计算机体系结构Lab3实验报告

廖洲洲 PB17081504

组相连Cache的实现

组相联映象：主存中每一块可以被放置在Cache中唯一的一个组(set)中的任意一个位置，组由若干块构成，若一组由n块构成，我们称N路组相联

- 组间直接映像
- 组内全相联
- 相联度N越高，cache空间利用率就越高，块冲突概率就越小，失效率就越低
- N值越大，失效率就越低，但Cache的实现就越复杂，代价越大

修改Cache块

原来直接映射的cache块中的cache_mem、cache_tags、valid和dirty数组增加一个维度，该维度的大小为WAY_CNT

```
//全部添加一个维度，该维度的大小为WAY_CNT，表示一个组有WAY_CNT路
reg [            31:0] cache_mem    [SET_SIZE][WAY_CNT][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_CNT];            // SET_SIZE
个TAG
reg                    valid        [SET_SIZE][WAY_CNT];            // SET_SIZE
个valid(有效位)
reg                    dirty        [SET_SIZE][WAY_CNT];            // SET_SIZE
个dirty(脏位)
```

实现并行命中判断

组相连cache需要在组内并行的判断每路line是否命中,Verilog中的for循环循环几次就是将相同的电路复制几次，因此可以直接用for循环实现并行判断

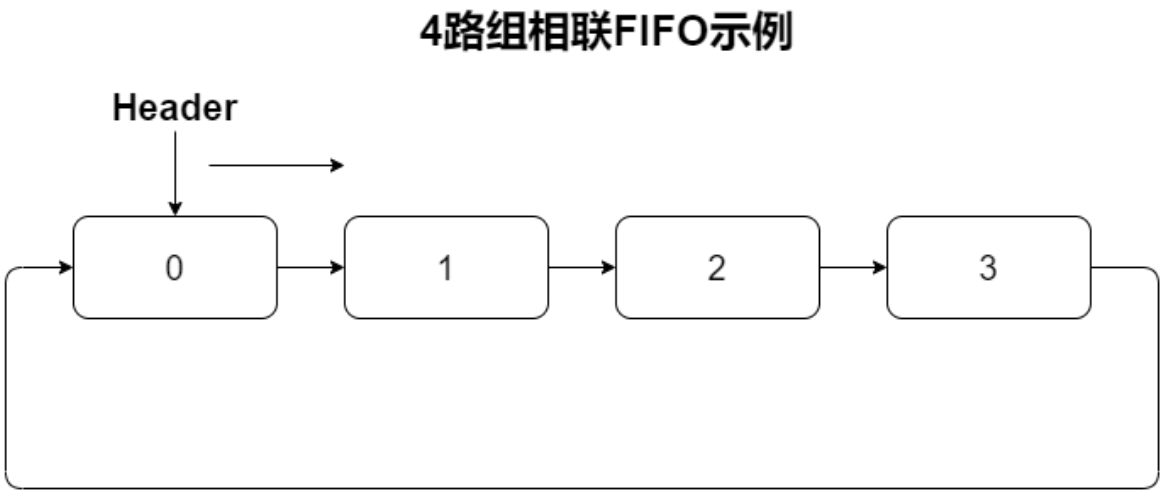
```
always @ (*) begin                  // 判断 输入的address 是否在 cache 中命中
    //实现组相连的地址命中判断，首先并行判断组内每路的valid是否有效,若有效再判断该路的tag,若
tag相等，则命中，保存该路地址
    //在Verilog中for循环表示的是电路的硬件行为，循环几次，就是将相同的电路复制几次，故直接用
for循环实现并行判断
    cache_hit = 1'b0;
    for(integer i = 0; i < WAY_CNT; i++) begin
        if(valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr) begin
            cache_hit <= 1'b1;
            way_addr <= i;
        end
    end
end
```

实现替换策略

实现了FIFO、伪LRU和LRU替换策略

FIFO的实现

cache初始化后，cache中每一路都是未使用的，因此对于开始时，每组的cache访问都是缺失的，当组未满时，按0、1、2……WAY_CNT-1的顺序换入，即当需要将数据换入且cache组有空时，按顺序选择候选块。之后cache中某组被填满后，该组的每一路都会是有效的，我们按先替换0，再替换1，然后替换2……替换WAY_CNT-1,替换1……的顺序来替换，于是实现了FIFO。于是每组维护一个header变量就可以了，header就是就是缺失时要被替换的块，它每替换一个块就加1，当大于WAY_CNT-1时又从1开始。相当于一个循环队列



主要代码

```
        SWAP_IN_OK: begin            // 上一个周期换入成功，这周期将主存读出的line写入
cache，并更新tag，置高valid，置低dirty
                        for(integer i=0; i<LINE_SIZE; i++)
 cache_mem[mem_rd_set_addr][header[mem_rd_set_addr]][i] <= mem_rd_line[i];
                        cache_tags[mem_rd_set_addr][header[mem_rd_set_addr]] <=
mem_rd_tag_addr;
                        valid     [mem_rd_set_addr][header[mem_rd_set_addr]] <=
1'b1;
                        dirty     [mem_rd_set_addr][header[mem_rd_set_addr]] <=
1'b0;
                        cache_stat <= IDLE;         // 回到就绪状态
                        //修改头指针
                        if(header[mem_rd_set_addr] < WAY_CNT) begin
                            header[mem_rd_set_addr] <= header[mem_rd_set_addr] +
1;
                        end
                        else begin
                            header[mem_rd_set_addr] <= 0;
                        end
                    end![1590723388385](C:\Users\廖洲洲
\AppData\Roaming\Typora\typora-user-images\1590723388385.png)
```

伪LRU的实现

为缓存每个组设置一组比特，每个比特对应缓存中的一路；在访问一组时开启一个特定比特，这一比特与包含所需块的路相对应，如果与一个组相关联的所有比特都被开启，除最近刚被开启的比特外，将所有其它比特关闭。在必须替换一个块时，处理器从相应被关闭的路中选择一个块，如果有多种选择，则随机选定。

主要代码

```
if(cache_hit) begin
    //修改used
    used[set_addr][way_addr] = 1'b1;
    if(used[set_addr] == MAX ) begin
        used[set_addr] = 0;
        used[set_addr][way_addr] = 1'b1;
    end
```

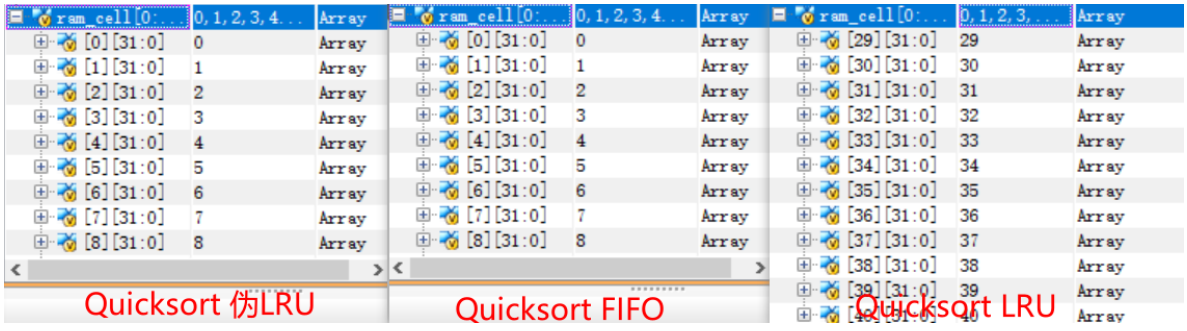
LRU的实现

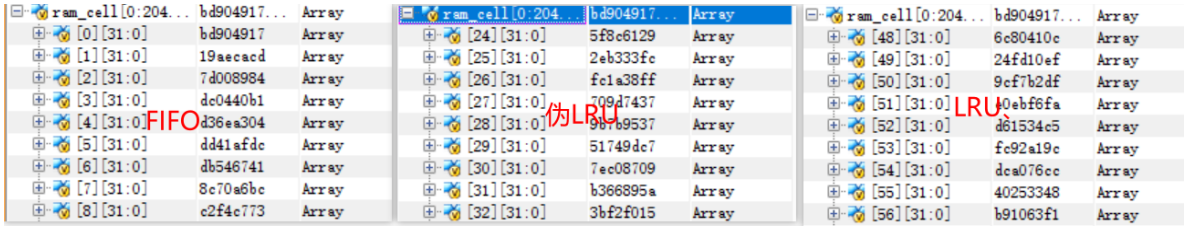
为缓存的每一路维护一个计数值used，当某一路被命中时，对应的计数值置为0，其它路计数值统统加1。相当于记录了该路上次被访问以来经历的时间，当需要替换块时，选择对应组中路的计数值最大的路替换出去即可，因为计数值最大的路被访问的时间总是离当前时间最久，于是实现了最近最少使用。

主要代码

```
always @(*) begin
    MAX = 0;
    replace_way_addr = 0;
    for(integer i = 0; i < WAY_CNT; i++) begin
        if(used[set_addr][i] > MAX) begin
            replace_way_addr = i;
            MAX = used[set_addr][i];
        end
    end
end

always @(posedge clk) begin
    if(cache_stat == IDLE && cache_hit && (rd_req | wr_req)) begin
        used[set_addr][way_addr] <= 0;
        for(integer i =0; i < WAY_CNT; i++) begin
            if(i != way_addr) begin
                used[set_addr][i] <= used[set_addr][i] +1;
            end
        end
    end
end
```





Cache的性能分析

Cache的资源占用

- Cache的有效大小为 $2^{(SET\_ADDR\_LEN+LINE\_ADDR\_LEN)}$*WAY_CNT 个字。
- Cache资源除了用于保存主存数据外，还需要维护有效位、脏位、标志及相应替换策略使用的资源。
- LRU 置换算法虽然是一种比较好的算法，但要求系统有较多的支持硬件。
- 为了排除主存大小对资源占用的影响，需要固定主存的大小。主存大小是 $2^{(LINE_ADDR_LEN+SET_ADDR_LEN+TAG_ADDR_LEN)}$个字。因此实验固定 LINE_ADDR_LEN+SET_ADDR_LEN+TAG_ADDR_LEN=9

固定主存大小，固定缓存组数，固定每路字数，观察不同路数下资源使用

8组，每路8字，随着路数增加，Cache 的大小也增加

FIFO 1路 2路 4路 8路

Resource	Estimation	Available	Utilizat...
LUT	1350	32600	4.14
FF	3025	65200	4.64
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	1	32	3.13

Resource	Estimation	Available	Utilizat...
LUT	2013	32600	6.17
FF	5121	65200	7.85
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	1	32	3.13

Resource	Estimation	Available	Utilizat...
LUT	3850	32600	11.81
FF	9305	65200	14.27
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

Resource	Estimation	Available	Utilizat...
LUT	7399	32600	22.70
FF	17665	65200	27.09
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

伪LRU 1路 2路 4路 8路

Resource	Estimation	Available	Utilizat...
LUT	1389	32600	4.26
FF	3017	65200	4.63
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	1	32	3.13

Resource	Estimation	Available	Utilizat...
LUT	2077	32600	6.37
FF	5124	65200	7.86
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	1	32	3.13

Resource	Estimation	Available	Utilizat...
LUT	3891	32600	11.94
FF	9319	65200	14.29
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

Resource	Estimation	Available	Utilizat...
LUT	7517	32600	23.06
FF	17708	65200	27.16
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

LRU 1路 2路 4路 8 路

Resource	Estimation	Available	Utilizat...
LUT	1339	32600	4.11
FF	3017	65200	4.63
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	1	32	3.13

Resource	Estimation	Available	Utilizat...
LUT	1335	32600	4.10
FF	3017	65200	4.63
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	1	32	3.13

Resource	Estimation	Available	Utilizat...
LUT	4852	32600	14.88
FF	10049	65200	15.41
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	1	32	3.13

Resource	Estimation	Available	Utilizat...
LUT	9956	32600	30.54
FF	19425	65200	29.79
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

结论：**从图中可以看出，FIFO、伪LRU、LRU都会随着缓存大小的增加而资源使用增加，因为FIFO和伪LRU使用的额外Reg和Wire相差不多，故资源使用相差不大。而LRU因为需要为每一路维护一个较大的寄存器，同时得实现查找最大值操作，故资源使用较FIFO和伪LRU多许多。同时发现，随着路数得增大，LRU的成本变得越来越高，与另外两种策略的资源使用相差越来越大。**

固定主存大小，固定缓存组数，固定每组路数，观察每路不同字数下资源的使用

主存大小不变，Cache大小保持不变，保持组数不变（8组），令路数*每路字数不变，减少路数，增大每路字数，观察资源使用情况

FIFO 8路每路8字 4路每路16字

Resource	Estimation	Available	Utilizat...
LUT	7399	32600	22.70
FF	17665	65200	27.09
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

Resource	Estimation	Available	Utilizat...
LUT	7273	32600	22.31
FF	18230	65200	27.96
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

伪LRU 8路每路8字 4路每路16字

Resource	Estimation	Available	Utilizat...
LUT	7517	32600	23.06
FF	17708	65200	27.16
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

Resource	Estimation	Available	Utilizat...
LUT	7324	32600	22.47
FF	18244	65200	27.98
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

LRU 8路每路8字 4路每路16字

Resource	Estimation	Available	Utilizat...
LUT	9956	32600	30.54
FF	19425	65200	29.79
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

Resource	Estimation	Available	Utilizat...
LUT	8447	32600	25.91
FF	18974	65200	29.10
BRAM	0.50	75	0.67
IO	78	150	52.00
BUFG	2	32	6.25

结论：**从图中可以看出，LRU依旧使用资源最多。除此之外，虽然缓存大小没变，但是随着路数的减少，需要维护的信息减少，故LUT的资源使用会减少，尤其是LRU的资源使用变化明显，说明LRU为维护相关信息需要消耗大量的硬件资源。**

不同策略下CPU Cache 性能

- 使用的是未添加了从memory load数据的代码，这样能减少对Cache命中率分析的影响
- 由于同一程序访问Cache次数是相同的，因此使用Cache缺失数可以代表Cache性能

快速排序(256个数)

1.组数不变（8组），每路字数不变（8字），增大路数，观察Cpu性能

总访问次数相同，因此比较缺失数即可比较性能好坏

FIFO 1路、2路、4路组相联

miss_count[31:0]	608	Array	miss_count[31:0]	460	Array	miss_count[...	396	Array
rd_count[31:0]	4049	Array	rd_count[31:0]	4049	Array	rd_count[31:0]	4049	Array
wr_count[31:0]	1949	Array	wr_count[31:0]	1949	Array	wr_count[31:0]	1949	Array
hit_count[31:0]	5390	Array	hit_count[31:0]	5538	Array	hit_count[3...	5602	Array
count[31:0]	5998	Array	count[31:0]	5998	Array	count[31:0]	5998	Array

伪LRU 1路、2路、4路组相联

miss_count[...	608	Array	miss_count[...	465	Array	miss_count[...	400	Array
rd_count[31:0]	4049	Array	rd_count[31:0]	4049	Array	rd_count[31:0]	4049	Array
wr_count[31:0]	1949	Array	wr_count[31:0]	1949	Array	wr_count[31:0]	1949	Array
hit_count[3...	5390	Array	hit_count[3...	5533	Array	hit_count[3...	5598	Array
count[31:0]	5998	Array	count[31:0]	5998	Array	count[31:0]	5998	Array

LRU 1路、2路、4路组相联

miss_count[...	608	Array	miss_count[31:0]	454	Array	miss_count[31:0]	396	Array
rd_count[31:0]	4049	Array	rd_count[31:0]	4049	Array	rd_count[31:0]	4049	Array
wr_count[31:0]	1949	Array	wr_count[31:0]	1949	Array	wr_count[31:0]	1949	Array
hit_count[3...	5390	Array	hit_count[31:0]	5544	Array	hit_count[31:0]	5602	Array
count[31:0]	5998	Array	count[31:0]	5998	Array	count[31:0]	5998	Array

结论：**从图中可以看出，单纯地增加路数，相当于增大缓存大小，因此缺失数随着路数的增加而减少。可以看到，当缓存较小时，LRU胜过其它方法;当缓存较大时，各方法无较大区别。**

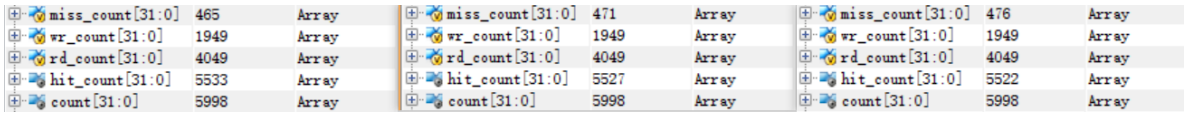
2.缓存大小不变，增加路数

保持缓存大小为128字，路数依次为2、4、8

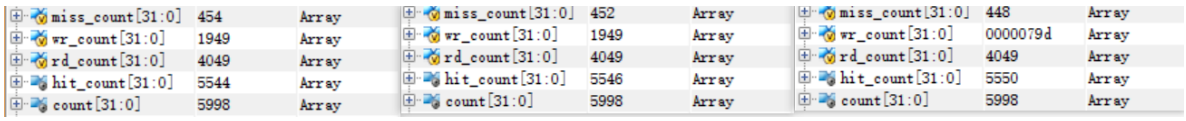
FIFO

miss_count[31:0]	460	Array	miss_count[31:0]	459	Array	miss_count[31:0]	457	Array
wr_count[31:0]	1949	Array	wr_count[31:0]	1949	Array	wr_count[31:0]	1949	Array
rd_count[31:0]	4049	Array	rd_count[31:0]	4049	Array	rd_count[31:0]	4049	Array
hit_count[31:0]	5538	Array	hit_count[31:0]	5539	Array	hit_count[31:0]	5541	Array
count[31:0]	5998	Array	count[31:0]	5998	Array	count[31:0]	5998	Array

伪LRU



LRU

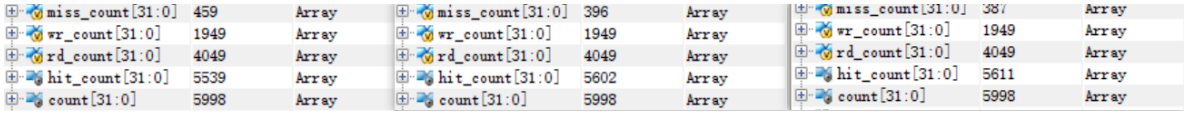


结论：**保持缓存大小不变，增加路数，对FIFO来说，缺失率会略有下降；对伪LRU来说，缺失率却上升了；但是对LRU来说，其随着路数的增加，缺失率减少。纵向比较来说，LRU的缺失率是最小的。**

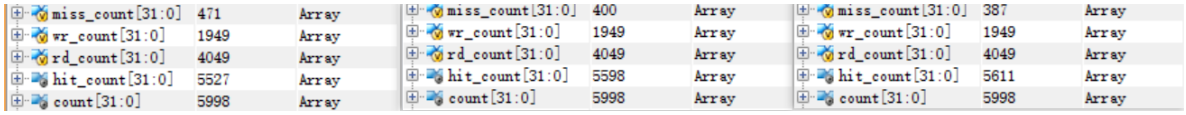
3.路数不变，增加缓存

保持路数为4路，组数分别为4组、8组、16组

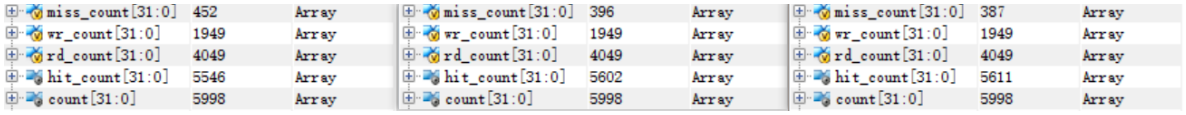
FIFO



伪LRU



LRU



结论：**对FIFO、伪LRU、LRU来说，随着组数的增加，即Cache容量的增大，缺失率均会减小。纵向比较来说，仍是LRU的性能最好。但是，随着缓存容量增加，性能差别不大**

矩阵乘法（16*16矩阵）

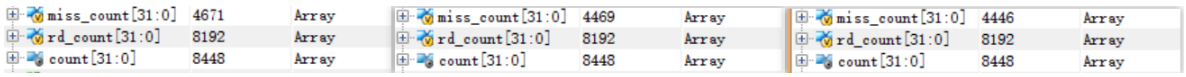
总访问次数相同，因此比较缺失数即可比较性能好坏

1.缓存大小不变，增加路数

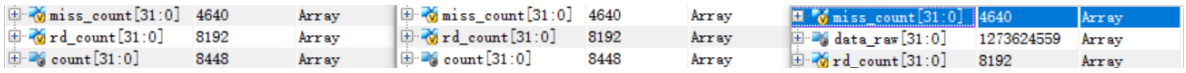
保持缓存大小为128字，路数依次为2、4、8

FIFO

伪LRU



LRU

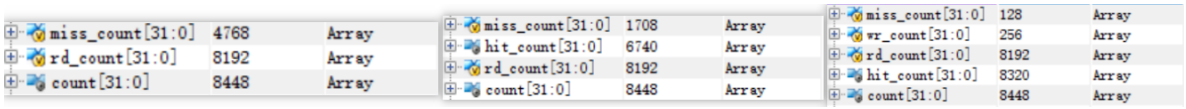


结论：**随着路数增加，缺失率有所下降。**

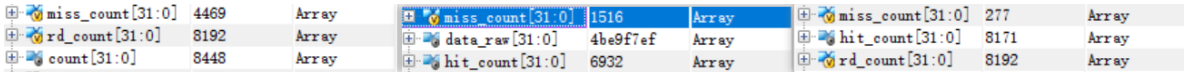
2.路数不变，增加缓存

保持路数为4路，组数分别为4组、8组、16组

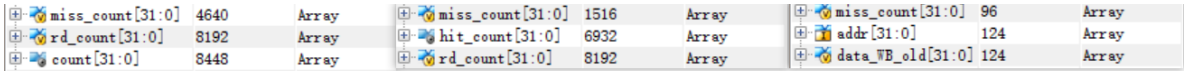
FIFO



伪LRU



LRU



结论：**随着缓存容量的增加，Cache失效率大大减少。大容量Cache适合矩阵运算，其中LRU的性能最好。**

总结

- 不同的Cache策略都会随着缓存大小的增加而使用更多硬件资源
- LRU的实现更加复杂，因此会占用更多的硬件资源
- 显然，随着Cache容量的增大，Cache的缺失率降低
- 一般来说，Cache随着组相联度的增大，即每组路数的增大，失效率降低
- 当Cache容量较小时，LRU的性能相对于其它策略来说会更好些；但是当容量增大，性能差别就会减小。
- 对于快速排序，权衡性能和电路面积，建议使用“4组，4路，每路8字，FIFO”策略，电路面积小，缺失率也不高。
- 对于矩阵运算，权衡性能和电路面积，建议使用“16组，4路，每路8字，FIFO”策略，失效率很低，电路面积也不会大。