6.25

S t s b

1. 64 24 6 2

2. 1 30 0 2

3. 128 22 7 3

4. 1 29 0 3

5. 32 22 5 5

6. 8 24 3 5

6.29

A.

CTCTCTCTCTCTCTCT, CICI, COCO

B.

Hit? Read value(or unknown)

N -

Y unknown

Y 0xC0

6.33

0x1788 0x1789 0x178A 0x178B

0x16C8 0x16C9 0x16CA 0x16CB

6.37

cache共有256个block，分别位于256个set中，每个block可以放下4个int类型的变量，所有的block可以放下1024个int类型的变量。

当N = 64：

映射关系：a[0][0] ~ a[15][63]、a[16][0] ~ a[31][63]、a[32][0] ~ a[47][63]、a[48][0] ~ a[63][63] 互相重叠。

sumA按照行来读取，所以每四次读取第一次都会miss，即miss rate = 25%。

sumB按照列来读取，所以每一次读取都会发生miss（读取后的block又会被覆盖），即miss rate = 100%。

sumC按照列来读取，但是每次读取后都会按照行再读取一次，所以每四次读取会有两次miss，即miss rate = 50%。

当N = 60

映射关系：a[0][0] ~ a[17][3]、a[17][4] ~ a[34][7]、a[34][8] ~ a[51][11]、a[51][12] ~ a[59][59]互相重叠，其中最后的a[51][12] ~ a[59][59]没有到达cache的尾部。

sumA按照行来读取，所以每四次读取第一次都会miss，即miss rate = 25%。

sumB按照列来读取，这里的情况有些复杂，我写了一个程序来分析：

#include <stdio.h>

#define SIZEOFCACHE 256

#define SIZEOFBLOCK 4

#define N 60

int main()

{

int cache[SIZEOFCACHE];

for (int k = 0; k < SIZEOFCACHE; ++k)

{

cache[k] = -1;

}

int read = 0;

int miss = 0;

for (int j = 0; j < N; ++j)

{

for (int i = 0; i < N; ++i)

{

//read a[i][j]

++read;

int position = i \* N + j;

int need\_start = position/SIZEOFBLOCK;

if (cache[need\_start%SIZEOFCACHE] != need\_start)

{

++miss;

cache[need\_start%SIZEOFCACHE] = need\_start;

}

}

}

printf("%g\n", miss/(double)read);

return 0;

}

输出结果为25%。

C.

将上面程序的循环部分更改为：

for (int j = 0; j < N; j+=2)

{

for (int i = 0; i < N; i+=2)

{

//read a[i][j] a[i+1][j] a[i][j+1] a[i+1][j+1]

++read;

int position = i \* N + j;

int need\_start = position/SIZEOFBLOCK;

if (cache[need\_start%SIZEOFCACHE] != need\_start)

{

++miss;

cache[need\_start%SIZEOFCACHE] = need\_start;

}

++read;

position = (i+1) \* N + j;

need\_start = position/SIZEOFBLOCK;

if (cache[need\_start%SIZEOFCACHE] != need\_start)

{

++miss;

cache[need\_start%SIZEOFCACHE] = need\_start;

}

++read;

position = i \* N + j + 1;

need\_start = position/SIZEOFBLOCK;

if (cache[need\_start%SIZEOFCACHE] != need\_start)

{

++miss;

cache[need\_start%SIZEOFCACHE] = need\_start;

}

++read;

position = (i+1) \* N + j + 1;

need\_start = position/SIZEOFBLOCK;

if (cache[need\_start%SIZEOFCACHE] != need\_start)

{

++miss;

cache[need\_start%SIZEOFCACHE] = need\_start;

}

}

}

输出结果为25%。

6.41

这个cache有16K个block，每个block可以放4个char类型的变量，也就是一个pixel的结构体，即cache总共可以放置16K个结构体。buffer里面一共有480 \* 640 = 300K个结构体，所以映射时会有18个完全重叠的，最后一次重叠3/4.

这个程序按照列来写，每四次写入只有第一次miss（每次都完整利用了一个block，没有读入block的浪费，此时miss rate只取决于block的大小），所以miss rate = 25%。

6.45

这个题要求我们利用第5章和第6章中学到的优化知识。对于第5章，就是减少循环的数据依赖，从而利用流水线并行执行；对于第6章，则是从两个方面（temporary、spatial）利用数据的“本地性”。

void transpose(int \*dst, int \*src, int dim)

{

int i, j;

for (i = 0; i < dim; ++i)

{

for (j = 0; j < dim; ++j)

{

dst[j\*dim + i] = src[i\*dim +j] /\* ! \*/

}

}

}

以上的关键语句中的乘法和加法已经实现了循环之间独立，src也是按照行读入的，但是dst却是按照列读入的，这样没有充分利用每一次读入的block。于是我们想到可不可以每一次读入dst[j\*dim + i]所在的block之后继续写入例如dst[j\*dim + i + 1] dst[j\*dim + i + 2]这样的变量，但是这样有需要src的部分变为src[(i+1)\*dim +j]等等，所以我们现在不仅要“横向”扩展dst，还要“纵向”扩展src，其实这是一种叫做blocking的技术，即每次读入一块数据，对此块数据完全利用后抛弃，然后读取下一个块。可以参考csapp网上给的注解：MEM:BLOCKING — Using blocking to increase temporal locality

设我们的数据块的宽度是B，由于我们要对两个数组进行读写操作，所以2B^2 < C（其中C是cache的容量），在此限制下B尽可能取大。

#define B chunkdatas\_length\_of\_side

void faster\_transpose(int \*dst, int \*src, int dim)

{

long limit = dim \* dim;

for (int i = 0; i < dim; i += B)

{

for (int j = 0; j < dim; j += B)

{

/\* Using blocking to improve temporal locality \*/

for (int k = i; k < i+B; ++k)

{

for (int l = j; l < j+B; ++l)

{

/\* independent calculations \*/

int d = l\*dim + k;

int s = k\*dim + l;

if (s < limit && d < limit)

{

dst[d] = src[s]

}

}

}

}

}

}

8.9

AB N

AC Y

AD Y

BC Y

BD Y

CD Y

8.13

保证x=4在x=3之前即可（拓扑排序），有三种情况：

A.

x=2 x=4 x=3

B.

x=4 x=2 x=3

C.

x=4 x=3 x=2

"x=4" "x=3"

+-------> printf("%d\n", ++x) +---> printf("%d\n", --x) +-->

|

|

|

+---------+-------> printf("%d\n", --x) +---------------------------->

x = 3 Fork

"x=2"

8.17

假设子进程正常退出；构成拓扑排序即可，有三种情况：

A.

Hello 1 Bye 0 2 Bye

B.

Hello 1 0 Bye 2 Bye

C.

Hello 0 1 Bye 2 bye

8.21

满足拓扑排序即可，两种情况：

A.

abc

B.

bac

+----->printf("a")+-------+

| |

| v

+---+----->printf("b")+--->waitpid+--->printf("c")+-->

fork

8.25

倒计时可以用alarm实现，其到指定时间后会raise一个SIGALRM信号， man 2 alarm部分描述：

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

alarm() arranges for a SIGALRM signal to be delivered to the calling process in seconds seconds.

If seconds is zero, any pending alarm is canceled.

In any event any previously set alarm() is canceled.

我们收到这个信号后，就要想办法终止等待中的读入并返回NULL。其中一个办法是使用setjmp.h ，我们在第一次使用setjmp(buf)时进入正常的读入（此时setjmp返回值为0），但当信号出现（时间截止），信号处理函数就会longjmp(buf，1) （此时setjmp返回值为1），根据返回值的不同，这时我们便进入return NULL语句。

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <setjmp.h>

#define TIMEOUT ((unsigned int)5)

#define SIZEOFBUF 1024

jmp\_buf buf;

void SIGALRM\_handler(int signum)

{

longjmp(buf, 1);

}

char \*tfgets(char \*s, int size, FILE \*stream)

{

if (signal(SIGALRM, SIGALRM\_handler) == SIG\_ERR)

{

perror("Failed to install SIGALRM\_handler");

return NULL;

}

else

{

alarm(TIMEOUT); /\* raise SIGALRM after TIMEOUT seconds \*/

}

if (!setjmp(buf))

{

return fgets(s, size, stream);

}

else /\* longjmp from SIGALRM\_handler \*/

{

return NULL;

}

}

int main(int argc, char const \*argv[])

{

char temp\_bufer[SIZEOFBUF];

char \*result = tfgets(temp\_bufer, SIZEOFBUF, stdin);

if (result)

{

printf("Input : %s\n", result);

}

else

{

fprintf(stderr, "Time out!\n");

}

return 0;

}

A.

000000001 000000

B.

Parameter Value

VPN 0x01

TLB index 0x01

TLB tag 0x00

TLB hit? N

Page fault? Y

PPN ---

9.17

/\* Global variables \*/

static char \*heap\_listp = 0; /\* Pointer to first block \*/

static char \*rover; /\* Next fit rover \*/

static void \*find\_fit(size\_t asize);

static void \*coalesce(void \*bp);

/\*

\* mm\_init - Initialize the memory manager

\*/

/\* $begin mminit \*/

int mm\_init(void)

{

/\* Create the initial empty heap \*/

if ((heap\_listp = mem\_sbrk(4\*WSIZE)) == (void \*)-1) //line:vm:mm:begininit

return -1;

PUT(heap\_listp, 0); /\* Alignment padding \*/

PUT(heap\_listp + (1\*WSIZE), PACK(DSIZE, 1)); /\* Prologue header \*/

PUT(heap\_listp + (2\*WSIZE), PACK(DSIZE, 1)); /\* Prologue footer \*/

PUT(heap\_listp + (3\*WSIZE), PACK(0, 1)); /\* Epilogue header \*/

heap\_listp += (2\*WSIZE); //line:vm:mm:endinit

/\* $end mminit \*/

rover = heap\_listp;

/\* $begin mminit \*/

/\* Extend the empty heap with a free block of CHUNKSIZE bytes \*/

if (extend\_heap(CHUNKSIZE/WSIZE) == NULL)

return -1;

return 0;

}

/\* $end mminit \*/

/\*

\* coalesce - Boundary tag coalescing. Return ptr to coalesced block

\*/

/\* $begin mmfree \*/

static void \*coalesce(void \*bp)

{

size\_t prev\_alloc = GET\_ALLOC(FTRP(PREV\_BLKP(bp)));

size\_t next\_alloc = GET\_ALLOC(HDRP(NEXT\_BLKP(bp)));

size\_t size = GET\_SIZE(HDRP(bp));

if (prev\_alloc && next\_alloc) { /\* Case 1 \*/

return bp;

}

else if (prev\_alloc && !next\_alloc) { /\* Case 2 \*/

size += GET\_SIZE(HDRP(NEXT\_BLKP(bp)));

PUT(HDRP(bp), PACK(size, 0));

PUT(FTRP(bp), PACK(size,0));

}

else if (!prev\_alloc && next\_alloc) { /\* Case 3 \*/

size += GET\_SIZE(HDRP(PREV\_BLKP(bp)));

PUT(FTRP(bp), PACK(size, 0));

PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 0));

bp = PREV\_BLKP(bp);

}

else { /\* Case 4 \*/

size += GET\_SIZE(HDRP(PREV\_BLKP(bp))) +

GET\_SIZE(FTRP(NEXT\_BLKP(bp)));

PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 0));

PUT(FTRP(NEXT\_BLKP(bp)), PACK(size, 0));

bp = PREV\_BLKP(bp);

}

/\* $end mmfree \*/

/\* Make sure the rover isn't pointing into the free block \*/

/\* that we just coalesced \*/

if ((rover > (char \*)bp) && (rover < NEXT\_BLKP(bp)))

rover = bp;

/\* $begin mmfree \*/

return bp;

}

/\* $end mmfree \*/

/\*

\* find\_fit - Find a fit for a block with asize bytes

\*/

static void \*find\_fit(size\_t asize)

{

/\* Next fit search \*/

char \*oldrover = rover;

/\* Search from the rover to the end of list \*/

for ( ; GET\_SIZE(HDRP(rover)) > 0; rover = NEXT\_BLKP(rover))

if (!GET\_ALLOC(HDRP(rover)) && (asize <= GET\_SIZE(HDRP(rover))))

return rover;

/\* search from start of list to old rover \*/

for (rover = heap\_listp; rover < oldrover; rover = NEXT\_BLKP(rover))

if (!GET\_ALLOC(HDRP(rover)) && (asize <= GET\_SIZE(HDRP(rover))))

return rover;

return NULL; /\* no fit found \*/

}

/\* $end mmfirstfit \*/