Cache 模拟器

李炳睿 2018013391

2020-04-02

目录

代码结构

整体代码基本分为如下几个部分.

- `config` 主要定义了一些常量, 为 Cache 的初始化以及运行提供帮助;
- `cache` 中定义了 Cache 类来模拟 Cache 的行为, 主要对外提供 `access` 方法, 表示一次对 Cache 的访问.
- `policy` 中定义了多种替换策略, 本次实验中主要实现了 LRU, 二叉树, 以及 SCORE 方法, 二叉树提供的服务较为全面, 而 LRU 和 SCORE 目前只能支持块大小为 8, 8 路组相连的 Cache 模拟.
- `utils` 中提供了一些 helper function
- `main` 中主要负责 tokenize 工作, 以及读写 Cache 的总逻辑.

实验结果与分析

Cache 布局

缺失率分析

在固定替换策略 (二叉树替换), 固定写策略 (写分配 + 写回) 的前提下, 不同 Cache 布局下的缺失率 (单位:1%) 与元数据 (单位:byte) 如下所示:

trace1	8B	32B	64B	trace2	8B	32B	64B
direct	4.94	2.20	1.46	direct	2.06	1.33	1.59
4-way	4.58	1.82	1.08	4-way	1.22	0.306	0.154
8-way	4.58	1.82	1.08	8-way	1.22	0.306	0.154
full	4.58	1.82	1.08	full	1.22	0.306	0.154

trace3	8B	32B	64B	trace4	8B	32B	64B
direct	23.40	9.84	5.27	direct	3.67	2.31	1.89
4-way	23.28	9.63	5.01	4-way	2.05	1.09	0.819
8-way	23.29	9.63	5.00	8-way	1.78	0.800	0.591
full	23.26	9.60	4.98	full	1.75	0.660	0.387

- 从主要趋势来看, 在固定块大小的条件下, 随着路数的增多 (直接映射可以看作 1 路, 而全相联可看作拥有最大路数), 缺失率逐渐减小; 在固定路数的情况下, 随着 Cache Line 大小的增加, 缺失率逐渐减小
- 除去主要的趋势之外, 我们也能有一些其他的发现; 随着路数的增加, 缺失率虽然再降低, 但是趋于收敛, 在 trace1/2 上体现得尤其明显, 在 4-way 之后, 缺失率就不再受路数影响了; 此外, 也出现了个别的例外, 在 trace3、块大小为 8, 路数由 4 路变为 8 路的实验中, 缺失率出现上升, 猜测可能是 index 位数减小导致了无法充分利用局部性的缘故.
- 随着块大小的增加, 缺失率降低的尤为明显, 只有一个例外, 在 trace2、直接映射, 块大小由 32B 变为 64B 时, 缺失率出现上升; 但这并不影响我们从上述表格中得出直接结论, 增加块大小比增加路数对缺失率降低的影响更大; 但事实上, 这个结论并不能够推广, 其一是不同程序拥有不同的特点, 这 4 个 trace 无法代表所有程序, 其二, 我们实验设置也并不充分, 事实上随着缓存行大小的增加, 缺失率先减小后增大;
- 此外, 在软件模拟的过程中, 更多是一种功能性的验证, 往往不能有更实际的体验; 例如, 在全相联的组织方式中, 运行时间很长, 但在硬件环境下的比对是并行执行的; 再例如, 缓存行大小增加时, 虽然替换次数减少, 但是每次替换的成本都会提高, 在模拟中并不能感受到这些特点.

元数据分析

metadata	8B	32B	64B
direct	102400	25600	12800
4-way	106496	26624	13312
8-way	108544	27136	13568
full	131072	31744	15616

- 在元数据方面, 我们发现随着路数增多, 元数据的存储量增大, 而随着缓存行大小的增大, 元数据的开销减小
- 上述现象发生的原因在于, 经过计算, 我们发现元数据的开销只与缓存行的数量和 tag 的位数有关; 而在同一缓存行大小的条件下, 二叉树所带来的元数据开销是相同的;

- 因此, 当缓存行大小不变, 路数增大时, 由于 tag 位增多 (线性增加), 开销增大; 当组织方式不变, 缓存行大小增加时, 缓存行的数量减少 (呈指数减少), 开销显著下降.

替换策略

缺失率与元数据分析

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, 不同替换策略下的缺失率 (单位:1%) 与元数据 (单位:byte) 如下所示:

	trace1	trace2	trace3	trace4	meta
LRU	4.58	1.22	23.285	1.790	112640
BT	4.58	1.22	23.287	1.783	108544
SCORE	4.58	1.22	23.292	1.779	118784

- 就目前的测例而言, 三种方法对缺失率的影响并不大, 差别仅在小数点后两位, 因此体现在缺失次数上就是几十次的差别
- 在元数据方面, 由于三次实验仅在替换策略上有差别, 因此元数据开销的差异也只体现在替换策略上; 设有 n 路组相连, 那么在一个组中, BT 需要 n 个 bit, LRU 需要 $n \log n$ 个, 而 SCORE 需要 $n \cdot d$ 个 bit, 其中 d 可由人为决定, 本次实验中选取了 6 位, 因此开销是最大的; 而从缓存行本身的开销来看, 不同的策略不会带来差别.

执行动作差异分析

- 对于 BT 策略, 在每一次访问发生后, 我们都要更改二叉树节点的值, 即从根到达该叶节点的路上, 对实际的方向做出反转; 而在选取替换块时, 则从根节点开始顺着节点的值的方向找到唯一的叶节点, 这就是将要被替换的节点.
- 对于 LRU 策略, 我们维护了一个栈, 当一次命中发生后, 我们找到栈中该元素并将到重新放回栈顶; 当一次缺失/替换发生后, 选择位于栈低的元素作为被替换的缓存行.
- 实际上二叉树策略是一种近似的 LRU 策略, 但是由于 LRU 是由过去

预测未来, 因此不能保证替换的选择是最优的, 这依赖于程序本身的特性, 所以这两种方法均无法一致地优于另一种方法.

- 对于 SCORE 策略, 事实上是 LRU 策略的一种抽象, 它会给一组中的每个缓存行一个得分; 当数据初次被装入缓存行, 会拿到一个初始分; 当某一行被命中后, 他的得分会增加一个定值; 当某一行被命中或装入时, 其他行的得分会减少一个定值; 此外, 我们预先设定了一个阈值, 当要选择被替换的缓存行时, 找到所以得分低于该阈值的缓存行, 并从中随机选择, 如果没有低于该阈值的缓存行, 则选择得分最低的那一行.

写策略

缺失率与元数据分析

在固定 Cache 布局 (块大小 8B，8-way 组关联)，固定替换策略 (二叉树替换算法) 的前提下, 不同写策略下的缺失率 (单位:1%) 与元数据 (单位:byte) 如下所示:

	trace1	trace2	trace3	trace4	meta
写回写分配	4.58	1.22	23.29	1.78	108544
写回写不分配	11.15	8.67	34.66	4.67	108544
写直达写分配	4.58	1.22	23.29	1.78	106496
写直达写不分配	11.15	8.67	34.66	4.67	106496

- 在缺失率方面, 写回与写直达对缺失率没有影响, 这是因为他们在 Cache 的装入和替换动作上没有差别;
- 在缺失率方面, 写分配与写不分配的缺失率有较大差别, 写分配策略能够明显降低缺失率;
- 在元数据方面, 写回比写直达的元数据开销要大, 这是因为写回策略使得缓存行中多了 dirty 位;
- 在元数据方面, 写分配与写不分配对元数据开销没有影响.

执行动作差异分析

- 对于写回和写直达而言, 在元数据方面, 写回要比写直达多了一个 dirty 位, 每当进行写 Cache 操作时将 dirty 位置为 1, 因此开销略

有增加, 但是在缺失率方面两种策略没有差别; 相比写直达而言, 写回带来的更多是速度上的加快, 但在此次模拟中并不能体会到.

- 对于写分配和写不分配而言, 元数据开销方面没有区别, 但是从实验数据中来看, 写分配要比写不分配的缺失率低, 这是因为在写缺失发生时, 写分配会把数据装入缓存行, 而写不分配策略不会; 但是, 在特殊的程序上也有可能出现相反的情况.

运行方式

编译方式: 在 src 目录下运行

```
g++ --std=c++11 cache.cpp policy.cpp utils.cpp main.cpp -o main
```

运行方式:

```
./main
Usage: ./main [src_trace]
Usage: ./main [src_trace] [rp] [w0] [w1]
Usage: ./main [src_trace] [rp] [w0] [w1] [dst]
```

例如

```
./main ../test_trace/1.trace BT back alloc ../out/dst.log
```

- 此种运行方式可以设置 Cache 模拟器的替换策略、写策略以及源 trace 文件
- 若要修改 Cache 布局, 请在 `config.h` 中修改 `WAY_NUM` 和 `LINE_SIZE` 的值, 在第 28 与 29 行
- 提醒: 请按照文档中所要求的配置进行测试, 对于文档中未要求的配置, 不保证得到正确结果.

集成测试: 在 src/下运行

```
python run.py [n]
```

- 其中 n 可以取值 3/4/5, 分别对应文档中实验流程 3/4/5 中的要求
- 提醒: 在使用此种方式前, 请确保所有源文件未被改动