CS 2021: Practice Exam 3 SOLUTION
Spring 2019
University of Minnesota

Exam period: 20 minutes
Points available: 40

Problem 1 (10 pts): Below is an initial memory/cache configuration along with several memory load operations. Indicate whether these load operations result in cache hits or misses and show the state of the cache after these loads complete.

```
-------------------- SOLUTION --------------------
MAIN MEMORY                          DIRECT-MAPPED Cache, 8-byte lines
| Addr | Addr Bits       | Value |   4 Sets, 8-bit Address = 3-bit tag
|------+-----------------+-------|
|   10 | 000  10 000     |    10 |   INITIAL CACHE STATE
|   14 | 000  10 100     |    11 |   |     |   |     | Blocks/Line |
|   18 | 000  11 000     |    12 |   | Set | V | Tag | 0-3   4-7   |
|   1C | 000  11 100     |    13 |   |-----+---+-----+-------------|
|   20 | 001  00 000     |    20 |   |  00 | 1 | 010 | 200   201   |
|   24 | 001  00 100     |    21 |   |  01 | 1 | 001 | 22    23    |
|   28 | 001  01 000     |    22 |   |  10 | 1 | 000 | 10    11    |
|   2C | 001  01 100     |    23 |   |  11 | 0 |   - | -           |
|   30 | 001  10 000     |   100 |
|   34 | 001  10 100     |   101 |   HITS OR MISSES?
|   38 | 001  11 000     |   102 |   | OPEARTION    | HIT/MISS? |
|   3C | 001  11 100     |   103 |   |--------------+-----------|
|   40 | 010  00 000     |   200 |   | 1. Load 0x48 | Miss      |
|   44 | 010  00 100     |   201 |   | 2. Load 0x4C | Hit       |
|   48 | 010  01 000     |   202 |   | 3. Load 0x24 | Miss      |
|   4C | 010  01 100     |   203 |
|------+-----------------+-------|   FINAL CACHE STATE
|      | Tag Set Offset  |       |   |     |   |     | Blocks/Line |          |
                                     | Set | V | Tag | 0-3   8-7   | Changed? |
                                     |-----+---+-----+-------------+----------|
                                     |  00 | 1 | 000 | 20    21    | ***      |
                                     |  01 | 1 | 010 | 202   203   | ***      |
                                     |  10 | 1 | 000 | 10    11    |          |
                                     |  11 | 0 |   - | -           |          |
```

Problem 2 (5 pts): Pyra Midmem read in a free online blog post "Memory for Morons" that there is no need to invest much money in buying RAM. Instead, one can configure the operating system's virtual memory system to use disk space as main memory leading to a much less expensive computer with a seemingly large memory. Pyra is quite excited about this as some programs she wants to execute fast need a lot of main memory and it would be nice to save some cash. Advise her on any risks or performance drawbacks she may encounter using such an approach.

SOLUTION: Disks are many orders of magnitude slower than the DRAM that is typically used for main memory. Disks are near the bottom of the memory pyramid offering many gigabytes of storage per dollar at the expense of speed. If Pyra needs speed, she is better off investing in more DRAM as her system will crawl if it attempts to use disk for main memory. In addition, she should consider getting a CPU with a large cache which is more expensive but even faster than DRAM.

Problem 3 (15 pts): Nearby is the definition for `base_scalvec()` which scales a vector by multiplying each element by a number. Write an optimized version of this function in the space provided. Mention in comments why you performed certain transformations.

```
int vget(vector_t vec, int idx){
  return vec.data[idx];
}
void vset(vector_t vec, int idx, int x){
  vec.data[idx] = x;
}
void base_scalevec(vector_t *vec, int *scale){
  int sc = *scale;
  vector_t v = *vec;
  for(int i=0; i < v.len; i+=2){
    int cur = vget(v,i);
    int new = cur * sc;
    vset(v,i,new);

    int cur2 = vget(v,i+1);
    int new2 = cur2 * sc;
    vset(v,i+1,new2);
  }
}
```

SOLUTION

```
void opt_scalevec(vector_t *vec, int *scale){
  // locals to avoid memory access
  int *data = vec->data, len = vec->len;
  int scal = (*scale), i;
  // unroll x2 with duplicate vars to
  // enable pipelining
  for(i=0; i < len-2; i+=2){
    // no function calls - inline bodies
    // to imrpove register use
    int cur0 = data[i+0];
    int new0 = cur0 * scal;
    data[i+0] = new0;
    int cur1 = data[i+1];
    int new1 = cur1 * scal;
    data[i+1] = new1;
  }
  // cleanup loop
  for(; i<len; i++){
    int cur0 = data[i+0];
    int new0 = cur0 * scal;
    data[i+0] = new0;
  }
}
```

Problem 4 (10 pts): Examine the two functions below which add elements of a row or column vector to all corresponding rows or columns of a matrix. Consider the benchmark timing of these two provided.

1. Explain which of these two functions is faster and **why**.
2. Suggest a way to increase the speed of the slower function with only moderate changes to the code.

```
// add given row to each row of mat
void matrix_addrow_vec(matrix_t mat,
                       vector_t row) {
  for(int i=0; i<mat.rows; i++){
    for(int j=0; j<mat.cols; j++){
      int elij = MGET(mat,i,j);
      int vecj = VGET(row,j);
      MSET(mat,i,j, elij + vecj);
    }
  }
}
// add given col to each column of mat
void matrix_addcol_vec(matrix_t mat,
                       vector_t col) {
  for(int j=0; j<mat.cols; j++){
    for(int i=0; i<mat.rows; i++){
      int elij = MGET(mat,i,j);
      int veci = VGET(col,i);
      MSET(mat,i,j, elij + veci);
    }
  }
}
// BENCHMARK TIMING:
//   SIZE     addrow     addcol
//    512 2.9040e-03 5.5230e-03
//   1024 5.9290e-03 1.3160e-02
//   2048 1.3809e-02 9.9269e-02
//   4096 5.0853e-02 3.6760e-01
//   8192 2.0867e-01 1.4719e+00
```

SOLUTION: The addrow() version is clearly faster than addcol() at all sizes and this disparity increases as the sizes of the matrices go up. At the largest size, addrow() takes aboute 0.2 seconds while addcol() takes 1.5 seconds, a seven-fold difference.

The reason is due to the layout of the matrix favoring traversal of rows: each row is contiguous in memory which means loading an element will bring nearby elements in the row into cache. This speeds up their access subsequently. The column version jumps non-contiguously through memory getting much less benefit from cache.

Re-writing addcol() to move across rows instead would greatly improve its memory access pattern leading to greater efficiency. This would involve inverting the inner and outer loops to `for(i) / for(j)` *as in the row version. This along with slight modifications to the setting would yield speedups*