Cache 替换框架及算法实现

2017011313 刘丰源

2020 年 4 月 8 日

目录

第一节　实现内容概要

1. 可修改参数的 cache 替换框架，包括缓存大小 (cacheSize)、相联度 (assoc)、块大小 (lineSize)、替换算法 (replPolicy)、写策略 (writeAlloc)。以继承的形式实现替换算法，每个算法实现为独立的文件。
2. 实现了基础的 LRU、PLRU、RANDOM ，根据文献 Peress Y, Finlayson I, Tyson G, et al. CRC: Protected lru algorithm[C]. 2010. 实现了 PROTECT_LRU 。
3. 根据输入的参数计算 tag 的长度，通过位操作，以 bit 为单位维护元数据，将所需内存降低至最小。
4. LRU 的队列使用 uint_8 数组保存,长度根据输入的参数确定,每个数据长度为 $\log_2(assoc)$ ，则数组长度为 $assoc * \log_2(assoc)/8$ 。
5. PLRU 使用 bit binary tree 实现，也由 uint_8 数组保存，数组长度为 $assoc/8$ 。

第二节　代码实现细节

2.1　运行代码

- **run.py**

用于运行框架的脚本，在文件中修改参数（相联度等）。然后执行 python3 run.py 即可运行。

- **main.cpp**

主要用于读文件，处理文件内容，通过接口将参数传给 cache 替换框架。

2.2　替换框架

- **utils.h**

一些工具函数,以及一些结构体的定义,cache 的元数据结构体。对于全相联的情况,tag 的长度为 64 ，对此情况进行了特判。部分代码如下:

Code

```
class LINE_STATE {
   private:
    u8* src;

   public:
    bool isValid() {
        if (tagLen < 63)
            return (*(ADDRINT*)(src)) >> 63;
        else
            return src[8] & 1;
    }

    bool isDirty() {
        if (tagLen < 63)
            return (*(ADDRINT*)(src)) >> 62;
        else
            return (src[8] >> 1) & 1;
    }

    void setValid(bool flag) {
        if (tagLen < 63) {
            if (flag)
                (*(ADDRINT*)(src)) |= (((ADDRINT)(1)) << 63);
            else
                (*(ADDRINT*)(src)) &= (~(((ADDRINT)(1)) << 63));
        } else {
            if (flag)
                src[8] |= 1;
            else
                src[8] &= (~(u8)(1));
        }
    }

    void setDirty(bool flag) {
        if (tagLen < 63) {
            if (flag)
                (*(ADDRINT*)(src)) |= (((ADDRINT)(1)) << 62);
            else
                (*(ADDRINT*)(src)) &= (~(((ADDRINT)(1)) << 62));
        } else {
            if (flag)
                src[8] |= 2;
            else
                src[8] &= (~(u8)(2));
        }
    }

    ...
};
```

- **cache_manager.[h|cpp]**

cache 替换框架的主要内容，主要的对外接口 LookupAndFillCache 接受两个参数：访存地址、访存类型。会处理参数为组 id ，传给替换算法，由算法决定哪块 cache 被替换。

主要用于保存 cache 的元数据，判断缓存是否命中。若命中，则调用算法的 UpdateReplacementState 接口，若未命中，则先调用算法的 GetVictimInSet 接口，然后再调用算法的 UpdateReplacementState 接口。针对每个函数的具体内容，以及每一行的作用，**在代码中通过注释详细写出**。

2.3　基础替换算法

- **replacement_state.h**

提供抽象类 CACHE_REPLACEMENT_STATE ，替换算法需要继承该类。子类需要实现 GetVictimInSet、UpdateReplacementState、InitReplacementState 三个纯虚函数。

- **REPL_RANDOM.h**

随机替换算法的实现非常简单：每次只需随机选择一块进行替换即可。这样的好处显而易见，由于没有维护任何额外的信息，也没有进行任何额外的计算，这样在额外空间消耗、信息维护用时、实现难度上有优势。另外。由于其随机性，它对各种故意构造的访问顺序具有一定的普适性。然而这种方法并没有考虑到访问时顺序的性质，也没有对访问的模式进行考虑。对于相同的程序，其运行效率也会不稳定，这是其随机性造成的。

- **REPL_LRU.h**

首先需要一个按位的 LRU 队列，由于无法预先知道相联度，而元素的长度依赖于相联度，所以定义全局变量 lruElemNum 和 lruElemLen ，在算法初始化时给这两个全局变量赋值。

Code

```
u32 lruElemNum;
u32 lruElemLen;

REPL_LRU(u32 _sets, u32 _assoc, u32 _pol)
        : CACHE_REPLACEMENT_STATE(_sets, _assoc, _pol) {
        lruElemNum = _assoc;
        lruElemLen = FloorLog2(_assoc);
        if (lruElemLen == 0)
            lruElemLen = 1;
        InitReplacementState();
    }
```

创建 LRU 队列时根据这两个全局变量调整数组长度。LRU 的队列使用 uint_8 数组保存，长度根据输入的参数确定，每个数据长度为 $\log_2(assoc)$ ，则数组长度为 $assoc*\log_2(assoc)/8$ 。部分代码如下：

Code

```
class LRUQueue {
   private:
    u8* queue;

   public:
    LRUQueue() {
        u32 len = (lruElemNum * lruElemLen + 7) / 8;
        if (len == 0)
            len = 1;
        queue = new u8[len];
    }

    u8 get_bit(u32 pos) { return (queue[pos / 8] >> (pos % 8)) & 1; }

    void set_bit(u32 pos, bool flag) {
        if (flag)
            queue[pos / 8] |= (((u8)(1)) << (pos % 8));
        else
            queue[pos / 8] &= (~(((u8)(1)) << (pos % 8)));
    }

    void set(u32 pos, u32 data) {
        for (u32 i = pos * lruElemLen; i < pos * lruElemLen + lruElemLen; ++i) {
            if (data & 1) {
                set_bit(i, true);
            } else
                set_bit(i, false);
            data >>= 1;
        }
    }

    ...
};
```

LRU 算法的实现中，每当需要替换的时候，都会找到最长时间未使用的块进行替换。具体而言，它维护了一个栈，每当一块被访问，就将其从栈底里去除一块并重新压栈。该算法充分利用了程序访问的局部性。如果程序访问的空间范围较小、缓存容量较大，这个算法的命中率将会较为可观。但是 LRU 无法处理数组大小大于 cache 容量的逐列扫描访问模式，缺失率将达到 100% 。

- **REPL_PLRU.h**

PLRU 使用 bit binary tree 实现，也由 uint_8 数组保存，数组长度为 $assoc/8$ 。相比于标准的 LRU 算法，其多了一定的随机性（某种程度上可以认为是简易的预测）。但是由于对于容量很大的 Cache，LRU 和 RANDOM 的命中率差别不大，所以 PLRU 的缺失率也和 LRU 接近。部分代码如下：

Code

```
class PLRUTree {
   private:
    u8* tree;

   public:
    PLRUTree() {
        i32 len = treeSize / 8;
        tree = new u8[len];
        memset(tree, 0, len * sizeof(u8));
    }

    void set_bit(i32 pos, bool flag) {
        if (flag) {
            tree[pos / 8] |= (((u8)(1)) << (pos % 8));
        } else {
            tree[pos / 8] &= (~(((u8)(1)) << (pos % 8)));
        }
    }

    void update(u32 wayID) {
        bool num[treeHeight];
        for (i32 i = 0; i < treeHeight; ++i) {
            num[i] = ((wayID & 1) == 1);
            wayID >>= 1;
        }
        i32 ind = 0;
        for (i32 i = treeHeight - 1; i >= 0; --i) {
            set_bit(ind, !num[i]);
            if (!num[i]) {
                ind = ind * 2 + 1;
            } else {
                ind = (ind + 1) * 2;
            }
        }
    }

    ...
};
```

2.4　拓展替换算法

- **REPL_PROTECT_LRU.h**

参考文献：Peress Y, Finlayson I, Tyson G, et al. CRC: Protected lru algorithm[C]. 2010.

传统 LRU 只考虑了最近访问时间，没有考虑访问频率。PROTECT_LRU 增加了对访问频率的考虑，其思路是：保护访问频率最高的 NUM_MU 个路不被替换，剩下的路通过传统 LRU 算法进行替换。同时为了减小计算量，频率统计的大小不能超过 MAX_COUNTER_VAL 。如果达到该数值，则将所有路的访问次数除以二。

该算法可以有效缓解常用数据被替换出去的问题，但是也有新的问题：可能存在“曾经常用，但是后续不再被使用的数据”长期占据缓存块，导致有效缓存减少。因此需要适当调整的 NUM_MU 和 MAX_COUNTER_VAL ，以达到更好的效果。

第三节　缺失率分析

3.1　参数

几种算法的结果都十分接近，以下以 lru 替换算法为例进行分析。

组织方式\块大小	8	32	64
直接映射	23.40	9.84	5.27
4 路组相联	23.28	9.63	5.01
8 路组相联	23.28	9.63	5.00
全相联映射	23.26	9.59	4.97

- **相联度**

从数据中可以看出，相联度对缺失率的影响并不大。增加相联度，相联度只会有轻微的降低（冲突减少）。替换算法和块大小才是影响缺失率的主要因素。

- **块大小**

增加块大小，缺失率有明显降低。除了冲突减少，还得益于数据的局部性，每次以块为单位进行替换使得 cache 能够预取数据，因此缺失率有了明显降低。

- **写分配**

以 8 路 8B 块分析写分配对缺失率的影响：写分配为 23.28 ，写不分配为 34.50 。由于局部性原理，刚写过的数据需要再次访问的可能性较大。写分配本质是就是在预测未来将要访问的数据。而由于实验给出的 cache 容量较大，使得写分配的预测结果准确率更高。

- **写直达**

在真实机器中，出于一致性考虑，会定期将 cache 的 dirty 块数据写入 memory ，或者通过特定的指令进行同步。但是在实验中，并不考虑以上两种情况，脏页永远不会主动刷入内存，因此写直达和写回的结果一致。

3.2　算法

LRU 由于拥有较大的视野，对块使用的频繁程度有着精确的估计，在大多数情况下都有着较低的缺失率。但实际上我们可以发现，每次更新缓存信息或替换块的时候，LRU 都需要遍历每个 assoc 来更新信息，即时间复杂度为 O(assoc)。当 assoc 数量较大的时候该算法的时间消耗将会大大增加。

随机算法的时间复杂度是 O(1) 的，这是因为它每次都是随机选择一个块，并且在更新缓存信息的时候什么都不做，所以理论上应该是最快的。同时其缺失率与 LRU 接近，所以在是一个非常实用且常用的替换算法。

PLRU 缺失率与 LRU 接近，由于其算法具有一定的随机性和预测性，所以应该缺失率应该位于 RANDOM 和 LRU 之间。但是由于 RANDOM 和 LRU 的缺失率本身就十分接近，因此三者缺失率都很接近。

PROTECT_LRU 在 cache 容量较小时，相比于另外三种算法有较好的表现，但是现代 CPU 都具有较大的 cache ，所以一般情况下优势并不明显。

第四节　实验收获

缓存替换算法是一个成本和效能、时间和空间、以及不同程序结构的多因素的权衡。通过这次实验，我了解到了各个缓存替换算法的大致思想，熟悉了几个基础算法的具体实现，对 cache 的理解也越发深刻了。（全相联的数据跑了一万年，让我深刻的意识到了该做法的不切实际）感谢老师和助教给我这次提升和锻炼的机会。