系统结构第一次实验报告

实验环境为 linux

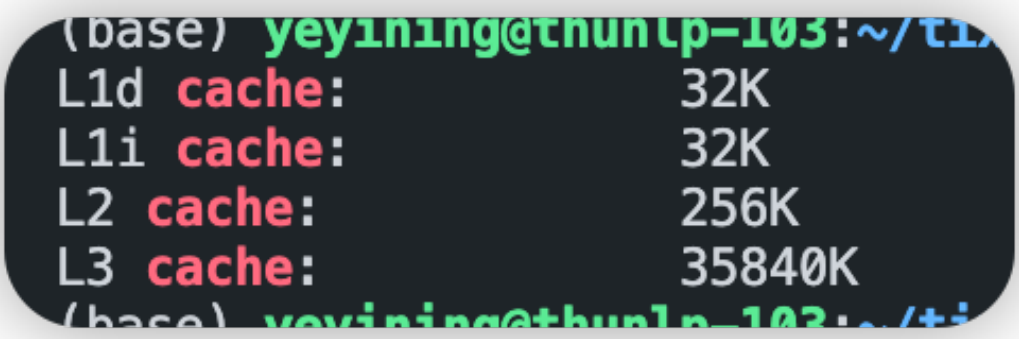
> 叶奕宁 2019013287 计97

1.测量Cache Size

首先调用

```
lscpu | grep cache
```

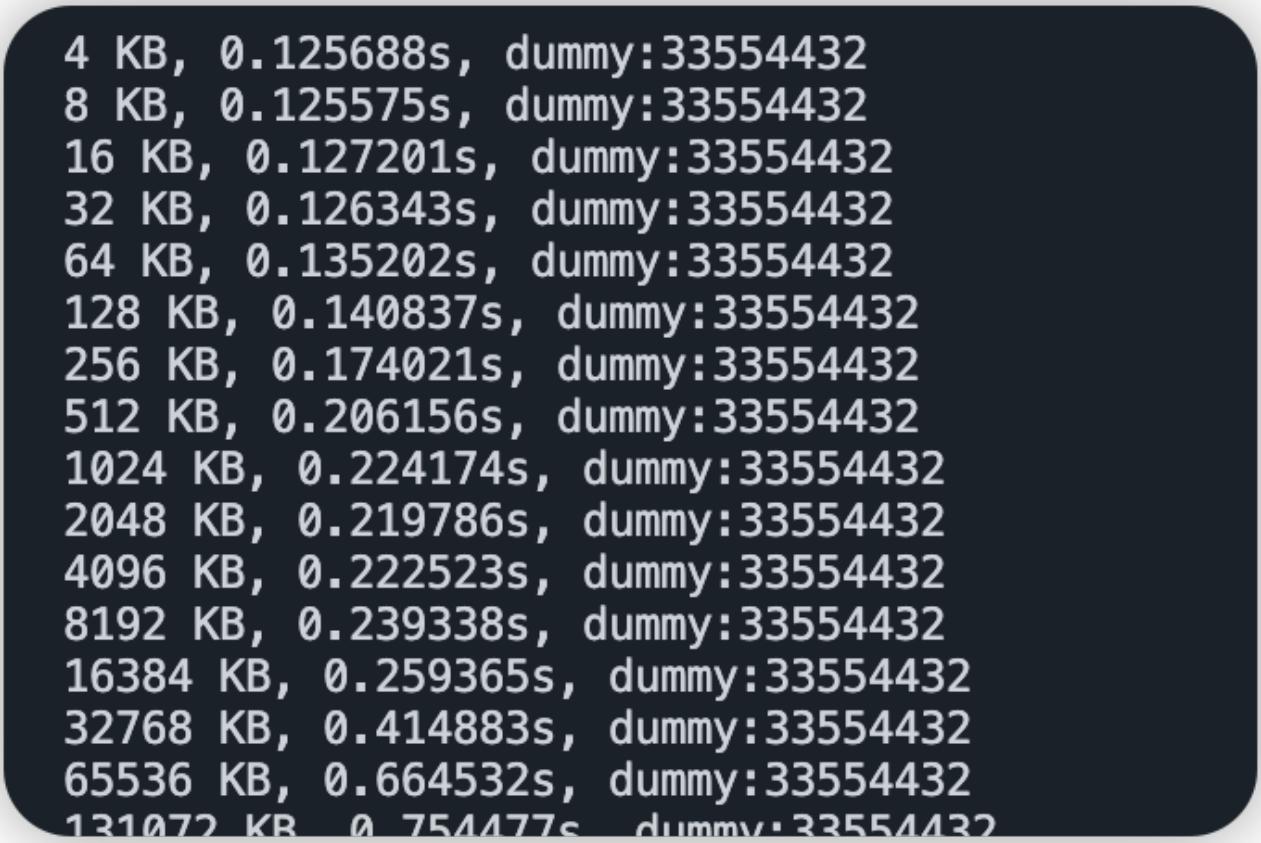
看出系统的缓存大小为

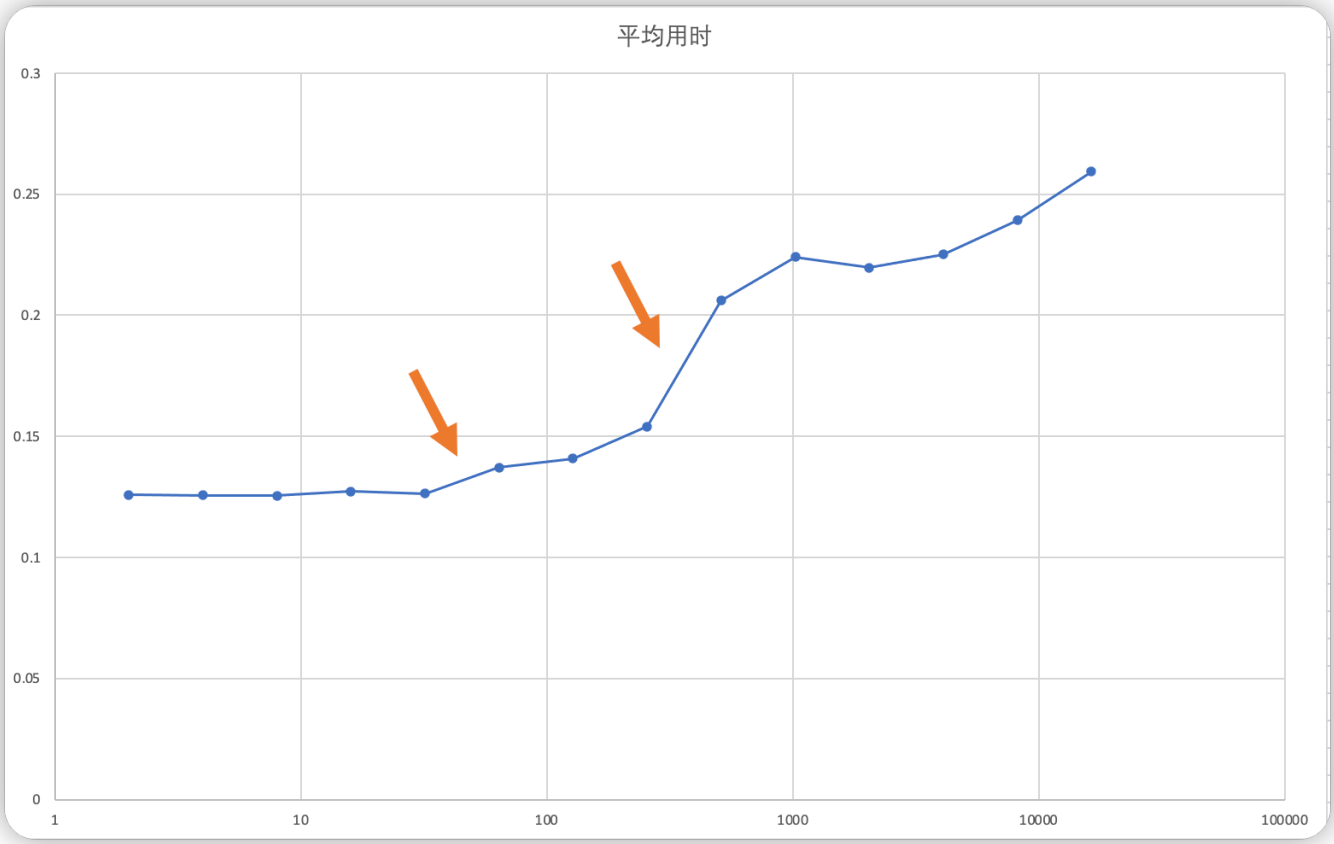


接下来，通过随机访存测量缓存大小

- 开一个数组
- 多次随机访问数组位置(用伪随机算法加速,提前算好随机数序列，千万别在循环内部生成随机数，要不然访存时间测不出来了...)
- 计算时间
- 改变数组大小多次尝试

得到访问时间如下：

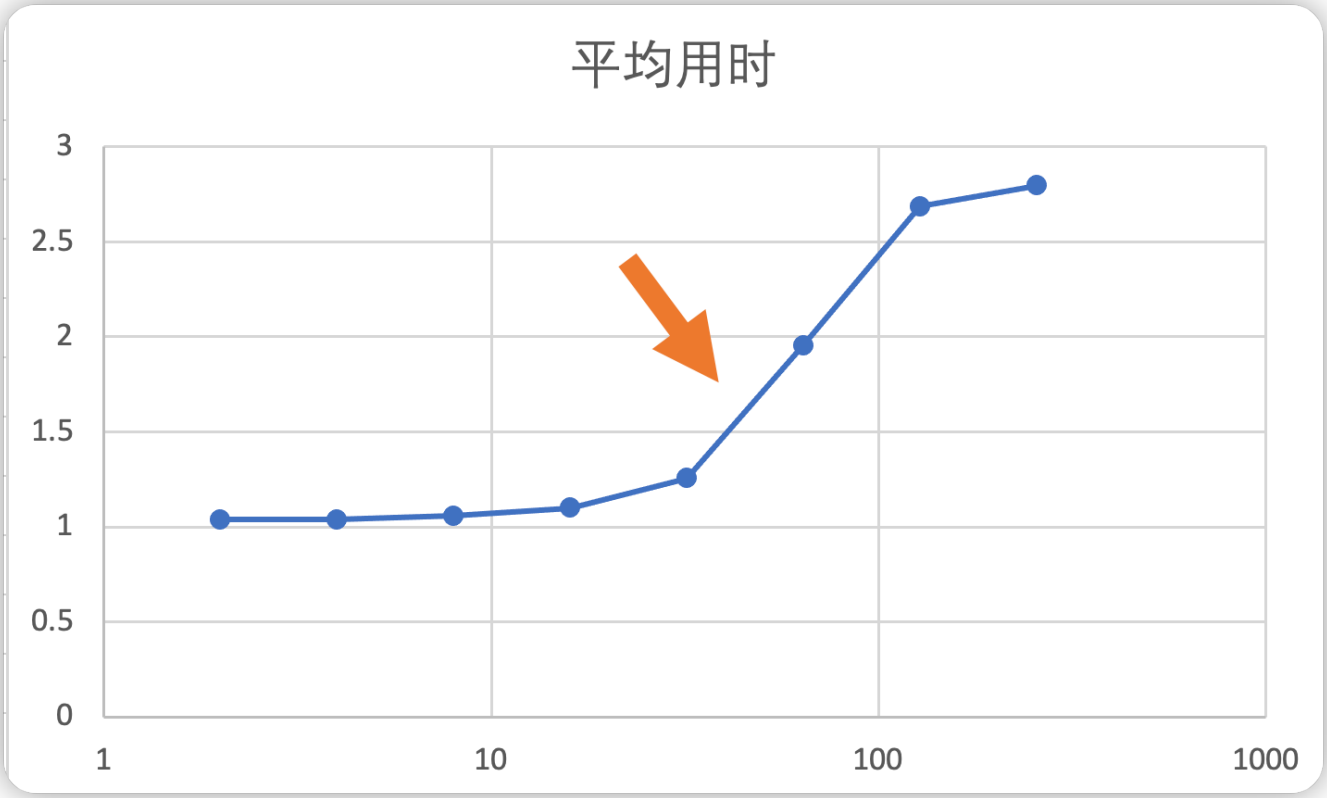




可以看出，当数组大小从`32KB->64KB,256KB->512KB`时，访存用时有一个跳变式的提升。这分别对应于L1缓存失效和L2缓存失效

2. 测量 Cache Line size

- 使用不同步长对大于 L1D Cache Size 的某个数组做相同次数的访问。
- 如果访问是间断的，对数组间隔顺序访问，命中率就会降低，平均访问延迟增大。当间隔达到一定的大小，即超过 Cache Line Size，将造成每次都缺失的最坏情况，平均访问延迟达到最大。



由图可以看出，当跨度`32B->64B`时，用时有一个明显的增大，cache缺失率变大，推测Cache Line大小为64B

3.测量L1D Cache相联度

算法

使用如下算法测量相联度：

- 使用一个 2 倍 Cache Size 大小的数组
- 将数组分为 2^n 块，只访问其中的奇数块
- 逐渐增大 n 的取值，当某一次访问时间变慢时，2^{n-2} 就是相联度

原理剖析

测量的原理是尝试制造“相联度冲突”强制触发CacheLine的驱逐，以此来推算出Cache的相联度。

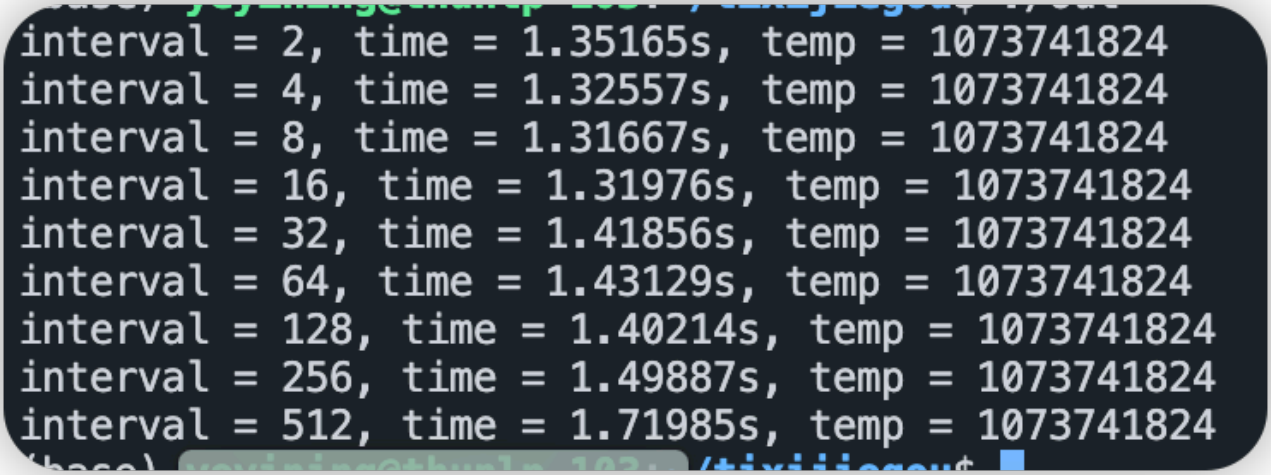
- 假设缓存为空，当访问一块连续的内存时，会在缓存的同一个`Way`中顺序加载。
- 由于一个大小等同于2倍`Cache Size`的数组被分为了2^n块，因此若已知相联度为2的正整数次方，单个块的大小只可能是1个`Way`总大小的2的整数次方倍（可以是负整数，即可以是2倍、1倍、1/2倍、1/4倍等）。
- 当单个块的大小为一个`Way`总大小的整数倍时，一个块会占满整数个`Way`。其后面紧跟着的一个不访问的块同样也会映射满整数个`Way`，再下一个需要访问的块同理，因此这样所有的奇数块会刚好“占满”缓存，不会浪费。
- 但只要n增大到恰好使得一个块的大小只相当于1/2个`Way`的大小。假如第一个块映射到了`Set 3`和`Set 2`，

那么下一个不访问的块会映射到`Set1`和`Set0`，再下一个访问的块又映射到了`Set 3`和`Set 2`。

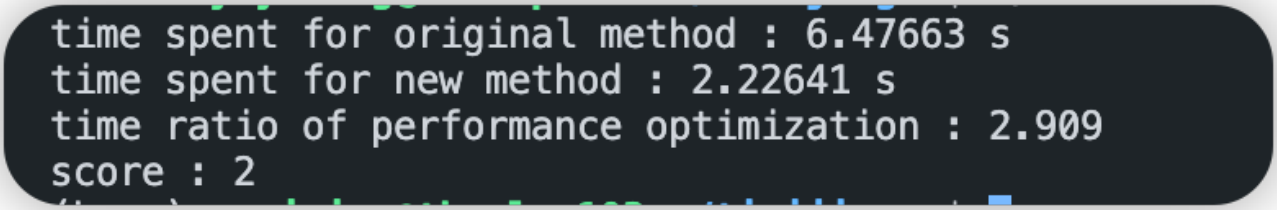
- ……
- 以此类推，所有访问的块都映射到`Set 3`和`Set 2`，所有不访问的块都映射到`Set 1`和`Set 0`。因此有一半的缓存都没有使用，导致缺失和替换的次数显著增加，访问时间明显增加。
- 此时，假设原数组分为了2^n块，缓存大小为S，数组大小为2S，一个块的大小为$\frac{S}{2^{n-1}}$，一个Way的大小为$\frac{S}{2^{n-2}}$，因此相联度为2^{n-2}。

实验

带入`Cache_Size = 32KB`，可以明显看出在分成32块时用时有一个明显的提升，因此推测相联度为8



4.矩阵乘优化



- 首先，将循环顺序改一下，让所有的调用对于最内层是顺序访问，发现速度提升2倍
- 接下来，对于b的调用提到外面，并且改成`register int`，发现变成了3倍
- 实际测试，用不同的`block size`分块计算好像并没有变得更好...不知道为啥

总体代码如下

```
register int ii, jj, kk, temp;
for (ii = 0; ii < MATRIX_SIZE; ii++) {
  for (kk = 0; kk < MATRIX_SIZE; kk++) {
    temp = a[ii][kk];
    for (jj = 0; jj < MATRIX_SIZE; jj++) {
      d[ii][jj] += temp * b[kk][jj];
    }
  }
}
```

> 最牛的是开了 -O3 以后优化了10倍，还是O3厉害(

5.可选部分2

可选部分1 没写.....

测量Cache 替换策略

我们考虑这件事，内存的平均访问时间通过如下方法计算：

$$
\begin{aligned}
T &= \text{hit} \times T_{\text{cache}} + (1 - \text{hit}) \times T_{\text{miss}} \\
&= \text{hit} \times A + B \\
&\propto \text{hit}
\end{aligned}
$$

其中 $T_{\text{cache}}, T_{\text{miss}}$ 对于一个芯片是常数。

这个道理告诉我们如何猜测替换策略：

- 对于一组数，不同的替换策略会导致不同的hit率
- 我们随机生成很多的数组，每一组都计算不同替换策略的hit率，同时统计用时
- 我们最终得到很多个hit率序列，还有一个时间序列
- 进行相关性分析，最相关的序列最可能就是我们猜测正确的hit率序列，对应的策略很可能就是CPU的cache替换策略

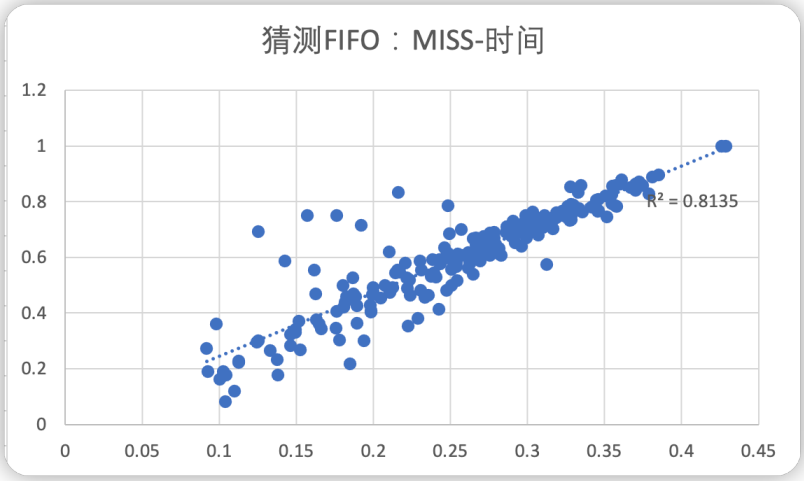
然而，我们该怎么精确控制cache的hit和miss呢？我们用 `posix_memalign` 函数，该函数可以帮助申请一大堆对齐的内存.

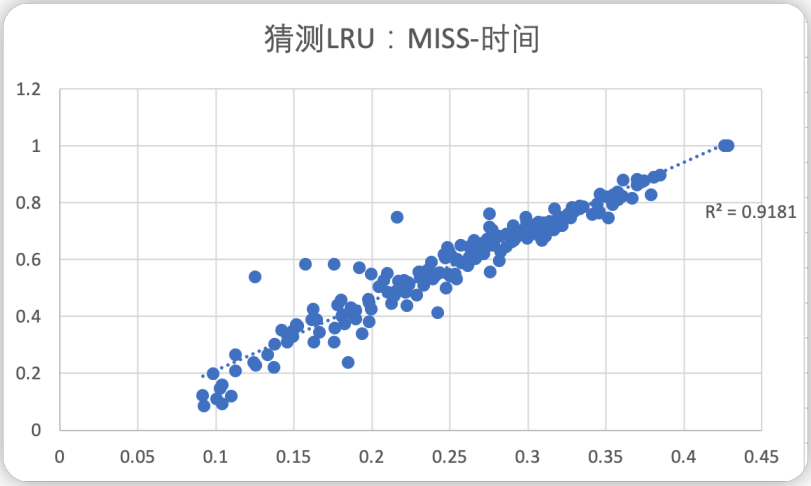
- 我们控制申请一大堆对齐为 `4096KB` 的内存，由于Cache里面的hash函数是低位地址，因此这些内存一定会对应到同一个 `Cache Set` 里
- 我们知道自己CPU的相联度为8，因此访问数组Arr中 `index=0,1,2,3,4,5,6,7` 的内容总能hit，而一旦访问别的，就会触发缺失，同时我们可以通过index的访问历史判断不同策略下具体会替换谁

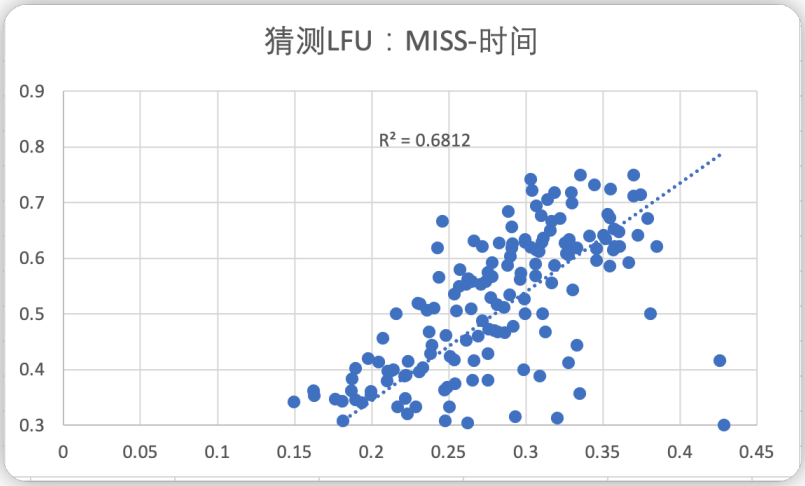
> 换句话说，我们用这种方法可以实时观测cache中的内容具体是什么。我们也可以用这种方法观察cache相联度

我们实现了FIFO类、LRU类、LFU类，对不同的序列模拟命中率，模型替换过程。

接下来我们进行随机模拟，得到猜测hit与实际时间的散点图，并进行拟合，结果如下：







拟合的结果分析上看，LRU的结果最符合线性，推测替换策略为LRU

虽然拟合以后看起来策略的R^2看起来都不高，但这是有道理的：

- 对于一个访问过程来说，每次访问时由于CPU占用等原因可能会有一个偏差P。因此整个序列都会有一个偏差，且每个点的偏差都不同，这取决于当时电脑的运行状况。最终带来拟合效果偏差
- 同时不知道CPU会不会有什么怪优化，怪缓存对结果造成影响...总之看不同策略对比还是挺有区别的.....

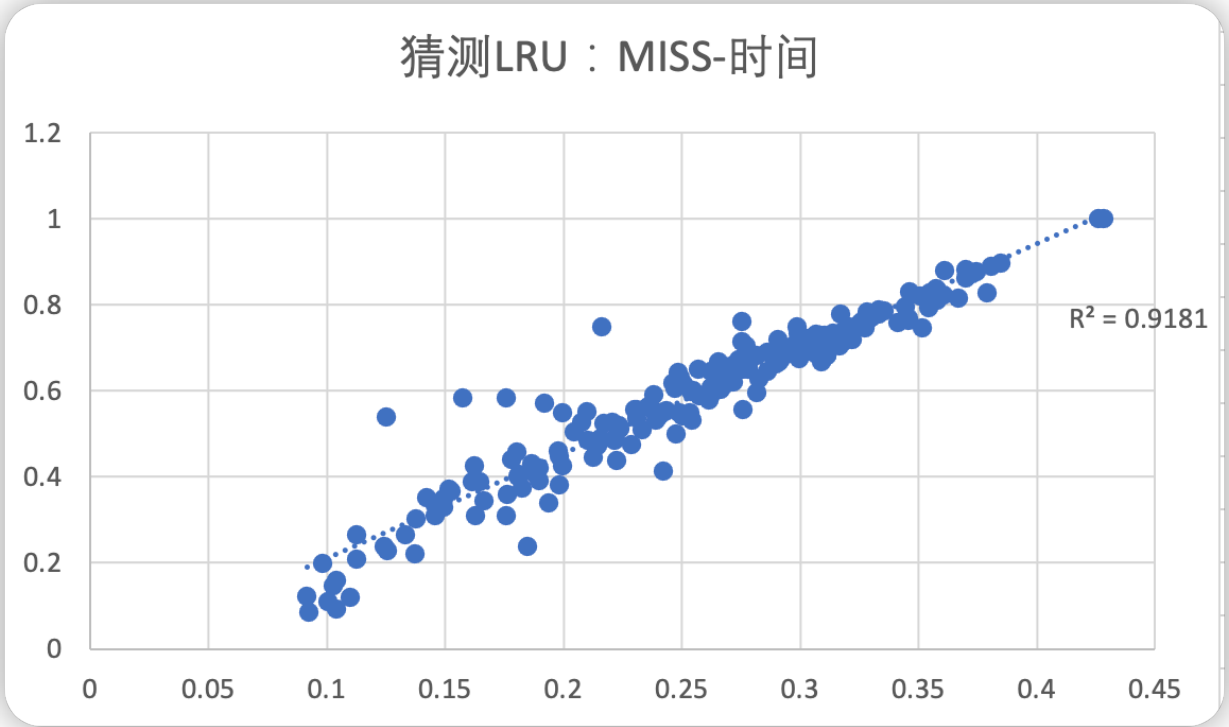
Sequence	Actual	FIFO	LRU	MRU	MRU_N	NRU	PLRU	PLRUl	QLRU_H00_M1_R0_U1	QLRU_H00_M1_R2_U1	QLRU_H11_M1_R0_U0	QLRU_H11_M1_R1_U2	QLRU_H21_M1_R0_U0_UMO	SRRIP
0 0? 1 0? 0? 0? 0? 2 0? 3 4 2? 5 5? 5? 6 0? 7 8 9 10 0? 11 9? 12 10? 9? 13 14 15 16 0? 17 2? 0? 18 19 20 0? 21 22 0? 19? 23 5? 24 25 4? 2? 4? 26	19	18	19	19	19	19	19	19	19	20	19	18	19	20
0 1 1? 0? 2 3 4 0? 2? 5 0? 6 0? 0? 0? 7 0? 0? 4? 8 0? 2? 8? 9 10 0? 9? 0? 11 12 13 0? 14 7? 15 16 17 0? 18 0? 19 20 9? 21 22 4? 23 24 0? 25 26	20	18	20	20	19	18	20	20	21	21	20	20	21	22
0 1 1? 1? 2 1? 1? 1? 1? 0? 3 4 5 6 7 6? 8 9 10 11 6? 6? 12 13 14 15 1? 3? 5? 6? 16 17 18 19 20 12? 21 22 23 24 25 26 1? 27 27? 6? 12? 6? 5? 6? 26?	15	14	15	13	15	15	15	15	15	15	15	15	15	18
0 0? 0? 1 2 0? 0? 0? 3 4 5 6 3? 7 8 3? 0? 9 3? 3? 10 11 12 13 3? 14 15 3? 16 17 18 19 20 2? 21 22 23 12? 24 21? 4? 2? 25 11? 26 4? 27 28 29 10? 22?	15	13	15	15	14	14	15	15	15	15	15	14	15	15
0 0? 1 1? 0? 0? 0? 2 0? 0? 3 4 5 6 7 0? 8 9 10 0? 0? 3? 11 12 13 8? 1? 14 6? 0? 3? 15 0? 9? 16 0? 17 1? 18 19 14? 20 8? 0? 7? 21 21? 22 23 0? 24	18	16	16	16	16	16	16	18	16	18	16	16	17	18
0 1 0? 2 3 3? 4 5 6 3? 7 8 9 3? 10 7? 3? 11 12 7? 3? 13 6? 0? 14 15 3? 4? 16 17 18 19 6? 4? 20 20? 21 3? 16? 22 23 24 22? 3? 25 3? 3? 25? 26 11? 27	17	16	16	16	16	15	17	17	17	17	17	16	16	17
0 1 2 0? 0? 1? 3 3? 4 0? 5 6 0? 7 8 0? 2? 0? 9 10 2? 3? 11 0? 10? 12 0? 5? 0? 13 3? 10? 14 15 16 17 9? 18 19 2? 20 3? 21 22 23 24 25 0? 26 13? 22?	16	15	16	16	16	16	16	18	19	16	15	16	18
0 0? 1 2 3 4 5 6 7 3? 5? 1? 8 9 5? 3? 10 1? 0? 4? 11 12 13 14 15 16 0? 2? 17 1? 18 19 20 15? 21 2? 11? 22 1? 23 9? 24 19? 25 26 27 19? 3? 21? 1? 7?	13	11	10	9	9	9	11	13	11	11	10	10	11	13
0 0? 0? 0? 1 2 3 4 3? 2? 5 6 7 2? 8 6? 9 7? 9? 4? 1? 10 11 12 13 4? 14 3? 15 13? 0? 16 17 18 19 20 21 0? 0? 3? 9? 22 23 0? 10? 16? 10? 22? 8? 24 25	17	15	17	16	16	17	17	17	16	14	15	15	16	15
0 0? 0? 1 2 0? 3 4 5 6 6? 5? 7 8 9 10 11 6? 5? 0? 12 0? 13 14 15 0? 14? 2? 16 17 13? 3? 13? 18 19 9? 0? 20 0? 6? 2? 0? 21 4? 0? 22 23 24 25 26 3?	14	15	15	15	15	16	15	14	15	17	17	15	16	17
0 1 1? 0? 2 3 4 1? 1? 2? 0? 5 6 7 1? 5? 8 9 10 11 8? 0? 6? 8? 12 9? 9? 13 10? 14 15 16 13? 10? 17 18 18? 19 20 21 22 23 24 25 26 27 28 9? 29 9? 1?	16	18	17	14	16	17	18	16	16	16	18	18	18	16
0 1 1? 0? 1? 2 1? 2? 3 4 0? 5 0? 0? 6 1? 7 2? 1? 8 9 10 1? 11 1? 12 13 9? 1? 14 8? 3? 15 16 1? 17 17? 18 1? 15? 19 17? 20 21 22 23 24 6? 5? 1? 17?	20	19	21	20	20	20	20	20	20	21	22	21	23	22
0 1 1? 1? 2 3 1? 1? 4 1? 5 6 5? 7 6? 3? 8 9 10 11 12 7? 13 14 15 16 17 18 19 10? 20 1? 21 19? 9? 5? 22 2? 9? 1? 23 24 17? 25 21? 3? 26 19? 27 3? 1?	12	13	13	13	13	13	13	12	13	13	12	12	12	14
0 1 2 0? 2? 3 4 5 2? 6 1? 5? 7 6? 0? 6? 2? 8 9 10 11 8? 12 7? 13 3? 14 13? 2? 15 16 17 6? 18 14? 19 20 18? 21 22 2? 22? 9? 23 24 25 26 1? 27 28 6?	13	15	14	15	14	15	12	13	13	14	14	14	14	13
0 0? 0? 1 2 3 0? 0? 4 0? 0? 0? 0? 5 6 3? 0? 7 8 4? 9 1? 0? 8? 9? 4? 10 11 12 13 5? 8? 14 3? 15 16 13? 6? 17 18 9? 19 15? 20 21 1? 0? 22 23 22? 15?	19	17	20	18	17	18	19	19	20	20	20	18	19	18
0 0? 1 1? 0? 0? 0? 0? 2 2? 3 0? 4 5 0? 0? 0? 6 7 8 9 1? 5? 0? 10 11 12 13 13? 14 15 1? 16 13? 17 8? 0? 4? 18 19 0? 20 21 22 23 0? 1? 9? 24 25 13?	18	17	17	15	15	16	18	18	19	18	18	17	18	21
0 0? 0? 1 1? 0? 2 3 0? 4 2? 0? 5 6 3? 7 3? 8 9 10 11 1? 12 13 2? 14 15 16 17 18 0? 19 9? 18? 20 20? 1? 21 11? 22 21? 8? 1? 22? 3? 2? 6? 23 23? 20? 24	15	15	15	15	15	15	15	15	16	16	17	17	16	19
0 0? 1 2 3 4 5 6 7 8 9 8? 4? 10 11 12 1? 12? 13 14 0? 15 6? 16 16? 17 18 6? 19 0? 13? 20 20? 4? 21 8? 1? 22 3? 23 0? 24 25 22? 26 27 28 8? 1? 29 30	9	9	9	8	10	9	10	9	10	8	9	9	9	11
0 1 2 0? 3 0? 4 5 2? 2? 6 7 8 9 10 8? 11 4? 5? 6? 4? 10? 12 13 10? 14 0? 15 2? 16 2? 5? 17 18 7? 19 20 21 6? 3? 9? 22 22? 8? 23 24 25 26 27 5? 28	11	13	11	13	13	13	12	11	10	10	10	10	10	10
0 0? 0? 1 2 3 4 5 0? 6 7 8 9 10 0? 11 0? 1? 12 13 14 0? 15 16 17 3? 6? 12? 18 0? 19 14? 12? 20 21 22 0? 3? 23 2? 5? 24 25 8? 0? 26 27 1? 28 29 30	11	7	10	1	9	8	11	11	12	13	9	9	9	10
0 0? 1 0? 0? 2 3 4 5 3? 6 7 7? 0? 3? 0? 8 3? 5? 0? 0? 9 10 11 12 0? 2? 0? 10? 0? 13 0? 14 5? 2? 15 0? 16 0? 0? 17 5? 18 19 15? 20 14? 0? 5? 21 22	24	22	25	23	23	24	23	24	24	24	25	25	24	25
0 1 1? 1? 1? 2 3 0? 4 0? 1? 5 6 7 3? 8 5? 9 10 5? 6? 11 0? 9? 5? 12 13 8? 8? 1? 2? 5? 14 15 16 17 10? 18 19 20 11? 8? 21 3? 22 8? 23 0? 24 25	16	16	16	16	16	16	17	16	15	15	15	16	16	17
0 1 0? 2 0? 3 4 2? 5 0? 5? 2? 6 7 4? 8 9 10 4? 11 12 8? 2? 3? 13 14 15 8? 16 10? 10? 17 0? 18 7? 19 12? 20 21 22 23 0? 24 25 26 27 17? 28 10? 28? 29	13	11	12	11	11	11	13	13	13	13	13	13	13	13
0 0? 1 2 0? 3 3? 3? 0? 4 3? 5 0? 6 7 8 9 10 3? 11 2? 12 13 3? 10? 14 15 16 17 18 0? 3? 19 3? 20 21 3? 3? 22 23 3? 24 25 13? 3? 21? 3? 26 27 13? 3?	18	18	20	20	20	19	18	18	20	20	20	20	20	20
0 1 2 0? 3 0? 4 3? 5 3? 4? 6 6? 7 8 9 10 11 9? 5? 5? 3? 9? 3? 12 13 14 15 0? 16 17 18 4? 19 4? 19? 20 21 5? 10? 14? 18? 22 23 19? 9? 21? 19? 3? 24 19?	16	17	15	16	15	16	16	16	16	17	17	17	17	17
0 0? 1 0? 0? 2 3 4 2? 5 6 0? 7 8 9 10 11 0? 12 13 14 5? 13? 15 4? 16 17 1? 0? 18 0? 19 13? 1? 20 21 5? 15? 9? 5? 0? 17? 17? 22 9? 0? 8? 23 24 19? 25	14	12	13	13	13	14	14	14	15	15	15	13	14	16
0 0? 0? 0? 0? 1 0? 0? 0? 2 0? 3 4 5 2? 6 2? 2? 7 8 9 2? 10 11 12 0? 0? 0? 2? 2? 13 14 15 0? 10? 16 17 10? 2? 18 8? 19 20 21 22 23 0? 24 25 26 25?	21	19	21	20	20	21	21	21	21	22	22	21	22	23
0 1 0? 2 1? 3 1? 3? 0? 0? 2? 0? 4 5 0? 0? 6 7 0? 8 9 7? 8? 10 9? 0? 0? 0? 11 0? 12 13 4? 14 13? 0? 3? 15 0? 1? 16 17 0? 18 0? 19 0? 20 7? 0? 0?	26	24	26	25	25	25	26	26	26	26	26	26	26	26
0 0? 1 0? 0? 0? 0? 2 3 4 5 0? 6 7 1? 8 9 8? 8? 3? 0? 7? 10 11 12 12? 13 14 3? 0? 6? 12? 14? 12? 15 16 0? 17 18 10? 1? 19 20 21 22 15? 23 23? 18? 24 2?	19	17	19	20	20	19	18	19	17	21	18	18	18	18
0 1 2 0? 2? 0? 3 2? 0? 0? 4 5 6 0? 2? 5? 2? 7 0? 8 5? 3? 8? 9 10 2? 2? 1? 0? 1? 11 12 8? 13 14 0? 15 1? 5? 3? 0? 16 14? 17 18 0? 1? 19 1? 20 0?	26	23	27	23	24	24	25	26	26	25	27	27	25	29
0 1 2 0? 3 4 5 0? 6 0? 2? 0? 0? 7 3? 8 7? 9 10 11 1? 7? 0? 12 13 5? 0? 14 15 7? 11? 16 0? 17 18 19 20 21 22 23 0? 11? 24 13? 25 3? 26 27 16? 28 29	14	11	14	14	14	12	14	14	15	16	13	13	12	14

- 查了一下资料，Intel的CPU主要是用 Tree-PLRU策略替换，本身的序列和LRU也有所不同，大体上和LRU一致，这应该也能解释为什么拟合效果差一些.....

测量Cache是否写直达

- 写直达策略在写命中和写不命中时的访问延迟相似
- 写回策略在写命中时的访问延迟明显小于写不命中

关键在于怎么控制写命中和写不命中，由上面的图我们可以很容易得到：MISS高与MISS低即为写命中和写不命中的平均水平



用时差距`0.1-0.45`还是很大的，最终推测写策略为写回

6.对本次实验的意见和建议。

无