Cache Analysis 实验报告

实验人：陈熠豪

学号：2017011486

邮箱：chenyiha17@mails.tsinghua.edu.cn

代码实现

- 内存管理：./src/mem.{cpp,h}
  - Mem 类用作内存管理，以 byte 为粒度实现最小内存占用
  - 初始化时确定 numBlock（block 数量）、blockBitWidth（block 位宽度），分配内存
  - 可通过成员函数访问和修改某一 block 或 block 中的某一位
  - 提供了一个测试函数进行随机读写，确保实现的正确性
  - 使用该类，在 Cache 类中维护 cache line 元数据、lru 数据和 tree 数据
- Cache 模拟：./src/cache.{cpp,h}
  - Cache 类模拟 cache 行为，建立相应大小的 cache line 元数据，维护 lru 或 tree 标记进行替换
  - lru 和 tree 均采用上课介绍的方法，基于 Mem 类使用最小内存实现
  - purge 成员函数为 cache 替换的总控函数
- 主函数：./src/main.cpp
  - 实现 trace 输入、解析， cache 模拟、输出和错误处理

实验

- 如何复现
  - 将 trace 文件置于 ./trace/，运行以下命令。该脚本命令会基于 make 进行构建，并进行三个必需的实验：不同 Cache 布局、不同替换策略、不同写策略

    ```
    ./run.sh
    ```

    访问 Log 将存于 ./result/ 目录下，命名格式为

    ```
    {trace_name}_{block_bit_width}_{way_bit_width}_{replace_algorithm}_{write_assign}_{write_strategy}.output

    trace_name:      trace 的名字，比如 astar
    block_bit_width:  cache block 的位宽，比如 block 大小为 8B，则为 3
    way_bit_width:    cache 路数的位宽，比如 8 路组关联则为 3。特殊的，直接映射为 DM，全关联为 FA
    ```

```
replace_algorithm： 替换算法，如lru、rand、tree
write_assign：    写分配 flag，1 为写分配，0 为写不分配
write_strategy：  写策略 flag，1 为写回，0 为写直达
```

这一步结果的备份存于 ./result/output.zip

- 在获得上一步结果后，运行以下命令，得到本次报告中使用的分析图。需要 Python3.7 及以上和必要的 package

```
./analyze.py
```

结果将存于 ./result/analyze.png 和 ./result/summary.csv，相同的备份已经在文件目录里了

- 结果与分析
  - **注意：以下的图表中，block、way 分别以其大小的位宽来表示，比如 block 为 3 时，表示 block 的大小为 8B。特别的，way 的值为 0 时表示直接映射，为 inf 时表示全相联。此外，assignment 和 write 分别为写分配、写策略的 flag，assignment 为 1 时表示写分配，为 0 时表示写不分配，write 为 1 时表示写回，为 0 时表示写直达**
  - 在固定替换策略（LRU），固定写策略（写分配+写回）的前提下，尝试不同的 Cache 布局

    block 为 8B 的结果如下

trace	block	way	algorithm	assignment	write	miss	hit	miss_ratio
astar	3	0	lru	1	1	116693	384775	0.232703
astar	3	2	lru	1	1	116737	384731	0.232791
astar	3	3	lru	1	1	116766	384702	0.232848
astar	3	inf	lru	1	1	116640	384828	0.232597
bzip2	3	0	lru	1	1	6654	537860	0.012220
bzip2	3	2	lru	1	1	6627	537887	0.012170
bzip2	3	3	lru	1	1	6627	537887	0.012170
bzip2	3	inf	lru	1	1	6627	537887	0.012170
mcf	3	0	lru	1	1	23238	484462	0.045771
mcf	3	2	lru	1	1	23232	484468	0.045759
mcf	3	3	lru	1	1	23232	484468	0.045759
mcf	3	inf	lru	1	1	23232	484468	0.045759
perlbench	3	0	lru	1	1	15640	491801	0.030821
perlbench	3	2	lru	1	1	10510	496931	0.020712
perlbench	3	3	lru	1	1	9084	498357	0.017902
perlbench	3	inf	lru	1	1	8901	498540	0.017541

block 为 32B 的结果如下

trace	block	way	algorithm	assignment	write	miss	hit	miss_ratio
astar	5	0	lru	1	1	48451	453017	0.096618
astar	5	2	lru	1	1	48291	453177	0.096299
astar	5	3	lru	1	1	48279	453189	0.096275
astar	5	inf	lru	1	1	48111	453357	0.095940
bzip2	5	0	lru	1	1	1686	542828	0.003096
bzip2	5	2	lru	1	1	1668	542846	0.003063
bzip2	5	3	lru	1	1	1668	542846	0.003063
bzip2	5	inf	lru	1	1	1668	542846	0.003063
mcf	5	0	lru	1	1	9273	498427	0.018265
mcf	5	2	lru	1	1	9263	498437	0.018245
mcf	5	3	lru	1	1	9263	498437	0.018245
mcf	5	inf	lru	1	1	9263	498437	0.018245
perlbench	5	0	lru	1	1	8253	499188	0.016264
perlbench	5	2	lru	1	1	5763	501678	0.011357
perlbench	5	3	lru	1	1	4172	503269	0.008222
perlbench	5	inf	lru	1	1	3350	504091	0.006602

block 为 64B 的结果如下

trace	block	way	algorithm	assignment	write	miss	hit	miss_ratio
astar	6	0	lru	1	1	25341	476127	0.050534
astar	6	2	lru	1	1	25123	476345	0.050099
astar	6	3	lru	1	1	25074	476394	0.050001
astar	6	inf	lru	1	1	24908	476560	0.049670
bzip2	6	0	lru	1	1	868	543646	0.001594
bzip2	6	2	lru	1	1	841	543673	0.001544
bzip2	6	3	lru	1	1	841	543673	0.001544
bzip2	6	inf	lru	1	1	841	543673	0.001544
mcf	6	0	lru	1	1	5515	502185	0.010863
mcf	6	2	lru	1	1	5502	502198	0.010837
mcf	6	3	lru	1	1	5502	502198	0.010837
mcf	6	inf	lru	1	1	5502	502198	0.010837

trace	block	way	algorithm	assignment	write	miss	hit	miss_ratio
perlbench	6	0	lru	1	1	5765	501676	0.011361
perlbench	6	2	lru	1	1	4329	503112	0.008531
perlbench	6	3	lru	1	1	3169	504272	0.006245
perlbench	6	inf	lru	1	1	1966	505475	0.003874

上述三组数据的对比图对应于分析图中的前三个子图

从关联数的角度分析，在所有 block 大小下，各个 trace 的 miss ratio 均随着关联数的增大而降低，但除了 perlbench 呈现出明显的的下降趋势，其余不太明显

从 block 大小的角度分析，在所有关联数下，各个 trace 的 miss ratio 均随着 block 的增大而明显降低

这样的结果符合对 cache 的理论分析：一定程度下，更高的关联度和更大的 block 可以有效提高 cache 命中率

在元数据空间开销上，元数据开销 = $2^{ts-bs} \times (2 + 64 - ts + nw)$ bits，其中 ts 为 cache 大小的位宽，bs 为 block 大小的位宽，nw 为关联数的位宽。在 cache 大小不变的情况下，更小的 block 和更高的关联数会增大元数据空间开销，且 block 大小位宽的影响为指数的，而关联数位宽的影响是线性的

- 在固定 Cache 布局（块大小 8B，8-way 组关联），固定写策略（写分配+写回）的前提下，尝试不同的 Cache 替换策略

trace	block	way	algorithm	assignment	write	miss	hit	miss_ratio
astar	3	3	lru	1	1	116766	384702	0.232848
astar	3	3	rand	1	1	116524	384944	0.232366
astar	3	3	tree	1	1	117307	384161	0.233927
bzip2	3	3	lru	1	1	6627	537887	0.012170
bzip2	3	3	rand	1	1	6627	537887	0.012170
bzip2	3	3	tree	1	1	6627	537887	0.012170
mcf	3	3	lru	1	1	23232	484468	0.045759
mcf	3	3	rand	1	1	23353	484347	0.045998
mcf	3	3	tree	1	1	24060	483640	0.047390
perlbench	3	3	lru	1	1	9084	498357	0.017902
perlbench	3	3	rand	1	1	9110	498331	0.017953
perlbench	3	3	tree	1	1	9165	498276	0.018061

这组数据的对比图对应于分析图中的第四个子图

可以发现，在使用的几个 trace 中，使用不同的替换算法在 miss ratio 上虽有差别，但并不明显。这一方面和 trace 的访问模式相关，另一方面，bzip2 并没有出现替换，perlbench 的替换次数也极少（350 左右），这也导致实验结果不太能反映出差异

在占用空间上

- rand 不占用额外空间
- tree 占用空间 = $2^{ts-bs-nw} \times (2^{nw} - 1)$ bits
- lru 占用空间 = $2^{ts-bs-nw} \times 2^{nw} \times nw = 2^{ts-bs} \times nw$ bits

其中 ts 为 cache 大小的位宽，bs 为 block 大小的位宽，nw 为关联数的位宽。显然，lru 会占用最大的空间，tree 会占用一定空间，而 rand 则完全不占用空间

在替换时，lru 会寻找该组内序号最大的一路的 block 替换，并更新所有路的序号；tree 会依照树节点标记找到叶子节点对应的 block 替换，并更新路径上的节点标记；rand 则随机选取一路进行替换

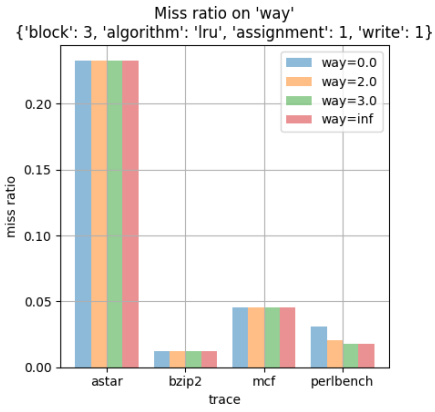
- 在固定 Cache 布局（块大小 8B，8-way 组关联），固定替换策略（LRU）的前提下，尝试不同的写策略

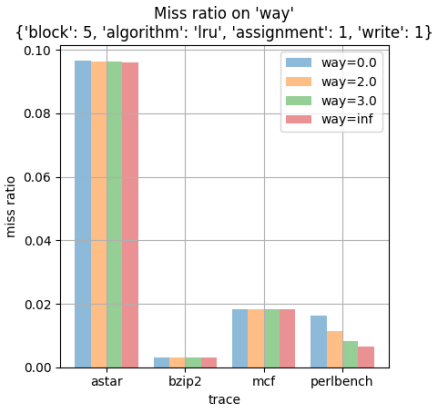
trace	block	way	algorithm	assignment	write	miss	hit	miss_ratio
astar	3	3	lru	0	0	173001	328467	0.344989
astar	3	3	lru	0	1	173001	328467	0.344989
astar	3	3	lru	1	0	116766	384702	0.232848
astar	3	3	lru	1	1	116766	384702	0.232848
bzip2	3	3	lru	0	0	47209	497305	0.086699
bzip2	3	3	lru	0	1	47209	497305	0.086699
bzip2	3	3	lru	1	0	6627	537887	0.012170
bzip2	3	3	lru	1	1	6627	537887	0.012170
mcf	3	3	lru	0	0	56593	451107	0.111469
mcf	3	3	lru	0	1	56593	451107	0.111469
mcf	3	3	lru	1	0	23232	484468	0.045759
mcf	3	3	lru	1	1	23232	484468	0.045759
perlbench	3	3	lru	0	0	23664	483777	0.046634
perlbench	3	3	lru	0	1	23664	483777	0.046634
perlbench	3	3	lru	1	0	9084	498357	0.017902
perlbench	3	3	lru	1	1	9084	498357	0.017902

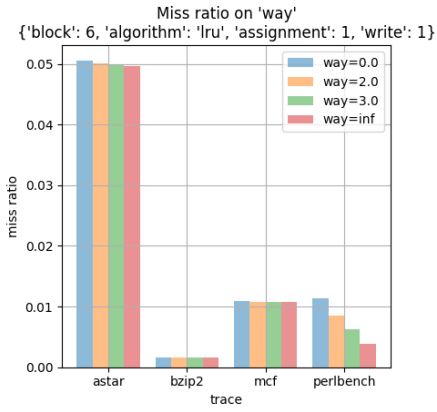
这组数据的对比图对应于分析图中的第五个子图

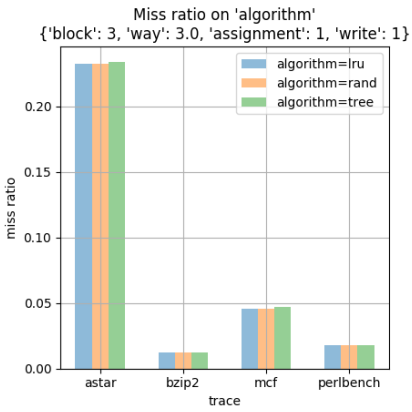
需要说明的是，写回和写直达在实现中并没有作实质性的区分，因为本次实验只是针对 cache 的模拟实验，如果要考虑写回和写直达的区别，则不可避免地涉及到多级 cache 、写入 buffer 和内存等更多的方面，太过复杂，所以未做考虑，因而写回和写直达的结果是一样的

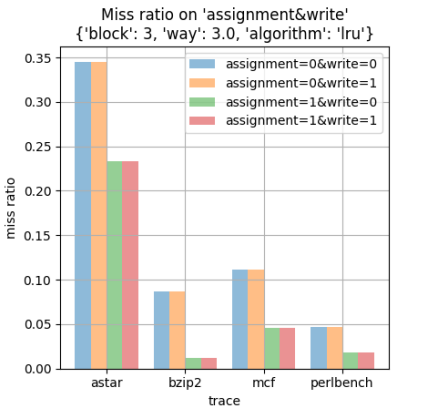
可以发现，使用写分配可以明显降低各个 trace 的 miss ratio，这是因为这些 trace 中均存在大量的局部写操作，故写分配能够有效利用

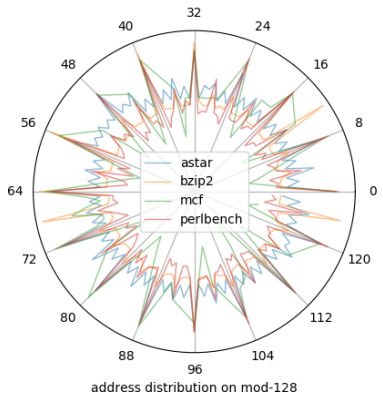


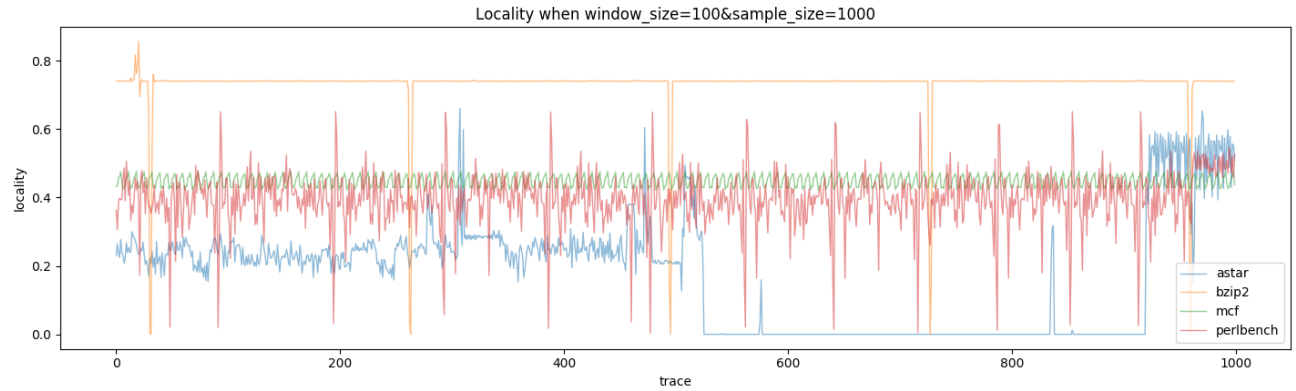












- trace 静态分析
  - astar、bzip2、mcf 和 perlbench 的访问地址分布如分析图的第六个子图所示。该图对所有访问地址进行 mod 128 后进行统计计数，以圆周表示循环地址空间，以半径表示计数大小的对数处理结果。该图可展示访问序列的空间局部性，可以看出在各个 trace 中，访问的地址确实存在一定的访问模式，如 perlbench 频繁访问以 8 对齐的地址，而相对的，astar 的访问空间则十分均匀，这种访问差异或许也可以解释它们的 miss ratio 之间的差异（astar 的 miss ratio 显著大于 perlbench 的 miss ratio）。
  - astar、bzip2、mcf 和 perlbench 的访问序列局部性如分析图的第七个子图所示。该图对各个 trace 的访问序列以滑动平均的方法计算局部性指标，具体的算法如下：
    - 对原始 trace，计算连续滑动窗口内的不同地址数量，形成新的序列 trace'
    - 对 trace'，计算连续滑动平均（卷积平滑处理），并做 normalize 后取剩余百分比，形成新的序列 locality
    - 对 locality 进行等距采样，得到最终结果 locality'，并呈现

当滑动窗口长度为 100，采样数为 1000 时，结果如图所示。该图可以展示访问序列的时间局部性，locality 值越接近 1 则该处的局部性越强。从中可以看出，bzip2 的局部性最强，而 astar 局部性弱且不稳定，这种差异和它们的 miss ratio 有着显然的相关性（bzip2 的 miss ratio 最低，而 astar 的 miss ratio 最高）。
