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.

<table>
<thead>
<tr>
<th>Addr</th>
<th>Addr Bits</th>
<th>Value</th>
<th>HIT/MISS?</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>000 10 000</td>
<td>10</td>
<td>Miss</td>
</tr>
<tr>
<td>14</td>
<td>000 10 100</td>
<td>11</td>
<td>Hit</td>
</tr>
<tr>
<td>18</td>
<td>000 11 000</td>
<td>12</td>
<td>Hit</td>
</tr>
<tr>
<td>1C</td>
<td>000 11 100</td>
<td>13</td>
<td>Hit</td>
</tr>
<tr>
<td>20</td>
<td>001 00 000</td>
<td>20</td>
<td>Miss</td>
</tr>
<tr>
<td>24</td>
<td>001 00 100</td>
<td>21</td>
<td>Hit</td>
</tr>
<tr>
<td>28</td>
<td>001 01 000</td>
<td>22</td>
<td>Hit</td>
</tr>
<tr>
<td>2C</td>
<td>001 01 100</td>
<td>23</td>
<td>Hit</td>
</tr>
<tr>
<td>30</td>
<td>001 10 000</td>
<td>100</td>
<td>Hit</td>
</tr>
<tr>
<td>34</td>
<td>001 10 100</td>
<td>101</td>
<td>Hit</td>
</tr>
<tr>
<td>38</td>
<td>001 11 000</td>
<td>102</td>
<td>Hit</td>
</tr>
<tr>
<td>3C</td>
<td>001 11 100</td>
<td>103</td>
<td>Hit</td>
</tr>
<tr>
<td>40</td>
<td>010 00 000</td>
<td>200</td>
<td>Miss</td>
</tr>
<tr>
<td>44</td>
<td>010 00 100</td>
<td>201</td>
<td>Hit</td>
</tr>
<tr>
<td>48</td>
<td>010 01 000</td>
<td>202</td>
<td>Miss</td>
</tr>
<tr>
<td>4C</td>
<td>010 01 100</td>
<td>203</td>
<td>Hit</td>
</tr>
</tbody>
</table>

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.

```c
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){
    for(int i=0; i < vec->len; i++){
        int cur = vget(*vec,i);
        int new = cur * (*scale);
        vset(*vec,i,new);
    }
}
```

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.

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 about 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.

```c
void opt_scalevec(vector_t *vec, int *scale){
// locals to avoid memory access
int *data = vec->data, len = vec->len;
int scal = (*scale), i;
for(i=0; i < len-2; i+=2){
    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;
}
}
```