Lab report #7

CSED 20170302 Kim Dae Hui

이번 랩은 cache lab이다. Part A와 B로 크게 두 부분으로 나누어져있는데 part A에서 먼저 캐시 시뮬레이터를 만들어서 입력받은 만큼의 캐시와 비슷한 형태의 2차원 배열메모리를 동적할당하여 이를 캐시처럼 생각하고 사용하게 한다. Part B에서는 이 데이터를 transpose해주는데 이 때 miss를 최소화하는 방식으로 blocking하는 것이다.

Part A(csim.c)

|  |
| --- |
| struct cacheParameter { // 입력받는 캐시의 기본정보를 확인하는 것  int s;  int b;  int E;  int S;  int B;  int hits;  int misses;  int evicts;  };  struct setLine { //한 라인 구조  int latestUsed;  int valid;  memoryAddress tag;  char \*block;  };  struct cacheSet { // 한 셋 구조  struct setLine \*lines;  };  struct cache { // 한 캐쉬 구조  struct cacheSet \*sets;  }; |

우선 .c파일에서 사용할 기본 struct들이다. cacheParameter에서 입력값들을 받아들일 것이고 setline, cacheSet, cache는 각각 line,setcache를 표현한다. Cache와 cacheSet은 후에 동적할당 할 것이기 때문에 포인터로 element를 주어 배열화 할 수 있도록 해놓는다.

|  |
| --- |
| struct cache initCache(long long numberOfSets, int numberOfLines, long long sizeOfBlock)  { //입력받은 값들을 바탕으로 캐시를 메모리안에 할당한다.  struct cache newCache;  struct cacheSet set;  struct setLine line;  int indexOfSet;  int indexOfLine;  newCache.sets = (struct cacheSet\*) malloc(sizeof(struct cacheSet) \* numberOfSets); =  for (indexOfSet = 0; indexOfSet < numberOfSets; indexOfSet++)  { //0의값으로 초기화해주는 단계이다  set.lines = (struct setLine \*) malloc(sizeof(struct setLine) \* numberOfLines);  newCache.sets[indexOfSet] = set;  for (indexOfLine = 0; indexOfLine < numberOfLines; indexOfLine++)  {  line.latestUsed = 0;  line.valid = 0;  line.tag = 0;  set.lines[indexOfLine] = line;  }  }  return newCache;  } |

입력 받은 후에 initCache함수를 통해서 동적할당을 시작하는 부분이다. 0으로 시작하여 사용자가 입력한 값 만큼의 cache 데이터를 동적할당 해준다.(initializing도 해준다.)

|  |
| --- |
| void cleanFunction(struct cache myCache, long long numberOfSets, int numberOfLines, long long sizeOfBlock)  {  //메모리를 초기화해주는 함수  int indexOfSet;  for (indexOfSet = 0; indexOfSet < numberOfSets; indexOfSet++)  {  struct cacheSet set = myCache.sets[indexOfSet];  if (set.lines != NULL) {  free(set.lines);  }  }  if (myCache.sets != NULL) {  free(myCache.sets);  }  } |

이 프로그램에서는 사실 메모리에 2차원 배열을 사용해서 상관없긴 하지만 실제로 캐시는 코스트가 매우 큰 저장장치 이므로 휘발성 메모리를 사용한다. 따라서 사용한 후 기존의 할당을 해제하고 캐시를 효율적으로 사용하도록 할당해제를 해주는 함수를 만든다.

|  |
| --- |
| int detectEmptyLine(struct cacheSet exampleSet, struct cacheParameter exampleParameter) {  // vaild값을 조회해서 empty한 line인지 아닌지를 판별한다.  int numberOfLines = exampleParameter.E;  int index;  struct setLine line;  for (index = 0; index < numberOfLines; index++) {  line = exampleSet.lines[index];  if (line.valid == 0) {  return index;  }  }  return -1;  } |

valid값을 탐지하여 line에 빈 값이 있는지 없는지를 체크하는 함수(evict여부를 확인할 때 full cache인지 확인하는데 쓰인다.)

|  |
| --- |
| int detectEvictLine(struct cacheSet exampleSet, struct cacheParameter exampleParameter, int \*usedLines) {  // eviction을 판명하기 위해서 변수들을 초기화한 이후에 최근 사용된 라인을 확인한다.  int numberOfLines = exampleParameter.E;  int maxFreqUsage = exampleSet.lines[0].latestUsed;  int minFreqUsage = exampleSet.lines[0].latestUsed;  int indexOfLine;  for (indexOfLine = 1; indexOfLine < numberOfLines; indexOfLine++) {  if (minFreqUsage > exampleSet.lines[indexOfLine].latestUsed) {  minFreqUsage\_index = indexOfLine;  minFreqUsage = exampleSet.lines[indexOfLine].latestUsed;  }  if (maxFreqUsage < exampleSet.lines[indexOfLine].latestUsed) {  maxFreqUsage = exampleSet.lines[indexOfLine].latestUsed;  }  }  usedLines[0] = minFreqUsage;  usedLines[1] = maxFreqUsage;  return minFreqUsage\_index;  } |

LRU를 찾아주기 위한 함수이다. 초기화를 거친 후에 간단한 알고리즘으로 최소 최대를 비교하여 이를 usedLine에 담아준다.

|  |
| --- |
| struct cacheParameter accessTheCacheData(struct cache myCache, struct cacheParameter exampleParameter, memoryAddress address) {  int indexOfLine;  int checkFullCache = 0;  int numberOfLines = exampleParameter.E;  int previousHit = exampleParameter.hits;  int tagSize = (64 - (exampleParameter.s + exampleParameter.b)); // t = m-s-b  //변수들 선언, 초기화후에 입력받은 주소에서 필요한 값들을 추출해낸다.  memoryAddress inputTag = address >> (exampleParameter.s + exampleParameter.b);  unsigned long long temp = address << (tagSize);  unsigned long long indexOfSet = temp >> (tagSize + exampleParameter.b);  struct cacheSet exampleSet = myCache.sets[indexOfSet];  for (indexOfLine = 0; indexOfLine < numberOfLines; indexOfLine++) {  if (exampleSet.lines[indexOfLine].valid) {  if (exampleSet.lines[indexOfLine].tag == inputTag) {  exampleSet.lines[indexOfLine].latestUsed++;  exampleParameter.hits++;  // 발리드하고 태그값이 같으면 hit  }  else if (!(exampleSet.lines[indexOfLine].valid) && (checkFullCache)) {  checkFullCache = 0; // empty값이 있으면 Full하지 않은것을 체크  }  //  }  if (previousHit == exampleParameter.hits) {  exampleParameter.misses++; // hit되지 않고 나오면 miss  }  else {  return exampleParameter;  }  int \*usedLines = (int\*)malloc(sizeof(int) \* 2);  int minFreqUsage\_index = detectEvictLine(exampleSet, exampleParameter, usedLines);  if (checkFullCache) //full cache가 hit되지 않을 경우 evict  {  exampleParameter.evicts++;  exampleSet.lines[minFreqUsage\_index].tag = inputTag;  exampleSet.lines[minFreqUsage\_index].latestUsed = usedLines[1] + 1;  }  else //그렇지 않으면은 empty에 tag적기  {  int empty\_index = detectEmptyLine(exampleSet, exampleParameter);  exampleSet.lines[empty\_index].tag = inputTag;  exampleSet.lines[empty\_index].valid = 1;  exampleSet.lines[empty\_index].latestUsed = usedLines[1] + 1;  }  free(usedLines);  return exampleParameter;  } |

우선 입력값들을 구분해야한다. 따라서 위 함수의

int tagSize = (64 - (exampleParameter.s + exampleParameter.b)); // t = m-s-b

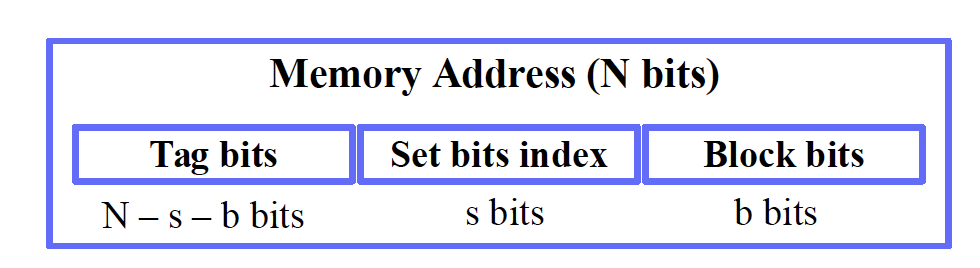
//변수들 선언, 초기화후에 입력받은 주소에서 필요한 값들을 추출해낸다.

memoryAddress inputTag = address >> (exampleParameter.s + exampleParameter.b);

unsigned long long temp = address << (tagSize);

unsigned long long indexOfSet = temp >> (tagSize + exampleParameter.b);

부분에서 메모리주솟값에서 tag를 우선 추출해내고 s와 b그리고 setindex도 추출해낸다. 이는



와 같은 메모리 어드레스 구조를 따라 구분해낸 것이다. 그 후 valid 여부를 확인하고 tag를 확인하여 hit여부를 검사, hit하지 않을 시 miss, 그리고 full cache에서 hit, miss하지 않은 경우 evict를 일으키도록 flow를 짠다.

와 같이 함수를 구성하였고 main함수에서는 arg를 입력받아서 파싱하여 해당하는 data들을 cacheParameter에 저장하고 실행하는 switch문을 사용했다.

Part B(trans.c)

|  |
| --- |
| char transpose\_32\_32\_desc[] = "Transpose the 32 x 32 matrix";  void transpose\_32\_32(int M, int N, int A[N][M], int B[M][N]) {    int BLOCK\_SIZE;  int rowIndex;  int colIndex;  int blockedRowIndex;  int blockedColIndex;  int eBlockDiagl;  int iBlockDiagl;  BLOCK\_SIZE = 8;  //32 32인데 int가 4byte이므로 block size는8인 line이여야 한다.  for (colIndex = 0; colIndex < M; colIndex += BLOCK\_SIZE) {  for (rowIndex = 0; rowIndex < N; rowIndex += BLOCK\_SIZE) {  //transpose를 해당위치에 맞도록 해준다.  for (blockedRowIndex = rowIndex; blockedRowIndex < rowIndex +  BLOCK\_SIZE; ++blockedRowIndex) {  for (blockedColIndex = colIndex; blockedColIndex < colIndex  + BLOCK\_SIZE; ++blockedColIndex) {  if (blockedRowIndex != blockedColIndex) {  B[blockedColIndex][blockedRowIndex] =  A[blockedRowIndex][blockedColIndex];  }  else {  eBlockDiagl = A[blockedRowIndex][blockedColIndex];  iBlockDiagl = blockedRowIndex;  }  }  //diagonal한 element에 대한 처리  if (colIndex == rowIndex) {  B[iBlockDiagl][iBlockDiagl] = eBlockDiagl;  }  }  }  }  } |

1. ./test-trans –M 32 –N 32

32x32 size의 int형 2차원 배열(여기서는 cache라고 치자.)을 transpose하는 과정을 해야한다. 우선 int가 4byte이므로 blocksize를 8으로 생각한다. Block size가 8인 경우에는 한 블럭안에서 한 index의 실행이 일어날 경우 cold miss만 발생하기 때문에 miss가 적게 발생한다. Row와 col을 block size에 맞게 전개해주고 이 원소들을 transpose할 수 있도록 반복문 안에 넣는다. Row 와 col값들이 transpose하게 들어가도록 해주고 diagonal한 원소들에 대한 처리를 아래에서 해주면 된다.

|  |
| --- |
| char transpose\_64\_64\_desc[] = "Transpose the 64 x 64 matrix";  void transpose\_64\_64(int M, int N, int A[N][M], int B[M][N]) {  int rowIndex, colIndex;  int reg1, reg2, reg3, reg4, reg5, reg6, reg7, reg8;  //64 64는 레지스터(실제로는메모리지만 레지스터로 생각하는 것)에 값을 저장한 후 그 값을 취하도록한다.  int ct;  for (rowIndex = 0; rowIndex < N; rowIndex += 8) {  for (colIndex = 0; colIndex < M; colIndex += 8) {  //8개단위로 레지스터를 사용하기 때문에 index를 8씩 조절해준다.  for (ct = 0; ct < 8; ct++) {  reg1 = A[colIndex + ct][rowIndex];  reg2 = A[colIndex + ct][rowIndex + 1];  reg3 = A[colIndex + ct][rowIndex + 2];  reg4 = A[colIndex + ct][rowIndex + 3];  //ct값을 이용하여 첫 miss값을 제외해줌  if (0 == ct) {  reg5 = A[colIndex + ct][rowIndex + 4];  reg6 = A[colIndex + ct][rowIndex + 5];  reg7 = A[colIndex + ct][rowIndex + 6];  reg8 = A[colIndex + ct][rowIndex + 7];  }  B[rowIndex][colIndex + ct] = reg1;  B[rowIndex][colIndex + ct + 64] = reg2;  B[rowIndex][colIndex + ct + 128] = reg3;  B[rowIndex][colIndex + ct + 192] = reg4;  }  for (ct = 7; ct > 0; ct--) {  //마지막 값들에 대한 처리  reg1 = A[colIndex + ct][rowIndex + 4];  reg2 = A[colIndex + ct][rowIndex + 5];  reg3 = A[colIndex + ct][rowIndex + 6];  reg4 = A[colIndex + ct][rowIndex + 7];  B[rowIndex + 4][colIndex + ct] = reg1;  B[rowIndex + 4][colIndex + ct + 64] = reg2;  B[rowIndex + 4][colIndex + ct + 128] = reg3;  B[rowIndex + 4][colIndex + ct + 192] = reg4;  }  B[rowIndex + 4][colIndex] = reg5;  B[rowIndex + 4][colIndex + 64] = reg6;  B[rowIndex + 4][colIndex + 128] = reg7;  B[rowIndex + 4][colIndex + 192] = reg8;  }  }  } |

2. ./test-trans –M 64 –N 64

이 배열의 경우에는 64byte로 크기가 크기 때문에 block size를 더욱 크게 해주어야 miss를 줄일 수 있다는 것을 짐작할 수 있다. 하지만 64x64이기 때문에 단순히 그런 방법으로는 사용할 수 없다. 따라서 레지스터를 이용하기로 하고(실제로는 그냥 변수지만 레지스터로 생각하는 것임) 한 index안에서 8개만큼의 수를 저장할 수 있도록 한다. 지역변수들의 값을 바꾸어 transpose하게 대입하는 방식을 사용하여야 conflict miss를 줄일 수 있다. 따라서 reg1~reg8까지의 지역변수를 temp처럼 사용하는 방식으로 한다.

|  |
| --- |
| char transpose\_61\_67\_desc[] = "Transpose the 61 x 67 matrix";  void transpose\_61\_67(int M, int N, int A[N][M], int B[M][N]) {  int rowIndex, colIndex;  int blockedRowIndex, blockedColIndex;  int temp;  //이 경우는 m=61인 case이기 때문에 32x32와 같이 조정한다.(단 67이 존재하기 때문에 blocksize=16으로함)  for (rowIndex = 0; rowIndex < N; rowIndex += 16) {  for (colIndex = 0; colIndex < M; colIndex += 4) {  for (blockedRowIndex = rowIndex; (blockedRowIndex < rowIndex + 16) &&  (blockedRowIndex < N); ++blockedRowIndex) {  for (blockedColIndex = colIndex; (blockedColIndex < colIndex + 4)  && (blockedColIndex < M); ++blockedColIndex) {  if (blockedRowIndex - rowIndex ==  blockedColIndex - colIndex) {  temp = A[blockedRowIndex][blockedColIndex];  }  else {  B[blockedColIndex][blockedRowIndex] =  A[blockedRowIndex][blockedColIndex];  }  }  for (blockedColIndex = colIndex; (blockedColIndex <  colIndex + 4) && (blockedColIndex < M); ++blockedColIndex) {  if (blockedRowIndex - rowIndex ==  blockedColIndex - colIndex) {  B[blockedColIndex][blockedRowIndex] = temp;  }  }  }  }  }  } |

3. ./test-trans –M 61 –N 64

첫번 째 경우와 비슷하게 작성한다. 하지만 N=67이기 때문에 block size를 16으로 해주어야 conflict miss를 줄일 수 있다. 그리고 이것은 정사각형이 아니기 때문에 index를 넘어가 capacity miss가 발생할 수 있는데 이를 방지해주는 부분도 삽입한다.