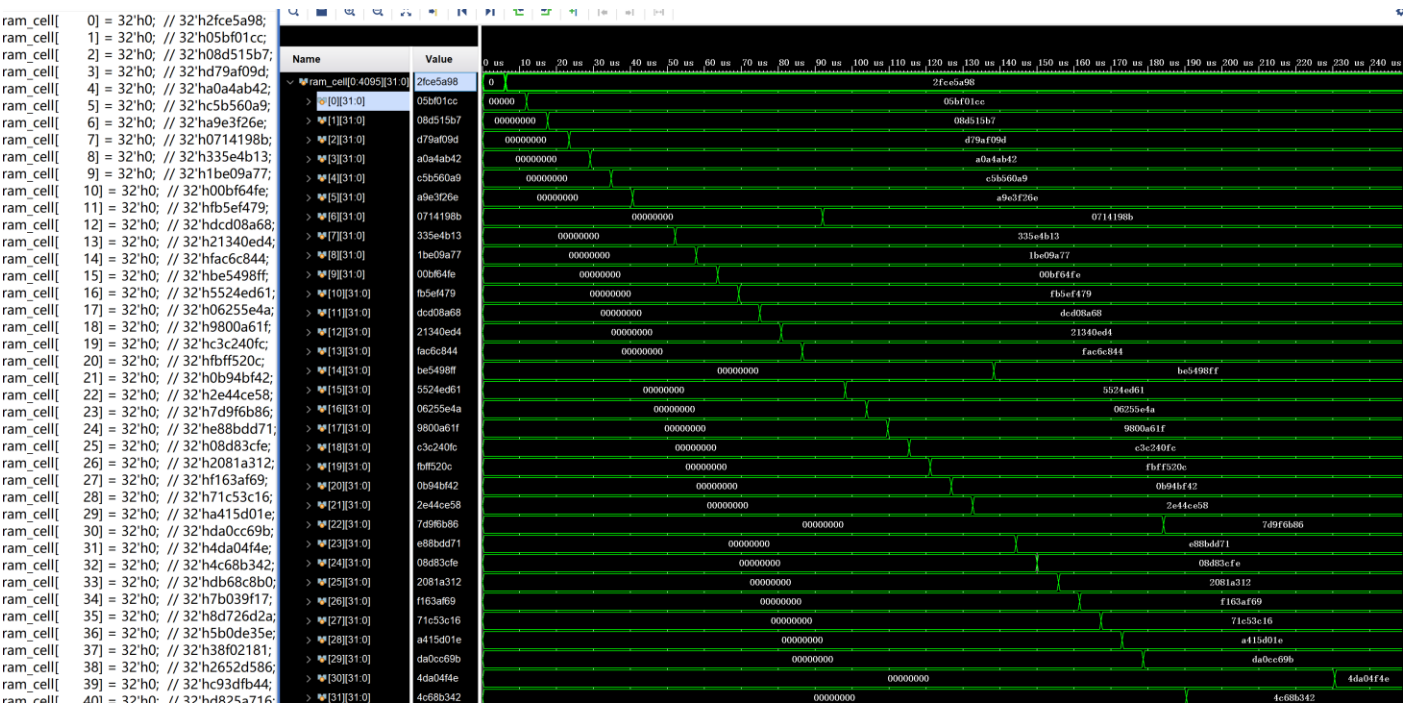
可见 validation_count 变量最后为 -1，结果符合预期。

快速排序与矩阵乘法正确性说明：

512 个随机打乱的数的快速排序仿真结果：



两个 16 × 16 的矩阵相乘仿真结果：



512 个数快速排序在不同的 SET_ADDR_LEN, WAY_CNT 以及不同的替换策略下的缺失率与综合出的硬件消耗：

LINE_ADDR_LEN 固定为 3，不需要改变。

512个数据进行快速排序统计数据

替换策略	SET_ADDR_LEN	WAY_CNT	cache_hit次数	cache_miss次数	cache缺失率	运行结束时间(ns)	LUT使用	FF使用	BRAM使用	I/O使用	hit+miss	miss_rate*(LUT+FF)
FIFO	1	1	12743	839	6.177%	624215	1068	2019	4	81	13582	190.6930496
	1	2	13042	540	3.976%	550352	2250	3084			13582	212.0718598
	1	3	13220	362	2.665%	433840	2888	5217			13582	216.0219408
	1	4	13333	249	1.833%	386556	6114	9506			13582	286.3628332
	2	1	13039	543	3.998%	506212	1918	3075			13582	199.6170667
	2	2	13225	357	2.628%	431928	4006	5189			13582	241.688632
	2	3	13334	248	1.826%	386132	6762	9447			13582	295.9676042
	2	4	13475	107	0.788%	327808	14665	17959			13582	257.0142836
	3	1	13226	356	2.621%	431256	1987	5188			13582	188.0650861
	3	2	13337	245	1.804%	385068	4041	9398			13582	242.420483
	3	3	13484	98	0.722%	323992	6594	17845			13582	176.3379473
	3	4	13509	73	0.537%	311524	12810	34702			13582	255.365631
	4	1	13343	239	1.760%	382732	3773	9386			13582	231.5565454
	4	2	13480	102	0.751%	325692	7613	17800			13582	190.8500957
	4	3	13509	73	0.537%	311524	13079	34620			13582	256.3707112
	4	4	13509	73	0.537%	311524	24511	68210			13582	498.353188
LRU	1	1	12783	799	5.883%	607060	1089	2022			13582	183.0134737
	1	2	13075	507	3.733%	491360	2240	3097			13582	199.2238993
	1	3	13245	337	2.481%	423240	3048	5277			13582	206.5619938
	1	4	13335	247	1.819%	385708	6447	9627			13582	292.3190988
	2	1	13075	507	3.733%	491156	1939	3079			13582	187.3160065
	2	2	13247	335	2.466%	422600	3998	5230			13582	227.6085996
	2	3	13341	241	1.774%	383370	7089	9532			13582	294.924238
	2	4	13433	149	1.097%	345615	14407	18174			13582	357.4266676
	3	1	13242	340	2.503%	424512	2036	5194			13582	180.989545
	3	2	13349	233	1.716%	379980	4302	9443			13582	235.7962745
	3	3	13435	147	1.082%	344768	7113	18034			13582	272.1697099
	3	4	13509	73	0.537%	311524	14760	35232			13582	268.6950375
	4	1	13344	238	1.752%	382308	3844	9404			13582	232.1472537
	4	2	13465	117	0.861%	332052	8022	17094			13582	216.3578265
	4	3	13509	73	0.537%	311524	14577	34961			13582	266.2548962
	4	4	13509	73	0.537%	311524	27011	69202			13582	517.1218525

从统计数据中得到的结论：

1、从上图的数据中可以看出，随着 cache set 数和组相连度的增加，整体的 miss_rate 降低，而硬件资源消耗增加。

2、LRU 与 FIFO 替换策略相比较来说，在 miss_rate 上 LRU 略低于 FIFO，而资源占用高于 FIFO。这就是复杂替换策略能降低缺失率但是消耗更多，硬件更复杂。

3、一个有趣的地方在于当选择 SET_ADDR_LEN 和 WAY_CNT 分别是 2/3 和 4/1 时，虽然 cache 容量是一样的，但不管是 miss 率还是资源占用数量都是 4/1 更低，简单的分析后，这应该是路数增加带来的多路选择器增加导致的，而 miss 率的降低可能是因为，当组增加时，隐性的将内存划分成了更多的块，而使得碰撞减少。

4、随着 cache 容量不断增加，miss 率不断降低，最终所有内存都会被读入 cache，但是这么做是有边界的，到达饱和以后再增加 set 数或相连度只会增加硬件消耗而没有丝毫收益。

根据 MissRate*（FF+LUT）的值来判断，取最小值，可知在这一问题中应该选择

FIFO + 3 SET_ADDR_LEN + 3 WAY_CNT

两个 16 × 16 的矩阵相乘在不同的 SET_ADDR_LEN, WAY_CNT 及不同的替换策略下的缺失率与综合出的硬件消耗：LINE_ADDR_LEN 固定为 3，不需要改变。

16×16矩阵乘法的统计数据

替换策略	SET_ADDR_LEN	WAY_CNT	cache_hit次数	cache_miss次数	cache缺失率	运行结束时间(ns)	LUT使用	FF使用	BRAM使用	I/O使用	hit+miss	miss_rate*(LUT+FF)
FIFO	1	1	3264	5440	62.500%	1487532	1068	2019	4	81	8704	1929.38
	1	2	3776	4928	56.618%	1376936	2250	3084			8704	3019.99
	1	3	4032	4672	53.676%	1321640	2888	5217			8704	4350.48
	1	4	4049	4655	53.481%	1317760	6114	9506			8704	8353.76
	2	1	3648	5056	58.088%	1404584	1918	3075			8704	2900.35
	2	2	3904	4800	55.147%	1349288	4006	5189			8704	5070.77
	2	3	5833	2871	32.985%	932416	6762	9447			8704	5346.51
	2	4	8558	146	1.677%	294576	14665	17959			8704	547.23
	3	1	3840	4864	55.882%	1363112	1987	5188			8704	4009.56
	3	2	6965	1739	19.979%	687904	4041	9398			8704	2685.02
	3	3	8558	146	1.677%	294556	6594	17845			8704	409.94
	3	4	8608	96	1.103%	280240	12810	34702			8704	524.03
	4	1	7531	1173	13.477%	565648	3773	9386			8704	1773.38
	4	2	8560	144	1.654%	293728	7613	17800			8704	420.44
	4	3	8608	96	1.103%	280040	13079	34620			8704	526.09
	4	4	8608	96	1.103%	280040	24511	68210			8704	1022.66
LRU	1	1	4032	4672	53.676%	1321644	1089	2022			8704	1669.88
	1	2	4032	4672	53.676%	1321640	2240	3097			8704	2864.71
	1	3	4032	4672	53.676%	1321640	3048	5277			8704	4468.57
	1	4	4289	4415	50.724%	1265920	6447	9627			8704	8153.34
	2	1	4032	4672	53.676%	1321640	1939	3079			8704	2693.49
	2	2	4032	4672	53.676%	1321640	3998	5230			8704	4953.26
	2	3	6081	2623	30.136%	878848	7089	9532			8704	5008.83
	2	4	8579	125	1.436%	292332	14407	18174			8704	467.90
	3	1	4032	4672	53.676%	1321640	2036	5194			8704	3880.81
	3	2	7157	1547	17.773%	646432	4302	9443			8704	2442.96
	3	3	8581	123	1.413%	291484	7113	18034			8704	355.36
	3	4	8608	96	1.103%	280040	14760	35232			8704	551.38
	4	1	7545	1159	13.316%	562624	3866	9404			8704	1767.00
	4	2	8585	119	1.367%	289788	8022	17904			8704	354.46
	4	3	8608	96	1.103%	280040	14577	34961			8704	546.38
	4	4	8608	96	1.103%	280040	27011	69202			8704	1061.17

从统计数据中得到的结论：

1、矩阵乘法的 cache 缺失率显著高于快速排序，这是因为两个程序本身的局部性差异。快速排序在划分到较小数组后，其局部性是很好的。而矩阵链乘所需空间多，局部性差，每一轮迭代都需要横跨 dp 数组，而不是在某一部分做访问。因此其缺失率较高。

2、当 cache 容量增大，缺失率降低。占用资源增加。

3、当 cache 容量达到一定程度后，增加组相连度带来的 miss 率降低显著优于继续增加 set 数量。

4、LRU 与 FIFO 替换策略相比较来说，在 miss_rate 上 LRU 略低于 FIFO，而资源占用高于 FIFO。这就是复杂替换策略能降低缺失率但是消耗更多，硬件更复杂。

根据 MissRate*（FF+LUT）的值来判断，取最小值，可知在这一问题中应该选择

LRU + 4 SET_ADDR_LEN + 2 WAY_CNT。

【实验总结】

本次实验的内容较上次实验整体更少，但是小问题比较多。

1. 对 miss 信号做 stall 时，需要把其优先级置为最高，因为其是在 MEM 段产生的结构冲突，而我们之前的冒险冲突都是在 EX 段和 ID 段产生，越靠后的冲突冒险优先级越高。
2. 对 miss 信号做 stall 时，需要把 5 段流水都 stall，即使是 WB 阶段。
3. verilog 编程中不能把 wire 信号写进时序逻辑，需要为其分配一个寄存器。原因是 wire 信号的不稳定导致在时序电路中会产生不稳定的结果。
4. 原本给出的实验框架中数据通路不支持连续两个周期的 stall。经过本次实验后，我对 cache 的结构及替换策略有了更深入的理解
5. 写回法的优劣：

   写回法的优点是：速度快并且减少了访存次数。但他最大劣势是：一致性很难维护。就像在这一次实验中，最后检查 mem 的结果时，不一定是完全正确的，因为很多块都还在 cache 中，并没有被写回，这就造成了一致性的问题。而在现代的计算机中，功能部件更加复杂，对一致性的要求也就更高，所以维护一致性也就有了更高的难度。