实验二 实现 Cache 模拟器

PB13011079 杨智

实验目的

1. 加深对Cache的基本概念、基本组织结构以及基本工作原理的理解；
2. 掌握Cache容量、相联度、块大小对Cache性能的影响；
3. 掌握降低Cache不命中率的各种方法以及这些方法对提高Cache性能的好处；
4. 理解LRU与随机法的基本思想以及它们对Cache性能的影响。

实验环境

操作系统：　mac OSX
IDE：　IntelliJ 2016.1.1
编程语言：　JAVA

注意事项：

要先设置好Cache再读入地址流文件，再执行（如果更新了Cache设置，则要重新读入地址流文件）

实现要求

设计与实现一个 Cache 模拟器，能模拟处理器中 Cache 的行为。处理器访存有三种类型：读指令、读数据和写数据，给出访存的地址和类型，我们的 Cache 的模拟器能够进行模拟这种带有 Cache 的访存行为，并能给出统计信息，如访存次数、Cache 命中次数、命中率等。

1. 基本要求：模拟器中必须具备下列配置项
    a. 能够设置 Cache 总的大小

   b. 能够设置 Cache 块的大小
   c. 能够设置Cache 的映射机制：直接映射、n-路组相联
   d. 能够设置 Cache 的替换策略：LRU、FIFO …
   e. 能够设置 Cache 的写策略：写回法、写直达法
2. 较高要求：模拟器中可以选择支持下列配置
   a. 能够设置将 Cache 分为数据 Cache 和 指令 Cache
   b. 能够设置预取策略
   c. 能够设置写不命中的调块策略
   d. 有友好的操作界面，如使用界面来配置 Cache
3. 完成之后，需要用你的模拟器分析以下问题：
   a. Cache 容量对不命中率的影响
   b. Cache 采取的映射机制对不命中率的影响
   c. Cache 块大小对不命中率的影响
   d. Cache 替换算法对不命中率的影响
4. 实验需要提交的内容包括：
   a. 实验源代码
   b. 实验最终的可执行文件
   c. 实验报告（包括设计思想、实验分析结论等）

实现的要求

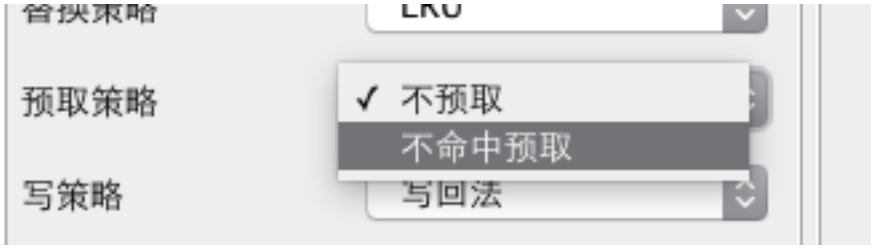
我写的Cache Simulator实现了基本要求的a、b、c、d、e，和较高要求的a、b、c、d

基本要求：

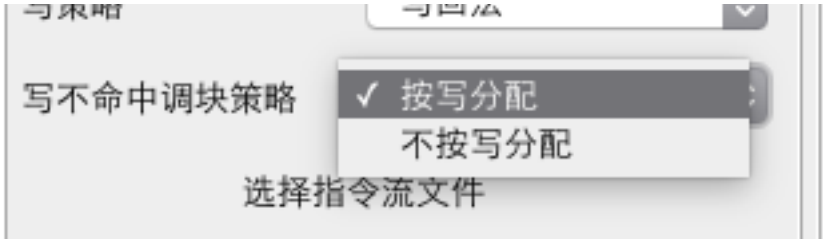
略

较高要求a：设置指令Cache和数据Cache

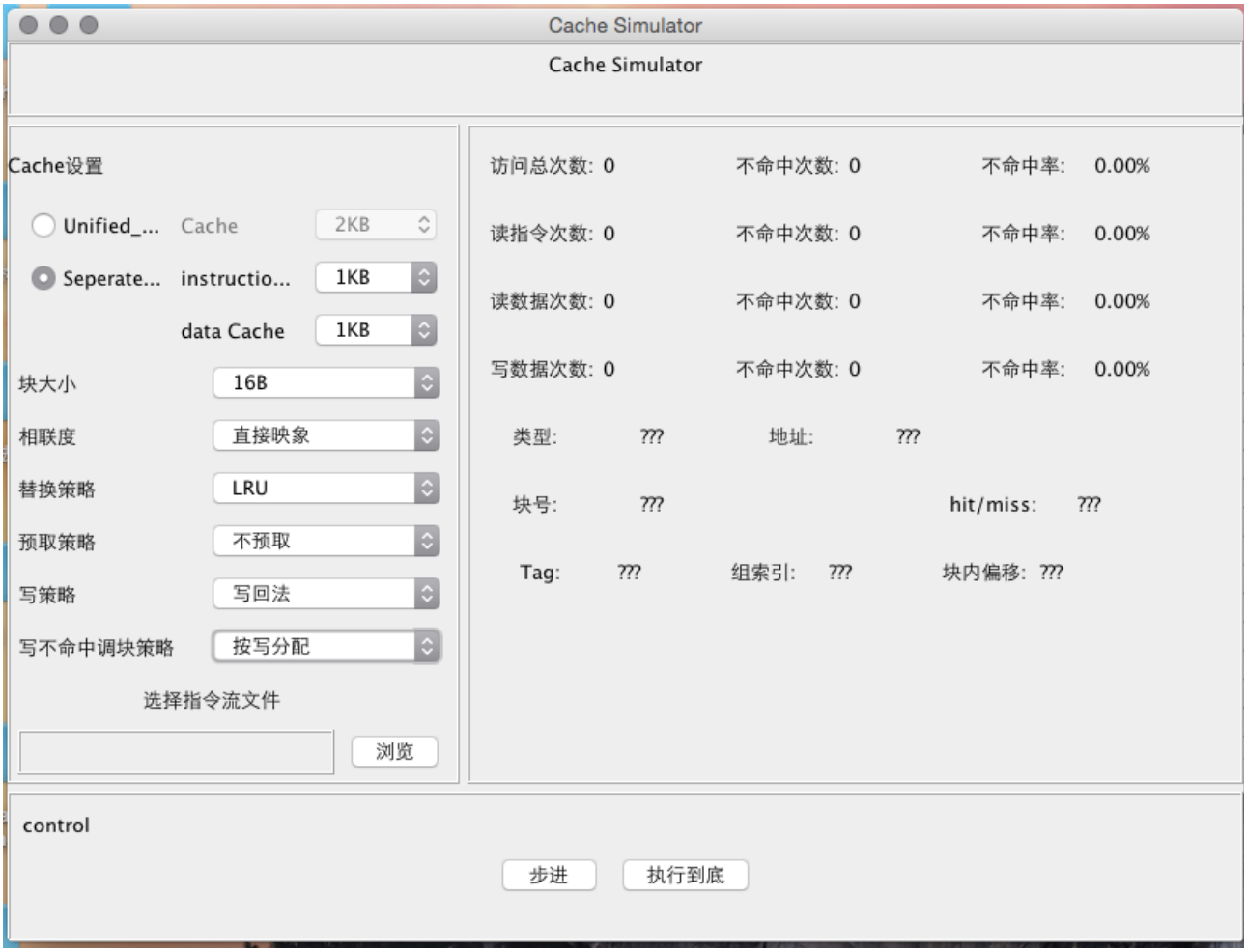
较高要求b：能够设置预取策略



较高要求c：能够设置写不命中的调块策略



较高要求d：有友好的操作界面，如使用界面来配置 Cache



使用模拟器分析问题

Cache 容量对不命中率的影响

设置如下：

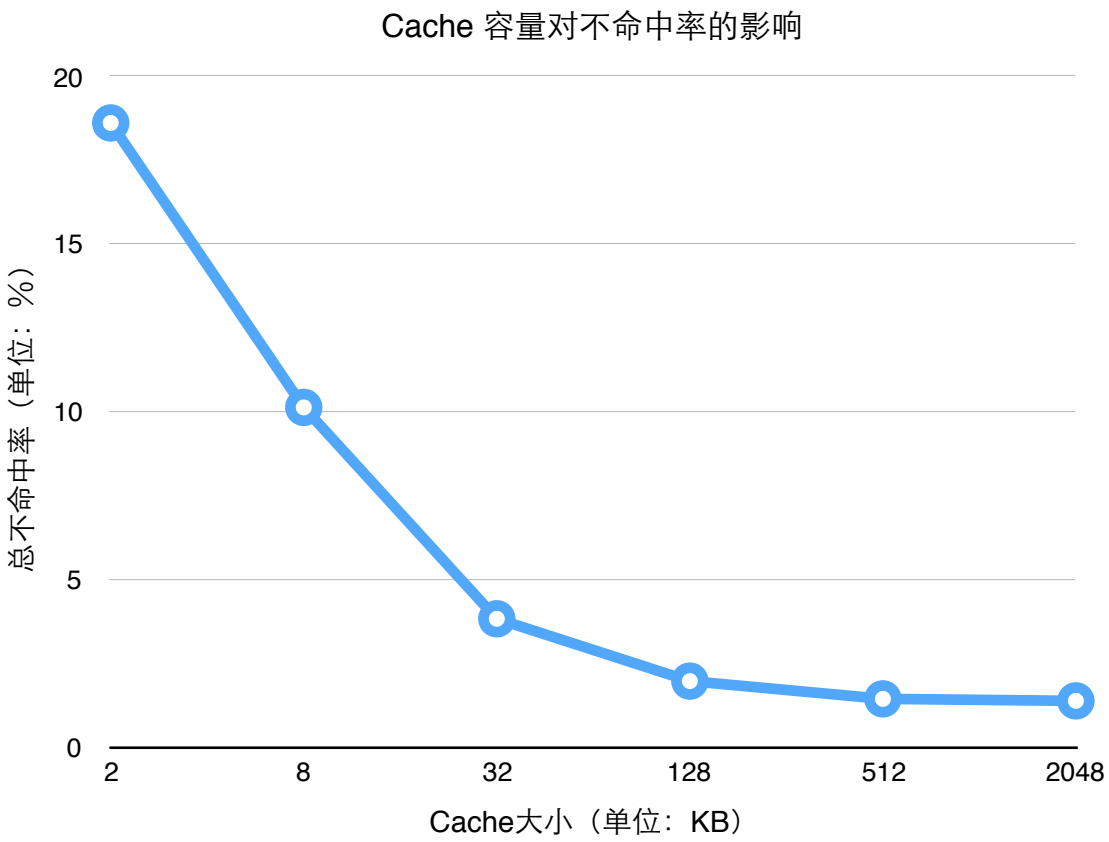
基本设置	
Cache大小	（变量）
块大小	16B
相联度	直接映象
替换策略	LRU
预取策略	不预取
写策略	写回法
写不命中策略	按写分配

基本设置	
指令流文件	cc1.bin

结果如下：

Cache大小	总不命中率
2KB	18.61%
8KB	10.12%
32KB	3.81%
128KB	1.95%
512KB	1.42%
2048KB	1.36%

图表：



结论：
Cache的容量越大，命中率越高，不命中率越低；不命中率随Cache大小增大而减小（不考虑抖动）。当Cache容量大到一定程度之后，对不命中率的影响就慢慢变小。

Cache 采取的映射机制对不命中率的影响

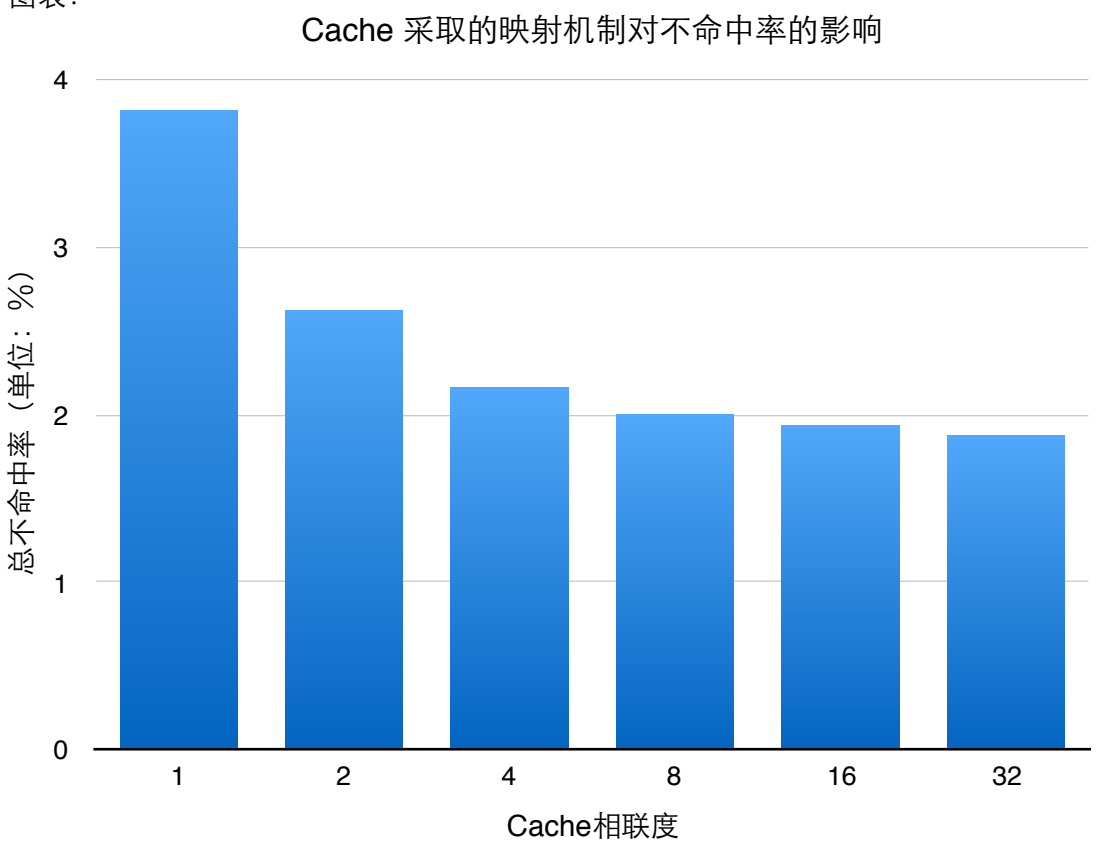
设置如下：

基本设置	
Cache大小	32KB
块大小	16B
相联度	（变量）
替换策略	LRU
预取策略	不预取
写策略	写回法
写不命中策略	按写分配
指令流文件	cc1.bin

结果如下：

相联度	总不命中率
（直接映射）1	3.81%
2	2.63%
4	2.16%
8	2.00%
16	1.93%
32	1.87%

图表：



结论：
Cache的相联度越高，命中率越高，不命中率越低；不命中率随Cache相联度增大而减小（不考虑抖动）。当Cache相联度大到一定程度之后，对不命中率的影响就慢慢变小。

Cache 块大小对不命中率的影响

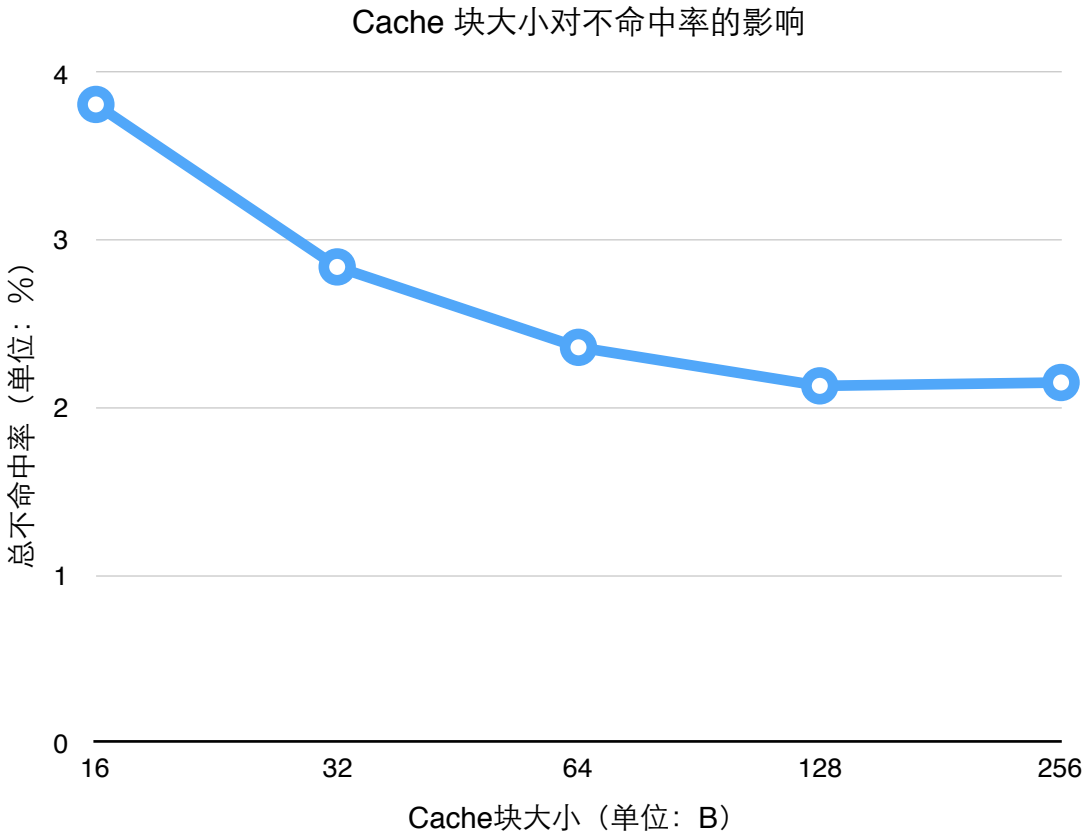
设置如下：

基本设置	
Cache大小	32KB
块大小	（变量）
相联度	直接映射
替换策略	LRU
预取策略	不预取
写策略	写回法
写不命中策略	按写分配
指令流文件	cc1.bin

结果如下：

块大小	总不命中率
16B	3.81%
32B	2.84%
64B	2.36%
128B	2.13%
256B	2.15%

图表：



结论：
Cache的块大小越大，命中率越高，不命中率越低；不命中率随Cache块大小增大而减小（不考虑抖动）。但是，当Cache块大小大到一定程度之后，对不命中率的影响就慢慢变小了。在某些情况下，甚至会出现块大小变大、而总不命中率稍微上升的情况。

Cache 替换算法对不命中率的影响

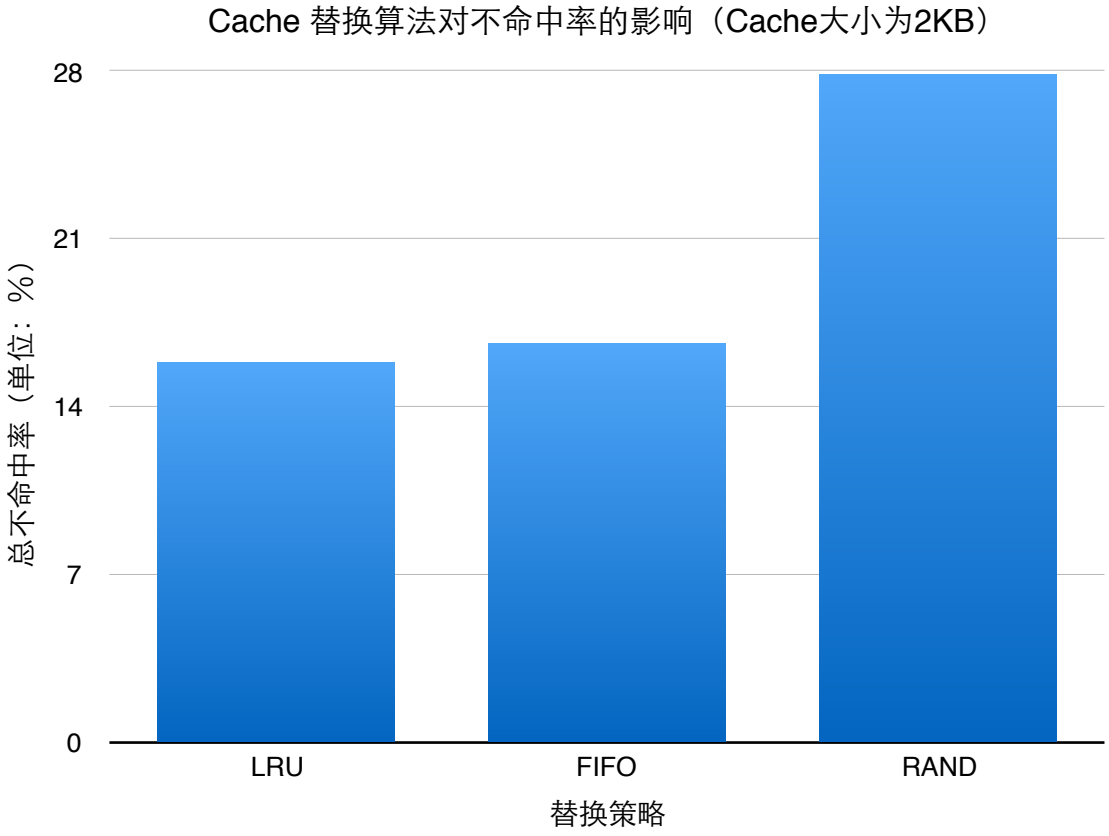
设置如下：

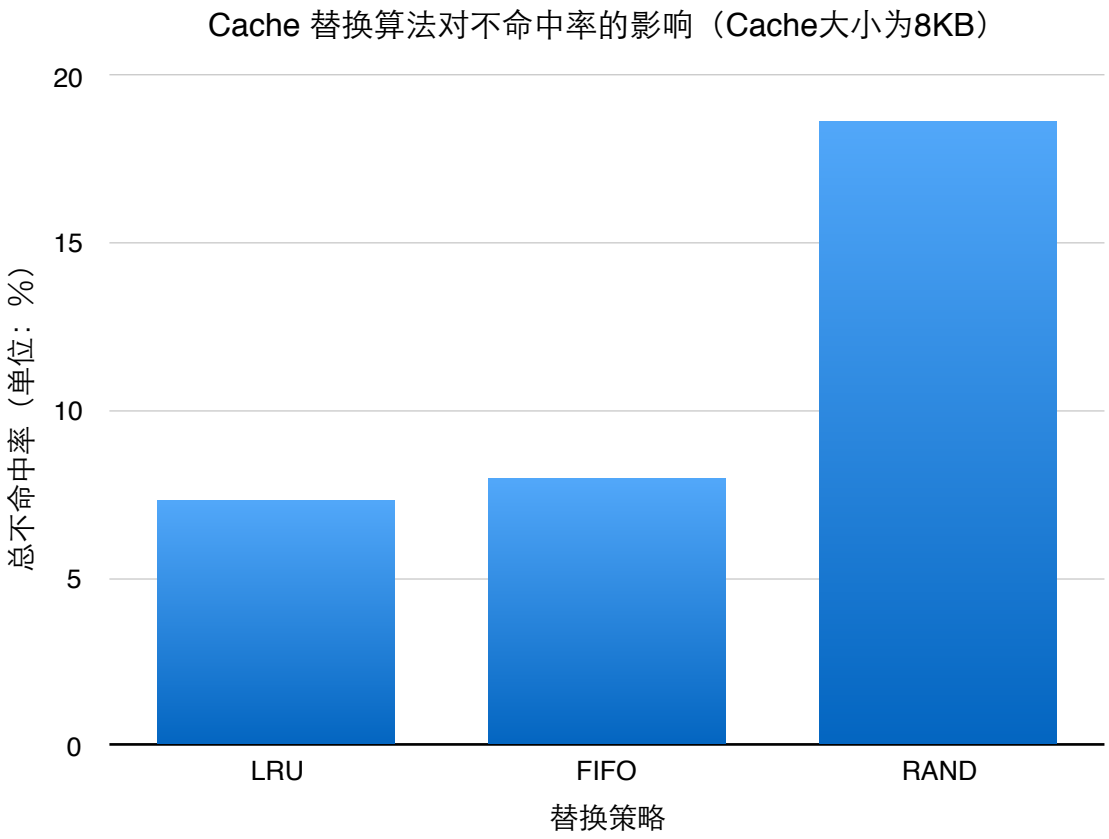
基本设置	
Cache大小	（辅变量）
块大小	16B
相联度	4
替换策略	（主变量）
预取策略	不预取
写策略	写回法
写不命中策略	按写分配
指令流文件	cc1.bin

结果如下：

替换策略	总不命中率（Cache大小为2KB）	总不命中率（Cache大小为8KB）
LRU	15.85%	7.33%
FIFO	16.62%	7.92%
RAND	27.87%	18.61%

图表：





结论：
大多数情况下，RAND替换策略明显劣于LRU和FIFO策略，而LRU策略略优于FIFO策略。原因是LRU很好地利用了局部性原理，FIFO也略符合局部性原理，而RAND完全不考虑局部性。担当Cache足够大时，这三者的差距会越来越小

补充1：Cache 预取策略对不命中率的影响

设置如下：

基本设置	
Cache大小	8KB
块大小	16B
相联度	4
替换策略	LRU
预取策略	（变量）
写策略	写回法
写不命中策略	按写分配
指令流文件	cc1.bin

结果如下：

预取策略	总不命中率
不预取	7.33%
不命中预取	8.72%

结论：
从上面的结果来看
“不命中预取” 劣于 “不预取”

补充2：Cache 写不命中调块策略对不命中率的影响

设置如下：

基本设置	
Cache大小	2KB
块大小	16B
相联度	1
替换策略	LRU
预取策略	不预取
写策略	写回法
写不命中策略	（变量）
指令流文件	cc1.bin

结果如下：

写不命中调块策略	总不命中率
按写分配	18.61%

写不命中调块策略	总不命中率
不按写分配	21.67%

结论：
“按写分配” 优于 “不按写分配”

设计思想

把CacheBlock用一个类来表示

```
private class CacheBlock {
    int tag;
    boolean dirty;
    long count;
    long time;

    public CacheBlock(int tag) {
        this.tag = tag;
        this.dirty = false;
        this.count = -1L;
        this.time = -1L;
    }
}
```

把Cache用一个类来表示

```
private class Cache {

    private CacheBlock cache[][];
    private int        cacheSize;
    private int        blockSize;
    private int        blockNum;
    private int        blockOffset;
    private int        blockNumInAGroup;
    private int        groupNum;
    private int        groupOffset;

    private long FIFO_order[];
    private long LRU_order[];

    public Cache(int cacheSize, int blockSize) {
        ……
    }

    public boolean read(int tag, int index, int ins_offset) {
        ……
    }
```

```
    public boolean write(int tag, int index, int ins_offset) {
      ……
    }

    public void prefetch(int nextins_block) {
      ……
    }

    public void replaceCacheBlock(int tag, int index) {
      ……
    }

    private void loadToCache(int tag, int index, int groupAddr) {
      ……
    }
  }
```

在打开地址流文件的时候，先对指令进行预处理，如把指令分解为如下的几个域

address form = [ins_block ins_offset] = [tag index offset]

并且对读入的指令进行分析

开始运行之前，先根据**Cache**的设置，对**Cache**实例化

```
public CCacheSim(){
    super("Cache Simulator");
    fileChooser = new JFileChooser();
    fileChooser.setFileFilter(new DinFileFilter());
    draw();
  }

  private void initCache() {
    read_data_hit = read_data_miss = 0;
    read_ins_hit = read_ins_miss = 0;
    write_data_miss = write_data_hit = 0;
    write_mem_total = 0;

    if (now_cacheType == 0) {
      united_Cache = new Cache(2 * 1024 * pow(4, now_csIndex), 16 * pow(2, now_bsIndex));
      ins_Cache = null;
      data_Cache = null;

    }
    else if (now_cacheType == 1) {
      united_Cache = null;
      ins_Cache = new Cache(1 * 1024 * pow(4, now_icsIndex), 16 * pow(2, now_bsIndex));
      data_Cache = new Cache(1 * 1024 * pow(4, now_dcsIndex), 16 * pow(2, now_bsIndex));
    }
  }
```

这是单步执行的代码，根据指令的类型执行相应的操作

```
private void simExecStep(boolean oneStepExec) {
    ip %= ins_size;
    if (ip == 0) {
        initCache();
        reloadUI();
    }
    int ins_kind = instructions[ip].ins_kind;
    int index = instructions[ip].index;
    int tag = instructions[ip].tag;
    int ins_offset = instructions[ip].ins_offset;

    boolean isHit = false;
    if (now_cacheType == 0) {
        if (ins_kind == 0) {
            isHit = united_Cache.read(tag, index, ins_offset);
            if (isHit)
                read_data_hit++;
            else {
                read_data_miss++;
                united_Cache.replaceCacheBlock(tag, index);
            }
        }
        else if (ins_kind == 1) {
            isHit = united_Cache.write(tag, index, ins_offset);
            if (isHit)
                write_data_hit++;
            else {
                write_data_miss++;
                if (now_allocIndex == 0) {
                    united_Cache.replaceCacheBlock(tag, index);
                    united_Cache.write(tag, index, ins_offset);
                } else if (now_allocIndex == 1)
                    write_mem_total++;
            }
        }
        else if (ins_kind == 2) {
            isHit = united_Cache.read(tag, index, ins_offset);
            if (isHit)
                read_ins_hit++;
            else {
                read_ins_miss++;
                united_Cache.replaceCacheBlock(tag, index);
                if (now_prefetchIndex == 1)
                    united_Cache.prefetch(instructions[ip].ins_block + 1);
            }
        }
    }
    else if (now_cacheType == 1) {
        if (ins_kind == 0) {
            isHit = data_Cache.read(tag, index, ins_offset);
            if (isHit)
                read_data_hit++;
            else {
```

```
                read_data_miss++;
                data_Cache.replaceCacheBlock(tag, index);
            }
        }
        else if (ins_kind == 1) {
            isHit = data_Cache.write(tag, index, ins_offset);
            if (isHit)
                write_data_hit++;
            else {
                write_data_miss++;
                if (now_allocIndex == 0) {
                    data_Cache.replaceCacheBlock(tag, index);
                    data_Cache.write(tag, index, ins_offset);
                }
                else if (now_allocIndex == 1)
                    write_mem_total++;
            }
        }
        else if (ins_kind == 2) {
            isHit = ins_Cache.read(tag, index, ins_offset);
            if (isHit)
                read_ins_hit++;
            else {
                read_ins_miss++;
                ins_Cache.replaceCacheBlock(tag, index);
                if (now_prefetchIndex == 1)
                    ins_Cache.prefetch(instructions[ip].ins_block + 1);
            }
        }
    }

    if (oneStepExec || ip == ins_size - 1)
        statisticUIUpdate(instructions[ip], isHit);

    ip++;
}
```

实验总结

Cache的工作原理是基于程序访问的局部性的。
在一个较短的时间间隔内，由程序产生的地址往往集中在存储器逻辑地址空间的很小范围内。指令地址的分布本来就是连续的，再加上循环程序段和子程序段要重复执行多次。因此，对这些地址的访问就自

然地具有时间上集中分布的倾向。 这种对局部范围的存储器地址频繁访问，而对此范围以外的地址则访问甚少的现象，就称为程序访问的局部性。

根据程序的局部性原理，可以在主存和CPU通用寄存器之间设置一个高速的容量相对较小的存储器，把正在执行的指令地址附近的一部分指令或数据从主存调入这个存储器，供CPU在一段时间内使用。这对提高程序的运行速度有很大的作用。这个介于主存和CPU之间的高速小容量存储器称作高速缓冲存储器(Cache)。