ICS I cachelab 报告

中国人民大学 sheriyuo

摘 要

RUC 2023-2024 计算机系统基础 I cachelab 的解题思路和实现。

1 Part A

1.1 基本结构

```
typedef struct block {
    bool isValid;
    int tag;
    int time;
} block;
typedef block* set;
typedef unsigned long addr_t;
typedef unsigned data_t;
typedef struct cache {
    int S;
    int s, e, b;
    set* head;
} cache;
typedef struct OP {
    char op;
    int size;
    addr_t addr;
} OP;
```

定义 `block` 作为一个缓存块，由 `block*` 连成一组，最后连成 `block**` 的 `cache`。

采用 LRU 来弹出，于是一个缓存块需要额外记录最近访问时间 `time`，`cache` 中的 `s, e, b` 即代表输入的 `-s -E -b`。

1.2 各种函数

通过内联的 `get_s` 和 `get_t` 函数来从一个特定的地址中获取索引和标识符。

```
static inline data_t get_s(addr_t addr, int s, int b) {
    return (addr >> b) & ((1 << s) - 1);
}
static inline data_t get_t(addr_t addr, int s, int b) {
    return addr >> (s + b);
}
```

cache 空间的分配使用 allocSet，清除使用 freeSet。

```
cache allocSet(int s, int b, int e) {
    int S = 1 << s;
    set* head = calloc(S, sizeof(set));
    for (int i = 0; i < S; i++)
        head[i] = calloc(e, sizeof(block));
    cache c = {S, s, e, b, head};
    return c;
}
void freeSet(cache *c) {
    for (int i = 0; i < c->S; i++) {
        free(c->head[i]);
        c->head[i] = NULL;
    }
    free(c->head);
    c->head = NULL;
}
```

使用 fetchOP 来从文件流中读取一条指令，用 sscanf 来格式化成操作类 OP。

```
OP fetchOP(FILE *file) {
    char ch[OP_LENGTH];
    OP op = {0, 0, 0};
    if (fgets(ch, OP_LENGTH, file))
    sscanf(ch, "\n%c %lx,%d", &op.op, &op.addr, &op.size);
    return op;
}
```

findBlock、findNewBlock 和 findEvictBlock 均采用顺序遍历组的形式，分别查找出对应块、空块和 LRU 弹出块，这一步骤可用 Hash 表优化至常数复杂度。

```
set findBlock(set s, int e, int t) {
    for (int i = 0; i < e; i++)
        if (s[i].isValid && s[i].tag == t)
            return s + i;
    return NULL;
}
set findNewBlock(set s, int e) {
    for (int i = 0; i < e; i++)
```

```
        if (!s[i].isValid)
        return s + i;
    return NULL;
}
set findEvictBlock(set s, int e) {
    int j = 0;
    for (int i = 1; i < e; i++)
        if (s[i].time < s[j].time)
            j = i;
    return s + j;
}
```

`loadOrStore` 函数通过判断 `op` 的类型来模拟 load 和 store 操作。对于一次索引为 s 标识为 t 的查找，如果 `findBlock` 命中，则查找命中；否则，选择一个新块更新其有效位和标识符，如果不存在新块则根据 LRU 找到弹出块更新。

`loadOrStore` 的 `isDisplay` 参数代表是否含 `-v`，`isMain` 函数代表是否被 `main` 直接调用，只有两者都同时为 `true` 才输出调试信息。以下是一个不含 Display(`-v`) 的 `loadOrStore` 函数：

```
void loadOrStore(cache *c, OP *op, bool isDisplay, bool isMain) {
    int s = get_s(op->addr, c->s, c->b);
    int t = get_t(op->addr, c->s, c->b);
    set res = findBlock(c->head[s], c->e, t);
    if (res) {
        hitCnt++;
    } else {
        missCnt++;
        res = findNewBlock(c->head[s], c->e);
        if (res)
            res->isValid = 1;
        else {
            evictCnt++;
            res = findEvictBlock(c->head[s], c->e);
        }
        res->tag = t;
    }
    res->time = localTime;
    localTime++;
}
```

`modify` 函数即分别实现一次 load 和 store 操作，此时传入的 `isMain = false`，只会输出一次调试信息。

```
void modify(cache* c, OP* op, bool isDisplay) {
    if (isDisplay)
        printf("%c %lx,%d ", op->op, op->addr, op->size);
```

```
    op->op = 'L';
    loadOrStore(c, op, isDisplay, false);
    op->op = 'S';
    loadOrStore(c, op, isDisplay, false);
    if (isDisplay)
        printf("\n");
}
```

调用 `./csim-ref -h` 复制得到帮助信息，调用 `print_HELP`，参数的处理采用了 `getopt` 的形式，得到 `optarg` 的 `<num>` 或 `<file>` 参数字符串，将其转换为初始化 `cache` 所需要的参数。

```
void getArguments(int argc, char **argv, int *s, int *e, int *b, char **t,
                  bool *isDisplay) {
    int opt;
    while ((opt = getopt(argc, argv, "hvs:E:b:t:")) != -1) {
        switch (opt) {
            case 'h': printHelp(); break;
            case 'v': *isDisplay = 1; break;
            case 's': *s = atoi(optarg); break;
            case 'E': *e = atoi(optarg); break;
            case 'b': *b = atoi(optarg); break;
            case 't': *t = optarg; break;
            default: printHelp(); break;
        }
    }
}
```

`main` 函数中，先打开文件流、设置缓冲区，并得到输入的参数，然后循环执行每一条操作，最后关闭文件、释放内存、输出结果。

```
int main(int argc, char *argv[]) {
    char *t = NULL;
    bool isDisplay = false;
    int s = 0, e = 0, b = 0;
    getArguments(argc, argv, &s, &e, &b, &t, &isDisplay);
    if (s <= 0 || e <= 0 || b <= 0 || t == NULL) {
        printf("Illegal Arguments\n");
        return -1;
    }
    cache c = allocSet(s, b, e);
    FILE *file = fopen(t, "r");
    setvbuf(file, NULL, _IOFBF, BUFFER_SIZE);
    OP op;
    while ((op = fetchOP(file)).op) {
        if (op.op == 'M')
        modify(&c, &op, isDisplay);
        else if (op.op == 'L' || op.op == 'S')
```

```
        loadOrStore(&c, &op, isDisplay, true);
        else if (op.op == 'I')
        continue;
    }
    fclose(file);
    freeSet(&c);
    printSummary(hitCnt, missCnt, evictCnt);
    return 0;
}
```

1.3 运行结果

driver.py 的运行结果：

```
Part A: Testing cache simulator
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27
```

-v 参数下，csim 与 csim-ref 也具有相同的调试信息输出：

```
sheriyuo@ROG-Strix-6P:~/cachelab-handout$ ./csim -v -s 2 -E 2 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1
L 22,1 hit
S 18,1 hit
L 110,1 miss
L 210,1 miss eviction
M 12,1
hits:4 misses:5 evictions:2
sheriyuo@ROG-Strix-6P:~/cachelab-handout$ ./csim-ref -v -s 2 -E 2 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:2
```

2 Part B

`trans.c` 的代码过于繁琐，详细实现见代码。

2.1 32×32

`cache` 参数为 `-s 5 -E 1 -b 5`，每一个缓存块可以存储 8 个 `int`，整个 32×32 的矩阵索引每 8 行一个循环，可以想到按照 8×8 的大小分块转置。

一个朴素的 8×8 分块可以做到 343 次脱靶，对于不在对角线上的块，A, B 中每一块均脱靶 8 次，已经达到理论最优。考虑在对角线上的块，由于此时 A, B 每一行索引都相同，会导致访问 B 一列时驱逐一行 A，仍存在优化空间。

可以发现直接在 B 中原址转置（即交换 `B[i][j]` 和 `B[j][i]`）是不会出现驱逐的，于是将 A 的每一行通过 8 个辅助变量按照原位置转移到 B 中，然后在 B 中原址转置，即可实现 $8+8$ 的理论脱靶次数。

理论总脱靶次数为 $\frac{32}{8} \times \frac{32}{8} \times (8+8) = 256$ 次，实际由于多记录了一些无关操作，总脱靶次数为 259 次。

2.2 64×64

跟 32×32 相比，64×64 的矩阵索引每 4 行一个循环，8×8 分块中前四行和后四行的索引相同，于是不能再采用朴素的 8×8 分块。

由于一次 `cache` 的读写是以 8 个 `int` 为一块，采用 4×4 分块会因为不能充分利用 `cache` 而增加脱靶数，所以进一步优化要以尽可能填满 `cache` 为原则。

对于对角线两侧的情况，将矩阵分块为如下形式（令 $\mathbb{A}$ 在对角线下方）：

$$\mathbb{A}_{8\times8} = \begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix}, \quad \mathbb{B}_{8\times8} = \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix}, \quad \mathbb{A}^{\mathrm{T}}_{8\times8} = \begin{bmatrix} A_0^{\mathrm{T}} & A_2^{\mathrm{T}} \\ A_1^{\mathrm{T}} & A_3^{\mathrm{T}} \end{bmatrix}$$

要充分利用 `cache`，就不能同时处理一列的两个块。

使用 8 个辅助变量，按照行转移将 $\begin{bmatrix} A_0 & A_1 \end{bmatrix}$ 移动到 $\mathbb{B}$ 中，得 $\begin{bmatrix} B_0 & B_1 \\ A_1^{\mathrm{T}} & A_0^{\mathrm{T}} \end{bmatrix}$。然后单独处理 A_3，提出每一个列向量转置替换 A_0^{T} 的行向量，同时将 A_0^{T} 移动到正确的位置，各需要 4 个辅助变量，操作结束后的矩阵为

$$\begin{bmatrix} A_0^{\mathrm{T}} & B_1 \\ A_1^{\mathrm{T}} & A_3^{\mathrm{T}} \end{bmatrix}$$

然后将 A_2^{T} 转移到 B_1 的位置即可完成转置。一次 $\mathbb{B} = \mathbb{A}^{\mathrm{T}}$ 的过程只处理了一侧，所以需要镜像地对对角线上方的块镜像处理一次。

对于对角线上的情况，可以继续按照原址转置的方法处理，一个形象化的处理过程如下

$$\mathbb{B} \Longrightarrow \begin{bmatrix} A_0 & A_1 \\ B_2 & B_3 \end{bmatrix} \Longrightarrow \begin{bmatrix} A_0^{\mathrm{T}} & A_1^{\mathrm{T}} \\ B_2 & B_3 \end{bmatrix} \Longrightarrow \begin{bmatrix} A_0^{\mathrm{T}} & A_1^{\mathrm{T}} \\ A_2 & A_3 \end{bmatrix} \Longrightarrow \begin{bmatrix} A_0^{\mathrm{T}} & A_1^{\mathrm{T}} \\ A_2^{\mathrm{T}} & A_3^{\mathrm{T}} \end{bmatrix} \Longrightarrow \mathbb{A}^{\mathrm{T}}$$

在交换 $A_1^{\mathrm{T}}, A_2^{\mathrm{T}}$ 时发生了放逐，总脱靶次数为 24 次，利用效率并不算高。

考虑按照对角线两侧的思路，选择 $\mathbb{B}$ 右一个块 $\mathbb{B}'$（此时还未更新到）来作为辅助块装载，这样因为辅助块导致的脱靶会在紧接着的非对角线转置中抵消。

采用相同的思路，有

$$\mathbb{B}\mathbb{B}' \Longrightarrow \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} \begin{bmatrix} B_0' & B_1' \\ A_0^{\mathrm{T}} & A_1^{\mathrm{T}} \end{bmatrix}$$

不能直接交换 $\mathbb{A}, \mathbb{B}$ 的块，采用原址转置来避免放逐

$$\begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} \begin{bmatrix} B_0' & B_1' \\ A_0^{\mathrm{T}} & A_1^{\mathrm{T}} \end{bmatrix} \Longrightarrow \begin{bmatrix} B_0 & B_1 \\ A_2 & A_3 \end{bmatrix} \begin{bmatrix} B_0' & B_1' \\ A_0^{\mathrm{T}} & A_1^{\mathrm{T}} \end{bmatrix} \Longrightarrow \begin{bmatrix} B_0 & B_1 \\ A_2^{\mathrm{T}} & A_3^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} B_0' & B_1' \\ A_0^{\mathrm{T}} & A_1^{\mathrm{T}} \end{bmatrix}$$

然后直接行交换 A_2^{T} 和 A_1^{T}，得到 $\begin{bmatrix} A_0^{\mathrm{T}} & A_2^{\mathrm{T}} \end{bmatrix}$，再行覆盖到 $\mathbb{B}$ 中即可

$$\begin{bmatrix} B_0 & B_1 \\ A_2^{\mathrm{T}} & A_3^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} B_0' & B_1' \\ A_0^{\mathrm{T}} & A_1^{\mathrm{T}} \end{bmatrix} \Longrightarrow \begin{bmatrix} B_0 & B_1 \\ A_1^{\mathrm{T}} & A_3^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} B_0' & B_1' \\ A_0^{\mathrm{T}} & A_2^{\mathrm{T}} \end{bmatrix} \Longrightarrow \mathbb{A}^{\mathrm{T}} \begin{bmatrix} B_0' & B_1' \\ A_0^{\mathrm{T}} & A_2^{\mathrm{T}} \end{bmatrix}$$

这样的脱靶次数为 20 次，多余的 4 次脱靶抵消掉了下一次转置输出部分的脱靶，除了 $(56, 56)$ 的块其余均可采用辅助块的优化。

枚举顺序为先 n 后 m 时，总脱靶次数为 1063 次，改为先 m 后 n 优化到了 1035 次。此顺序的改变只影响到了对角线上的情况，共减少了 7×4 次脱靶，刚好对应了抵消的脱靶数，只有先 m 后 n 的列遍历才能利用上这 4 次脱靶的抵消。

理论总脱靶数为 $\frac{64}{8} \times \frac{64}{8} \times (8+8) + 8 = 1032$ 次，实际总脱靶数为 1035 次。

2.3 61×67

考虑把 61×67 的矩阵尽可能拟合成 4 或者 8 的倍数，测试发现在朴素的分块下，采用 4×17 的分块可以达到最小的脱靶次数，为 1848 次。

由于不规则分块带来的不完全利用，此处难以再用辅助变量进行高效率优化，利用 67 带来的无循环性，分块列数应该为质数，为 17 时效果最好。行数取 $2 \sim 8$ 带来的脱靶数影响低于列数的选择。

2.4 运行结果

```
Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         259
Trans perf 64x64           8.0         8        1035
Trans perf 61x67          10.0        10        1848
          Total points    53.0        53
```