CacheSimulator 报告

计 71 钟闰鑫 2017010306

April 2020

1 CacheSimulator 架构设计

CacheSimulator 主要由如下几部分组成

- cache　组成部件，包括 replace_policy，way_metadata 等
- file_parser　对 trace 文件进行处理
- test　各部件的单元测试
- bench　具体实验代码
- 其他　定义了通用的数据结构，包括 Bitset，PerfStats(用于收集访问时数据) 等，位于 include 文件夹中

在整个 CacheSimulator 的设计中，主要以 oop 为目标，让整个代码更加具有可读性。

由于有元数据数组开销的限制，为了便于统一管理，使用了 Bitset 来作为所有元数据的存储器，因为这个 Bitset 是基于 char 数组实现的，其元数据所需的位数会尽可能接近最少。

具体编译，使用方法见主目录下 README.md，访问 log 见 log 文件夹。

2 替换策略

替换策略的采用了多态的方式实现,基类位于 cache/include/replace_policy.h。

实现了 LRU, Binary Tree, Rand, FIFO, LIP(LRU Insertion Policy)[1], BIP(Bimodal Insertion Policy)[1], Score[2]

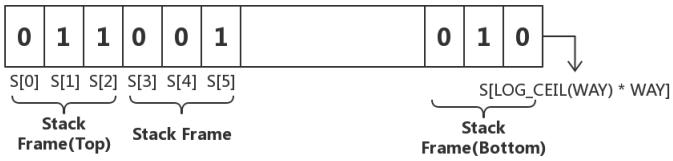

图 1: LRU Stack(WAY = 3)

2.1 LRU 系列

2.1.1 LRU

LRU 采用 Bitset 作为内部栈的实现，对于一个有 W 路相连的 cache，栈中一个栈帧需要表示一个 cache 块，因此一个栈帧需要 $\log(W)$ 个 bit，而一个组内共有 W 路，因此总共需要 $W\log(W)$ 个 bit，如图1。

这里将栈顶作为 MRU 位 (Most Recently Used)，栈底作为 LRU 位 (Least Recently Used)。当访问一个已存在的 block 时，对应的栈帧会被放置到栈顶，其余栈帧依次下压；当 cache 已满时，LRU 替换算法会将栈底栈帧对应的 cache block 作为 victim 置换出去，并将其放置于栈顶，其他栈帧同样依次下压。

2.1.2 FIFO

FIFO 继承于 LRU，只是在访问与替换时有所不同。访问一个 block 时，不会对栈做任何操作，而当需要替换块时，FIFO 会将栈底的栈帧对应的 block 作为 victim 置换，然后放到栈顶。每次替换栈底，放进栈顶，就做到了先进先出。

2.1.3 LIP 与 BIP

LIP 与 BIP 都来源于 Qureshi 的论文 [1]，其基本思想就是基于 LRU 构建一个低 overhead 但有比 LRU 更好效果的替换算法。

LIP 与 LRU 唯一不同点在于 LRU 在替换的时候会将新的 block 放在栈顶 (MRU 位)，而 LIP 则是将新的 block 放在栈底 (LRU 位)，只有当该 block 在 LRU 位被再次访问时，此 block 才会被放入栈顶 (MRU 位)。这个

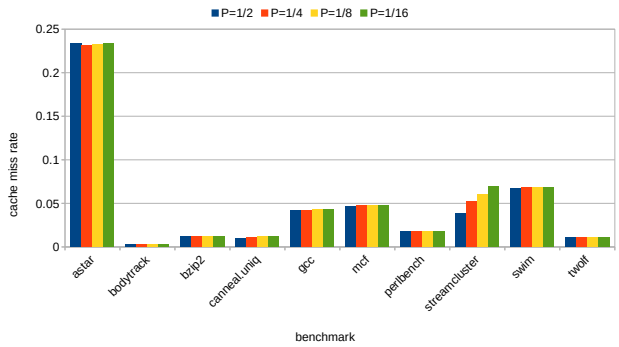

图 2: BIP，有 P 的概率新块被放到 MRU 位

替换算法主要是为了解决当工作集合远大于 cache 容量，或者 cache 经常被污染时 LRU 的 cache 命中率不好的问题。例如，对于 cache 被污染的情况，LIP 可以利用首先把新块放到 LRU 位，来使得污染源能快速从 cache 中排出，从而保证良好的 cache 命中率。

而 BIP 则是结合了 LIP 与 LRU，考虑到确实有相当一部分程序是 LRU-friendly 的，对于这种情况，LIP 的性能就不如 LRU 了，因此 BIP 通过一个预先设定的概率 P，每次替换的时候进行随机选择替换 MRU 位或者 LRU 位的块。

- 由于 BIP 涉及到一个参数 P，这里对该参数进行初步实验以找出最好参数，便于在之后的综合实验中进行比较。这里 cache 设置为 128KB 大小，块大小为 8B，8 路组相连，写回与写分配。

  可以看到，在大多数的 benchmark 中，$P = \frac{1}{2}$ 有着最好的 cache 命中率，因此最终测试选择 $P = \frac{1}{2}$。由于 P 越大，越容易把新块放到 MRU 位置，也就越近似于 LRU 算法，这也侧面说明这些 benchmark 对于 LRU 都比较友好。

2.2 Binary Tree

Binary Tree 替换算法是对 LRU 算法的近似，采用一颗二叉树来进行访问情况的保存，如图3

二叉树内部节点取值 (0 或 1) 来指示更久未被访问的子树，在对某个 block 进行访问后，会沿着其对应叶子节点到根节点的路径，将对应的内部

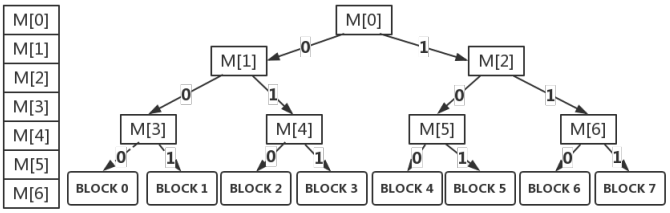

图 3: Binary Tree(WAY = 8)

节点指向另一个子树；当需要替换块时，从根节点向下搜索达到的叶子节点即为对应的 victim。

可以看到，元数据开销只有二叉树的内部节点，对于一个每组 W 路的 cache，每组只需要 $W-1$ 的 bit 开销

2.3 Rand

随机替换算法十分简单，在需要替换块时，则随机选择一个块作为 victim，注意到这样 (在不考虑随机算法的 overheaed) 是不需要任何额外的元数据。

2.4 Score

Score 替换算法来源与 Duong 的论文 [2]。其基本思想是通过每个块的"Score" 来判断将哪个块作为 victim 替换出去。其中包括初始 score 设定，动态 score 更新，victim 阈值下随机选择等。

- 初始 score 设定

  Score 算法在运行时利用过去一段时间的命中率进行动态初始 score 更新，而最开始的初始 score 需要预先设置，这里对此进行实验，如图4，可以看到初始值对于命中率的影响

- threshold 设定

  Score 算法在寻找被替换的块时会判断所有块的 score,对于小于 threshold 的 block 进行随机选取，若所有 block 的 score 都大于 threshold，

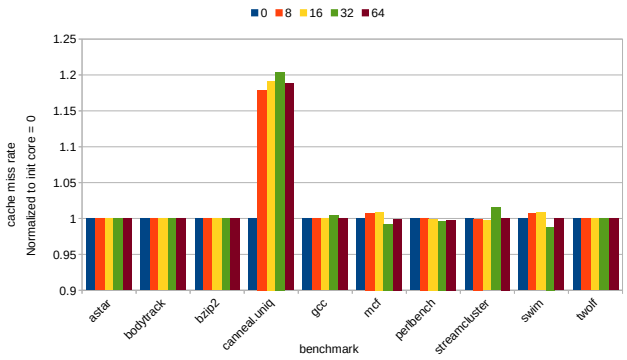

图 4: 初始 score 对缺失率的影响 (归一化到 init score = 0)

则选取 score 最小的那个 block，下面实验 threshold 对命中率的影响，如图5，可以看到当 threshold 为 48 时在各个 benchmark 上都有着最低的缺失率。

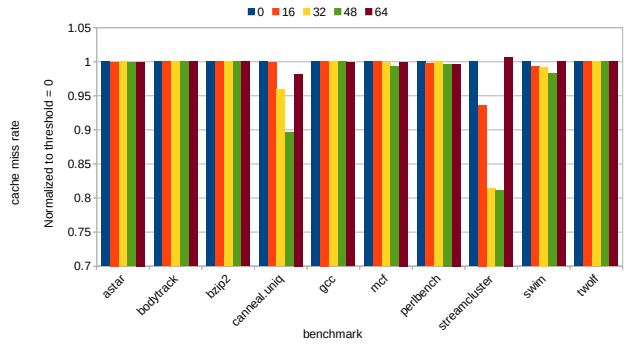

图 5: threshold 对缺失率的影响

2.5 综合实验

2.5.1 元数据开销

下面对各个替换算法元数据开销进行分析，对于每组有着 W 路的 cache

- LRU 系列算法 (LRU, LIP, FIFO, BIP)，其元数据只有栈结构的开销，为 $W\log(W)$

- Binary Tree 算法需要 $W-1$ bit 开销
- Score 算法需要很多元数据，但大多数元数据都是可调的。记 score 需要 bit 数为 S，由于每个 block 都需要一个 score，则整个组需要 WS 个 bit，并且还需要 S bit 来存初始 score 值，1bit 来存初始 score 更新的方向，除此之外还需要存 threshold 与过去一段时间 cache miss 的数目，这部分所需 bit 数记为 T，综合算的 Score 算法的元数据开销为 $(W+1)S+T+1$ bit
- Rand 算法在除去随机数生成的开销外没有其他元数据开销

2.5.2 各替换算法比较

考虑 128KB 的 cache, 块大小为 8B，8 路组相连，使用写回与写分配，下面对各种替换算法进行分析

	LRU	Rand	Binary Tree	FIFO	LIP	BIP	Score
astar	0.232848	0.232274	0.232862	0.232182	0.232946	0.23313	0.232268
bodytrack	0.002482	0.002482	0.002482	0.002482	0.002482	0.002482	0.002482
bzip2	0.01217	0.01217	0.01217	0.01217	0.01217	0.01217	0.01217
canneal.uniq	0.007644	0.00828	0.007668	0.008067	0.009573	0.009604	0.008371
gcc	0.040979	0.041105	0.04095	0.041312	0.041828	0.041698	0.041145
mcf	0.045759	0.045968	0.045759	0.045988	0.046234	0.046571	0.045996
perlbench	0.017902	0.017951	0.017833	0.018004	0.018244	0.018053	0.017973
streamcluster	0.02382	0.030246	0.024778	0.025882	0.039644	0.037842	0.029885
swim	0.065368	0.06581	0.065361	0.06546	0.066568	0.067004	0.06579
twolf	0.011402	0.011402	0.011402	0.011402	0.011402	0.011402	0.011402

表 1: 替换算法对 cache 命中率的影响

从表1和图6可以看到，在这些 benchmark 中，LRU 替换算法综合效果是最好的，而 Binary Tree 替换算法紧随其后，其相比与 LRU 算法差距非常小。其他算法，则在某些 benchmark 上效果很差。

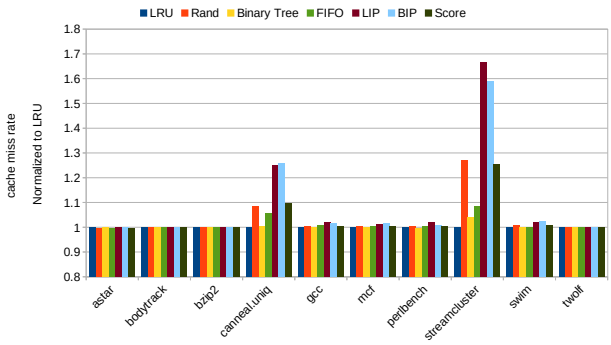

图 6: 替换算法对 cache 缺失率的影响 (Normalized)

3 Cache 布局

考虑 128KB 的 cache，使用 LRU 替换算法，写回与写分配，下面分析不同 cache 布局对于缺失率的影响，所有数据见表2与表3

	8B_direct	32B_direct	64B_direct	8B_all	32B_all	64B_all
astar	0.233961	0.0983772	0.0526813	0.232597	0.0959403	0.0496702
bodytrack	0.00254693	0.00254693	0.00254693	0.00248153	0.00248153	0.00248153
bzip2	0.0206147	0.013311	0.0158967	0.0121705	0.00306328	0.0015445
canneal.uniq	0.0107284	0.0107284	0.0107284	0.00749657	0.00749657	0.00764959
gcc	0.0423671	0.0133939	0.00849165	0.0408895	0.0118736	0.00673088
mcf	0.0494465	0.0219677	0.0145952	0.0457593	0.018245	0.0108371
perlbench	0.0366683	0.0231377	0.0189401	0.017541	0.00660175	0.00387434
streamcluster	0.04342	0.04342	0.04342	0.00851587	0.0128326	0.0205322
swim	0.0657502	0.0215902	0.0122166	0.0653577	0.0211087	0.0114646
twolf	0.011847	0.00387719	0.0027132	0.0114017	0.00340083	0.0019738

表 2: 不同布局对缺失率的影响

3.1 元数据开销

除去替换算法，写策略等的元数据开销，对于一个 cache 块来说，需要一个 valid 位和 tag 来帮助判断 cache 是否命中。

地址为 64 位，记 cache 总大小为 C bytes，相连度为 W，块大小为 B bytes，则有 $index$ 需要位数为 $\log(\frac{C}{BW})$，tag 需要位数为 $64-\log(B)-\log(\frac{C}{BW})$ 因此布局方面元数据开销 (指 valid 位与 tag 位) 为

$$(1+(64-\log(B)-\log(\frac{C}{BW})))\times\frac{C}{B}=(65-\log(\frac{C}{W}))\times\frac{C}{B}\quad bit$$

3.2 相连度

相连度是 cache 的一个重要参数，一般来说，提高相连度能够降低 cache 缺失率，但同时也会增加组内查找的开销。各块大小的相连度比较见图7

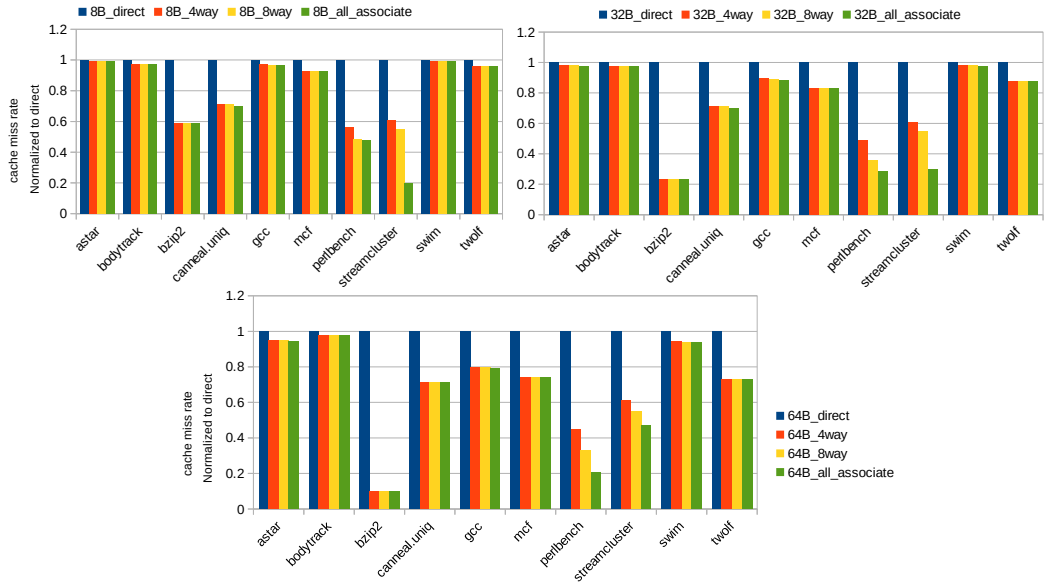

图 7: 相连度对 cache 缺失率的影响 (Normalized)

可以看到，对于所有的块大小配置，随着相连度的提高，cache 缺失率也随之下降。

3.3 块大小

cache 块越大，越能利用好空间局部性，但同时，cache 的行数减少，缺失率增大，并且块越大取块的时间就越长，增加了缺失损失。块大小的比较见图8

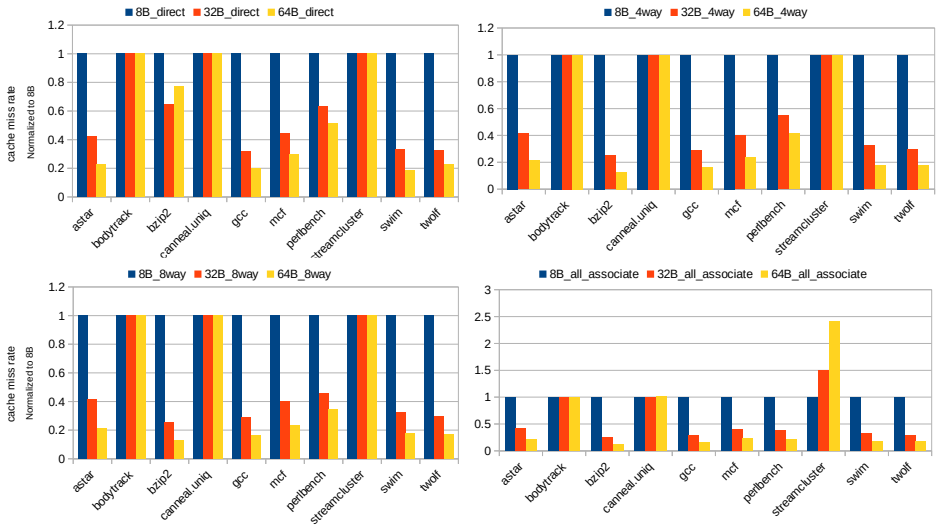

图 8: 块大小对 cache 缺失率的影响 (Normalized)

可以看到，大多数情况下，cache 块越大缺失率越小，此时 cache 行数减少带来的劣势要小于更好利用空间局部性带来的优势。

4 写策略

考虑 128KB 的 cache, 块大小为 8B，8 路组相连，并采用 LRU 替换算法，下面对写回/写直达 (wb/wt) 与写分配/写不分配 (alloc/nalloc) 进行分析

4.1 元数据开销

写回需要一个 dirty 位来标注是否被修改，而写直达则不需要，因为它是每次都往 Memory 里写，有着强一致性。而写分配和写不分配在元数据开销上都没有额外的开销。

4.2 cache 缺失率

如表4和图9所示，可以看到在绝大多数 benchmark 下，写分配相比于写不分配的 cache 缺失率要低的多。由于写不分配在写缺失的时候会直接

	8B_4way	32B_4way	64B_4way	8B_8way	32B_8way	64B_8way
astar	0.232791	0.0962993	0.0500989	0.232848	0.0962753	0.0500012
bodytrack	0.00248153	0.00248153	0.00248153	0.00248153	0.00248153	0.00248153
bzip2	0.0121705	0.00306328	0.0015445	0.0121705	0.00306328	0.0015445
canneal.uniq	0.00763138	0.00763138	0.00763138	0.00764413	0.00764413	0.00764413
gcc	0.041095	0.0119783	0.00677936	0.0409787	0.011924	0.00675415
mcf	0.0457593	0.018245	0.0108371	0.0457593	0.018245	0.0108371
perlbench	0.0207118	0.011357	0.00853104	0.0179016	0.00822165	0.00624506
streamcluster	0.0264714	0.0264714	0.0264714	0.0238197	0.0238197	0.0238197
swim	0.0653841	0.0211812	0.011491	0.0653676	0.0211647	0.0114646
twolf	0.0114037	0.00340911	0.00198416	0.0114017	0.00340083	0.00197588

表 3: 不同布局对缺失率的影响

将数据写入内存，不会更新 cache，因此当经常对小部分地址进行写操作的时候，显然写不分配由于不更新 cache 就会使得 cache 缺失率很高。

4.3 Memory 访问

写直达与写回的，写分配与写不分配的一个重要区别在于对于下级内存 Memory 的访问，访问 Memory 的时间显然大于只访问 cache 的时间，而因此 Memory 的访问次数是比较这些策略的关键因素。

从图10可以看到，写直达与写不分配非常明显得提高了 Memory 的访

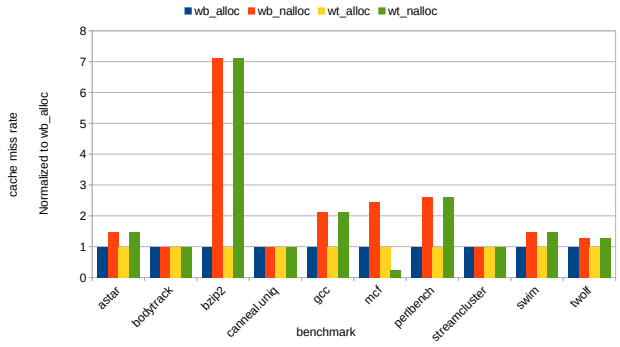

图 9: 写策略对 cache 缺失率的影响 (Normlized)

	wb_alloc	wb_nalloc	wt_alloc	wt_nalloc
astar	0.232848	0.344989	0.232848	0.344989
bodytrack	0.002482	0.002482	0.002482	0.002482
bzip2	0.01217	0.086699	0.01217	0.086699
canneal.uniq	0.007644	0.007644	0.007644	0.007644
gcc	0.040979	0.086652	0.040979	0.086652
mcf	0.045759	0.111469	0.045759	0.0111469
perlbench	0.017902	0.046634	0.017902	0.046634
streamcluster	0.02382	0.02382	0.02382	0.02382
swim	0.065368	0.096104	0.065368	0.096104
twolf	0.011402	0.014508	0.011402	0.014508

表 4: 写策略对 cache 缺失率的影响

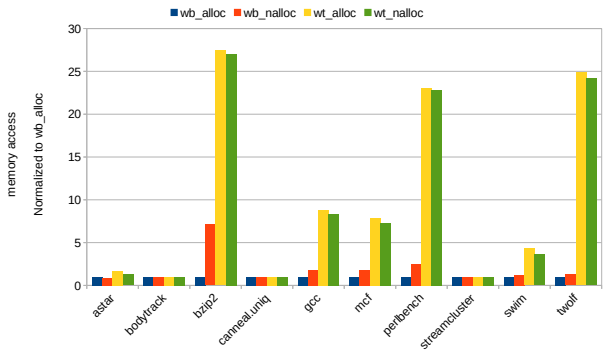

图 10: 写策略对 Memroy 访问次数的影响 (Normalized)

问次数，在一些 benchmark 上 (bzip2 等) 甚至到达了 20x 左右的访问次数翻倍。由于 Memory 的访问时间大于 cache 访问时间，在实际使用中，写回、写分配策略性能会显著优于写直达、写不分配策略。

参考文献

[1] Qureshi, Moinuddin K., et al. "Adaptive insertion policies for high performance caching." ACM SIGARCH Computer Architecture News 35.2 (2007): 381-391.

[2] Duong, Nam, et al. "SCORE: A score-based memory cache replacement policy." 2010.