

高等计算机体系结构

C-Burst 设计

姜天洋

2018310804

要求分析

- Memory Cache size: 128 MB
- Block size: 4 KB
- Cache replacement algorithm: LRU（性能区域）+C-Bursty（能耗区域）
- Initial partition: 性能区域（50%）+C-Burst（50%）
- 可以容忍的 performance loss:（相对全部用 LRU 管理，缺失率提高 10%）
- 每次划分调整的粒度是 2MB

由条件可知，cache 内共可以存储 $128\text{MB}/4\text{KB}=32768$ 个 entry。

算法分析与实现

PR 组织

PR 区域使用的是 LRU 算法。LRU 类似第一次实验中的方法，采用全相连的方式。由于这次跑的 trace 规模较大，所以不能简单的采用复杂度为 $O(N)$ 的 LRU，我采用了复杂度为 $O(\log N)$ 借助 set 实现的 LRU。

EAR 组织

```
class EAR
{
private:
    set<BG>p[35];
    set< block >st;
    ll tot, up;
}
class BG
{
private:
    set<block>*q;
    ll ep;
}
```

EAR 及其相关的数据结构实现见上图。BG 代表 Block Group，里面有一个 set 存放相应的 block，ep 用来标记该 BG 属于哪个 block group。EAR 的 p 是一个 set 的链表，由于 STL 不支持 set 套 set，所以 BG 采用了指针的方式管理一个 set。

set<block>st 是用来方便根据 trace 的 tag 方便地对 EAR 内的数据进行查询，否则每次查询的复杂度都是 $O(N)$ ，复杂度无法接受。

EAR+PR 组织

将其上两种方法整合，就得到了以下方法。对于每次动态调整划分的区域，由于如果每次探测再进行调整，该方法无法充分发挥调整后的作用。可能一次调整本来会有作用，但是因为 EAR+PR 的 hit rate 相较只使用 PR 的更低，所以很有可能紧接着又进行一次调整。所以我设置了每间隔 1000 条 trace 进行一次调整探测。并且，epoch 的时间间隔设置为 1ms。

另外，在论文中，需要 In Buffer 这样一个内核缓冲区缓冲还未被插入 EAR 区域的 BG，并且经过思考，我认为如果之后有 trace 命中了 In Buffer 内的数据，不应该算为命中率，

因为如果算作命中,这里也可以认为是一个小型的PR,那么此方法将与PR的方法没有区别;同时 In Buffer 内的数据还要进行去重,因为不去重插入 EAR 的数据就会重复,这显然对查找的开销及空间的浪费。

具体的流程为:

1. 首先读一个新的地址,以及它的时间间隔。如果加上时间间隔之后超过当前时间戳,则进行第2步;如果还在当前时间戳内,则进行第5步。
2. 检查 EAR 剩余空间,是否满足新块组大小,若满足,则直接建立新块组,将 In Buffer 中的数据拷贝进新块组,并根据其访问量大小,加入到 EAR 中;若剩余空间不满足则进行第3步。
3. 从 EAR 最大的 BG 的 set 开始搜索,将其逐出。
4. 重复第三步,直到剩余空间大小满足新块组大小,逐出顺序是按照大时间优先,再按大小来的。然后再建立新块组,将 In Buffer 中的数据拷贝进新块组,并根据其访问量大小,加入到 EAR 中。
5. 查看 PR 区域中是否命中,若命中则记录命中;若不命中,进行第6步。
6. 搜索 EAR 中是否命中,命中则记录命中,并将数据块 tag 升级至 PR 中,根据时间戳顺序和去除数据块后 BG 大小改动其位置,同时 PR 替换出的地址加入下面对应时间戳的 BG 中,并调整加入地址后 BG 的位置(所以在 PR 的 lru 数据结构中,cacheline 要存它来自于哪个 epoch),如果对应的 epoch 已经不在 EAR 中,则弃置;若不命中则进行第7步。
7. EAR 未命中,把地址存在当前 epoch 的 BG 中加入,并在 epoch 结束后一起加到 EAR 中。

实验结果

由于查询 EAR 内的 cacheline 需要的时间太长,我选取了 MSR 的 lvm0 trace 的前 2w 条进行了实验。对于一条 trace,由于要对 trace 进行切分,所以只要 trace 中的任意一个访问出现了 miss,我就认为这条 trace miss 了。需要输出 miss trace 的间隔的 50%的值,由于很多 trace 是同时到来的,而这种同时到来的 trace 一起发生 miss 的概率又很高,所以间隔为 0 的量很大,我不仅输出了所有 miss trace 的间隔的 50%的值,还输出了 miss trace 的间隔的 80%的值。

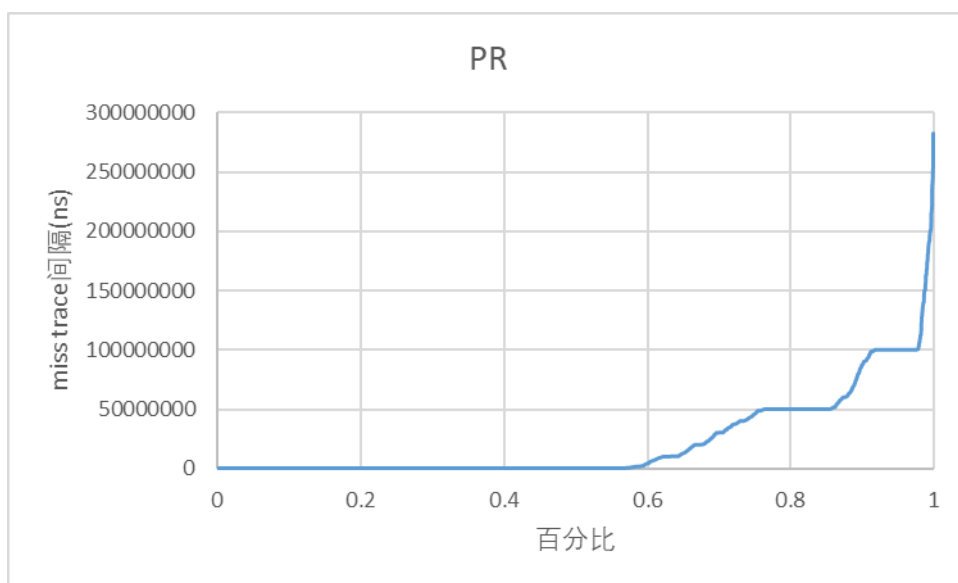
PR 方法

命中率 79.75%

50%的 miss trace gap: 1000ns

80%的 miss trace gap: 49999000ns

CDF 图



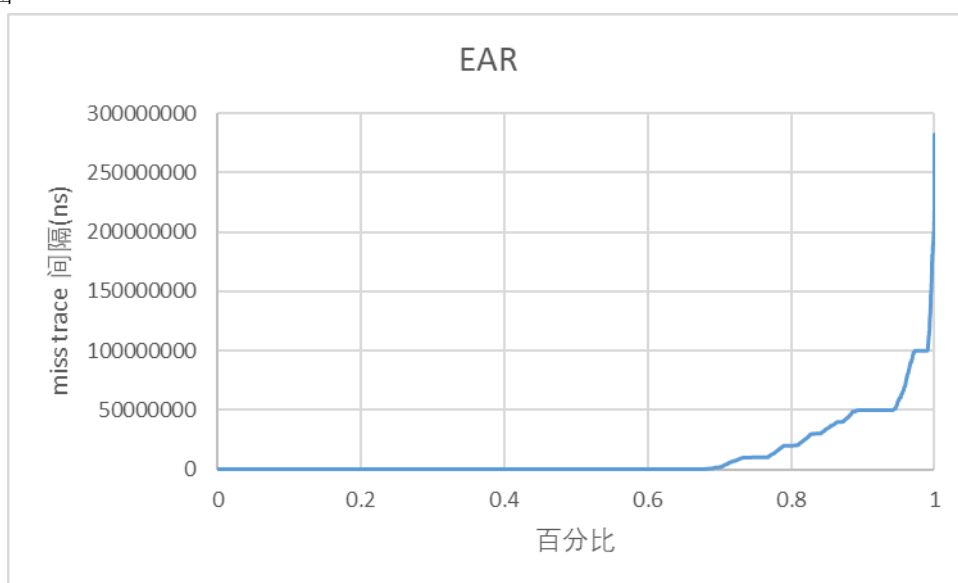
EAR 方法

命中率 62.13%

50%的 miss trace gap: 0ns

80%的 miss trace gap: 19998000ns

CDF 图



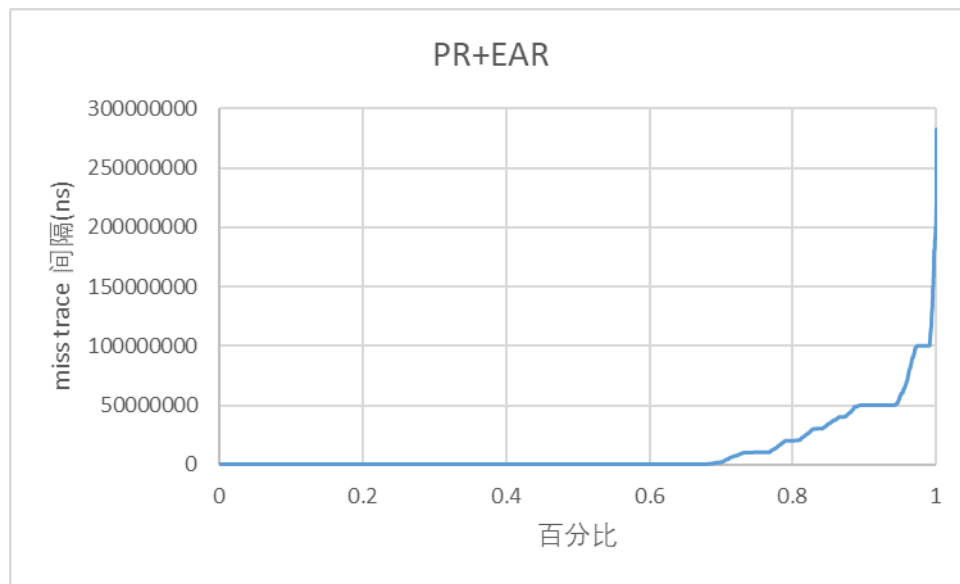
EAR+PR 方法

命中率 62.22%

50%的 miss trace gap: 0ns

80%的 miss trace gap: 19999000ns

CDF 图



实验截图

D:\codeblocks\debug\bin\Debug\debug.exe

```
1000 ok
2000 ok
3000 ok
4000 ok
5000 ok
6000 ok
7000 ok
8000 ok
9000 ok
10000 ok
11000 ok
12000 ok
13000 ok
14000 ok
15000 ok
16000 ok
17000 ok
18000 ok
19000 ok
20000 ok
EAR 62.13
```

```
Process returned 0 (0x0)   execution time : 53.624 s
Press any key to continue.
```

上图为单纯 EAR 方法的截图

D:\codeblocks\debug\bin\Debug\debug.exe

```
1000 ok
judge 1000 1 41.40 3 29.70
2000 ok
judge 2000 1 64.65 3 51.10
3000 ok
judge 3000 1 72.77 3 58.67
4000 ok
judge 4000 1 76.88 3 62.17
5000 ok
judge 5000 1 79.50 3 64.34
6000 ok
judge 6000 1 73.33 3 59.23
7000 ok
judge 7000 1 67.66 3 53.99
8000 ok
judge 8000 1 69.14 3 54.60
9000 ok
judge 9000 1 71.24 3 55.90
10000 ok
judge 10000 1 73.05 3 57.06
11000 ok
judge 11000 1 74.64 3 58.35
12000 ok
judge 12000 1 75.80 3 59.64
13000 ok
judge 13000 1 76.85 3 60.55
14000 ok
judge 14000 1 75.12 3 58.49
15000 ok
judge 15000 1 76.13 3 59.55
16000 ok
judge 16000 1 77.05 3 59.94
17000 ok
judge 17000 1 77.86 3 60.36
18000 ok
judge 18000 1 78.52 3 61.12
19000 ok
judge 19000 1 79.17 3 61.73
20000 ok
judge 20000 1 79.75 3 62.22
LRU 79.75
EAR+LRU 62.22
```

```
Process returned 0 (0x0)   execution time : 54.014 s
Press any key to continue.
```

上图为 PR 及 PR+EAR 的方法截图，judge 是空间进行调整的 log

实验分析

从实验结果中容易看出，LRU 的方法是最好的，这是毋庸置疑的。结合 PR 和 EAR 的方法，借助了 ghost LRU 调整大小，对 hit rate 相比较普通的 EAR 有微弱的优势，这可能是和 epoch 的设置也有一定关系。从最后的实验截图中也可以看出 EAR+PR 的方法的确进行了

若干次的空间调整。