AW #5

6.41 Given:

1) sizeof(char) = 1
2) sizeof(int) = 4
3) buffer begins at memory address 0
4) cache is initially empty
5) memory accesses are to the entries of the array buffer.
6) cache has 4-byte lines
7) 640 x 480 array of pixels (2-d array)

Miss Rate = 1 - Hit Rate

Calculating Hit Rate

First, we must calculate the size of our pixel struct. Since we have four char elements in our struct, we know that our struct is 4 bytes long.

Next, we visualize the pixel buffer below. We know that our pixel buffer is a 2-dimensional array of size 480 x 640. Thus, the array appears as follows, where each p_i is 4 bytes:

$p_{0,0}, p_{1,0}, p_{2,0}, \ldots, p_{477,0}, p_{478,0}, p_{479,0}, p_{480,0}$

$p_{0,1}, p_{0,1}, p_{0,2}, \ldots, p_{0,477}, p_{0,478}, p_{0,479}, p_{0,480},$

where each p_i = [r | g | b | a] (1b, 1b, 1b, 1b)

Since we know that the cache is initially empty we know that there will be a miss whenever we will access the red in the struct. However, the subsequent 3 accesses of green, blue, and a will be hits because of spatial locality and the fact that arrays are contiguously stored in memory, as well as the cache is 4-bytes long. This pattern continues at a constant rate for every p_i. Therefore, the hit rate of our cache simplifies to 3/4.

Calculating Hit Rate

Miss Rate = 1 - Hit Rate

$= 1 - \frac{3}{4} = \boxed{\frac{1}{4}}$

6.45 One optimization we could perform on the transpose function is loop unrolling. The idea behind loop unrolling is the fact that we should reduce loop iterations by increasing work per iteration. The effect of loop unrolling is that it makes our program run faster because it reduces the number of iterations and enables the program to exploit instruction-level parallelism. In this case, the work being done would be the swiping of columns and rows. Therefore, our program becomes:

```
void transpose(int *dst, int *src, int dim){
    int i,j;
    for (i=0; i<dim; i++){
        for (j=0; j<dim/4; j+=4) {
            dst[(j)*dim+i] = src[i*dim+(j)];
            dst[(j+1)*dim+i] = src[i*dim+(j+1)];
            dst[(j+2)*dim+i] = src[i*dim+(j+2)];
            dst[(j+3)*dim+i] = src[i*dim+(j+3)];
        }
    }
}
```

Another optimization, we could make is tiling. Tiling would exploit the fact that the transpose is inherently accessing within a temporal local setting. Therefore, we could split our tiles into 8x8 tiles in order to exploit the benefit of tiling.

```
void transpose(int *dst, int *src, int dim){
    int i,j
    int tile = (dim/8);
    for (i=0; i<dim; i++){
        for (j=0; j<tile; j+=8){
            dst[(j)*dim+i] = src[i*dim+(j)];
            dst[(j+1)*dim+i] = src[i*dim+(j+1)];
            dst[(j+2)*dim+i] = src[i*dim+(j+2)];
```

```
dst[(j+3)*dim+i] = src[i*dim+(j+3)];
dst[(j+4)*dim+i] = src[i*dim+(j+4)];
dst[(j+5)*dim+i] = src[i*dim+(j+5)];
dst[(j+6)*dim+i] = src[i*dim+(j+6)];
dst[(j+7)*dim+i] = src[i*dim+(j+7)];
}
}
}
```