Cache 系统组织与设计 实验报告

计 76 陈之杨 2017011377

2020.4

1 使用方法

`cache_sim.cpp` 为源代码，使用 `g++ -o cache_sim cache_sim.cpp -O2` 命令编译。运行 `cache_sim` 进行 cache 模拟，参数如下所示：

- `-block [blocksize]`，设置缓存块大小为 `blocksize`。
- `-alloback/allothro/aroback/arothro`，设置写策略为写分配-写回/写分配-写直达/写绕过-写回/写绕过-写直达。
- `-full/direct/4way/8way`，设置映射规则为全关联/直接映射/4 路组关联/8 路组关联。
- `-lru/random/tree`，设置替换策略为 LRU/随机替换/二叉树替换。
- `-log [filename]`，输出日志到 `filename` 文件中。

当某项参数缺省时，采用默认 cache 布局（块大小 8B，8 路组关联，LRU 替换策略，写分配，写回）。提交的 `astar.log` 等文件为默认布局下，给定 trace 文件的访问日志。程序默认从标准输入读入，如果要从指定 `trace` 文件中读入，需要重定向输入，如`./cache_sim < astar.trace`。

程序运行完毕后，会在标准输出中打印 cache 命中率。

2 实现细节

程序中定义了 `Cache` 类作为访问的接口，内部定义了一个 `Group` 类数组，描述 cache 中的每个组（将直接映射视作 1 路组关联，全关联将整个 cache 视为一个组）。访问某一内存地址时，`Cache` 类负责提取索引位，找到对应的组，然后将具体的访问交给 `Group` 类执行。

每个 `Group` 类中，用一个二维 `char` 数组 `metaData` 维护每个块的元数据，再定义一个 `Replace` 类执行替换策略。

Replace 类是一个抽象类，只定义了 insert,access,replace 三个接口的形式，对于不同的替换策略，分别定义一个类继承 Replace 实现接口。例如，LeastRecentlyUsed 类中定义了 char 数组 stack 实现 LRU 的堆栈法，BinaryTree 类中定义了 char 数组 tree 维护二叉树信息，其中二叉树顶点用完全二叉树的方式组织，即顶点 p 的左右儿子分别为 2p 和 2p+1，这样整个二叉树可以用组大小个连续空间实现。

所有 cache 系统中需要维护的信息都用 char 数组存储，例如 8 路组关联的 LRU 策略需要 24 位，使用 3 个 char 存储，8 路组关联的二叉树策略需要 8 位，使用 1 个 char 存储（由于二叉树的根从 1 开始，需要额外的 1 位）。8 路组关联 8B 块的元信息使用 7 个 char 存储。为了方便从 char 数组中读取或写入连续一段二进制位，实现了 editBits 和 readBits 两个函数统一操作。

3 实验结果

由于所有的参数设置都需要测试 4 个重点 trace 的结果，以下用 a/b/c/d 的形式表示 astar,bzip2,mcf,perlbench 四个重点 trace 的命中率分别为 a%,b%,c%,d%（保留四位有效数字，加粗为最优命中率）。

3.1 局部性分析

首先对测试的 trace 进行局部性分析。

程序 reuse_dist.cpp 中统计了 trace 对每个地址相邻两次访问的距离（重用间距），以 5 次读/写为单位，相应的直方图如图 1所示。

总体来看 astar.trace 的局部性是较差的，有很多地址空间的访问间距集中在 200 $\sim$ 300 的范围中，推测由于 A* 算法是启发式搜索算法，缺少顺序访问的规律性。bzip2.trace 的局部性最好，几乎所有地址空间的访问间距都不超过 20，推测这是由于 bzip2 是压缩程序，运行过程中很少会相隔一段时间访问同一块数据。mcf.trace 的局部性较好，但是有 50000 左右个地址的访问距离集中在 600 左右，可以推测这是由于程序局部出现了 cache 不友好的代码（例如，两次顺序访问同一个长度为 50000 左右的数组，超出了 cache 的容量）。perlbench.trace 的直方图曲线比较平滑，局部性介于 astar 和 bzip2 之间。之后的实验结果也与这四个 trace 的局部性分析结果基本一致（bzip2 命中率最高，astar 最低）。

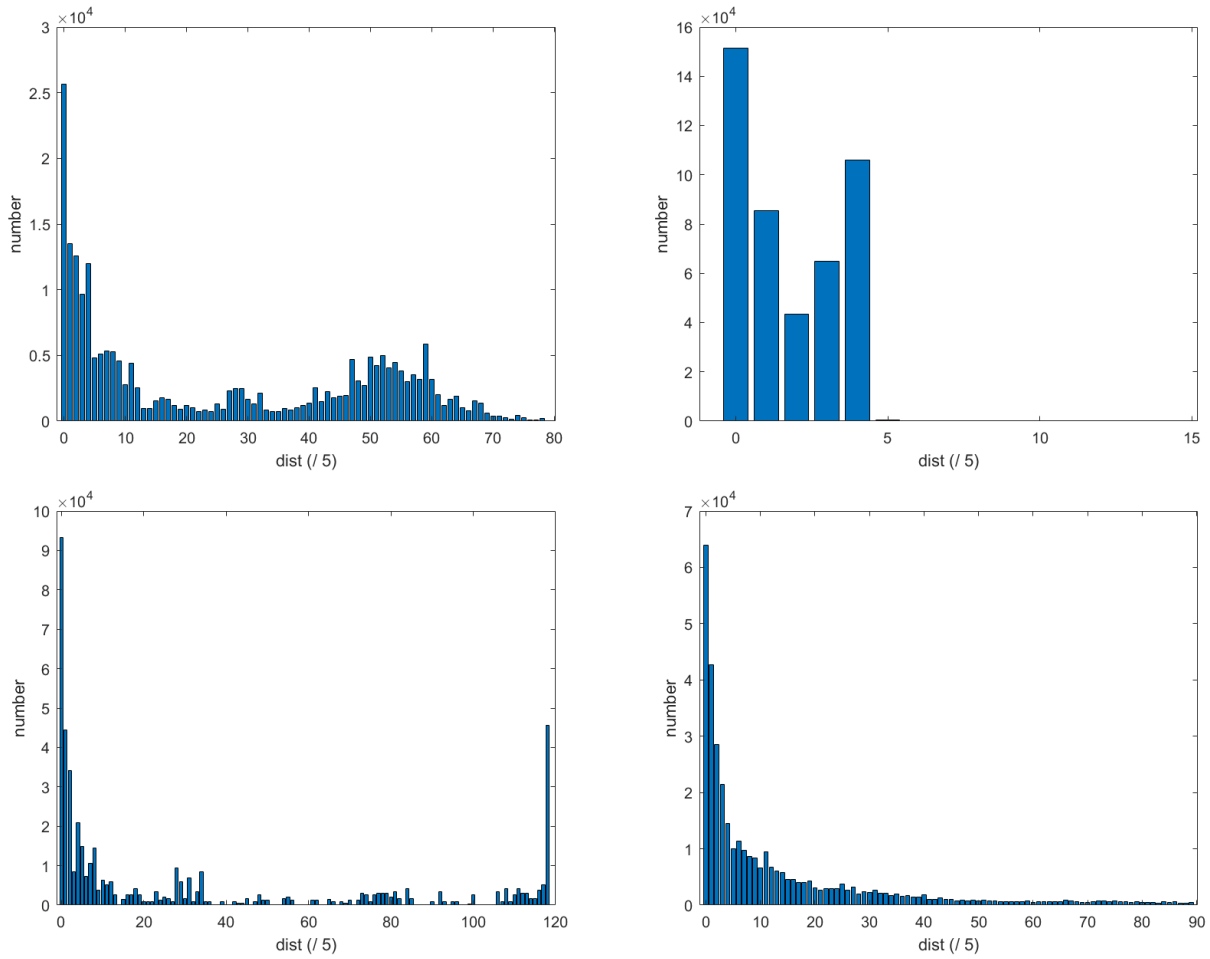

图 1: 重用间距的直方图。左上、右上、左下、右下依次对应 `astar,bzip2,mcf,perlbench`。注意横轴以 5 个距离为单位，纵轴表示数量。

3.2 组织方式

组织 \ 块大小	8B	32B	64B
直接映射	76.60/97.94/95.06/96.33	90.16/98.67/97.80/97.69	94.73/98.41/98.54/98.11
4 路组关联	76.72/98.78/95.42/97.93	90.37/99.69/98.18/98.86	94.99/99.85/98.92/99.15
8 路组关联	76.72/98.78/95.42/98.21	90.37/99.69/98.18/99.18	95.00/99.85/98.92/99.38
全关联	76.74/98.78/95.42/98.24	90.41/99.69/98.18/99.34	**95.03**/**99.85**/**98.92**/**99.61**

观察上表可以发现，总体来说，关联数越高，块大小越大，cache 的命中率越高。

首先考虑关联数。很容易证明高关联数的命中率一定是严格优于低关联数的，但是实验结果显示这种影响微乎其微，而如果考虑硬件实现的话，高关联数的代价是远远高于低关联数的。假设 cache 块的数量为 n，那么直接映射/常数路组关联硬件上需要实现的连接数是 $O(n)$ 的，而全关联则是 $O(n^2)$ 的。另外考虑到 LRU 替换策略需要维护一个栈存储访问顺序，如果采用全关联，栈的规模是巨大的，因而不具有实用性。这点在程序模拟时也有所体现：由于修改访问栈时需要移动栈中元素，每次访问的代价在最坏情况下是 $O(n)$ 的，因此程序模拟全关联时的执行速度很慢（需要几分钟才能模拟完毕）。关联数越高，元数据开销空间越大，因为每个 cache 块都要占用一个 LRU 的栈寄存器，但是关联数越高，栈中每个元素就需要更多的位数（全相联为 $\log n$ 位，而常数路组相联只需要常数位）。

再考虑块大小。测试的 4 个 trace 都是块越大，命中率越高。如果程序有较好的局部性，那么增加块大小无疑是有利于提高命中率的（读取一个块时，块中其它字节很可能之后也会被访问到）。但如果程序的局部性较差，增加块大小很可能反而降低命中率，因为在 cache 中存入了过多无用的数据。注意到 `astar` 在增加块大小后命中率显著提升，推测是因为 A* 算法需要用数据结构维护所有的搜索状态，如果 cache 块太小，一个搜索状态可能需要多次读取内存。但如果块大小达到了搜索状态的大小，那么只需要一次读内存就可以把整个状态装入 cache。此外，增加块大小，块的数量和索引位长度就减少了，从而可以减少元数据开销。

3.3 替换策略

替换策略	命中率
LRU	76.72/**98.78**/**95.42**/98.21
随机	**76.77**/98.78/95.40/**98.22**
二叉树	76.71/98.78/95.42/98.22

三种策略的命中率基本一致，二叉树替换策略是 LRU 策略的近似，因而略低于 LRU。随机替换策略的命中率有一定的波动性，但与 LRU 不相上下。

LRU 的替换逻辑最复杂，访问命中时需要线性扫描整个栈，找到对应位置并挪到栈顶，这涉及 $O(n)$ 个寄存器的赋值（假设采用 n 路组相联）。替换则需要将栈底元素挪到栈顶，也涉及

$O(n)$ 个寄存器的赋值。LRU 的元数据开销也最大，为 $O(n \log n)$ 位。二叉树替换是 LRU 的近似，每次访问/替换只需要操作 $\log n$ 位的取反，开销较小，元数据也只需要 $O(n)$ 位。随机替换最为简单，访问时无须操作，替换时返回一个随机数即可，也不需要存储任何元数据。

由此也可以看出，随机替换虽然逻辑简单，但有很强的实用性，并不比逻辑复杂的 LRU 差。

3.4 写策略

写策略	写回	写直达
写分配	**76.72/98.78/95.42/98.21**	**76.72/98.78/95.42/98.21**
写绕过	65.50/91.33/88.85/95.34	65.50/91.33/88.85/95.34

因为我们在程序模拟中只考虑命中率的区别，所以写回和写直达的结果是一致的，在模拟时也只有写回时每个 cache 块需要额外存储 1 位元数据的区别。这里无须详细讨论。

由于被写的地址在之后很可能会被访问，总体上写分配的命中率大大超过写绕过。这其中对 `astar` 的影响最大。推测是由于 A* 算法会频繁地修改搜索状态中的值（例如搜索树顶点的父亲），因此读写交替较多，使用写分配能大大提高命中率。