6.823 Computer Architecture	DUE: 3/13/2002
Timothy Danford, Kilian Pohl, Bryan Russell	
Problem Set 3	*Instructors: Krste Asanovic, Srinivas Devadas*

1 Problem 1

a

Loop A incurs a cache miss approximately $\frac{1}{4}$ of its memory accesses. Loop B never hits the contents of the cache. This translates into 256 cache misses in Loop A and 1024 cache misses in Loop B. The reason is the combination of the row-major ordering of the matrix A and the direct-mapped nature of the cache. Each row of the matrix contains 4 cache lines worth of data (4 consecutive lines). When loop B sums the matrix, it sums down the columns. Therefore, as each entry in the matrix is touched, cache lines are loaded at positions in the cache that are multiples of 4. After every 16 rows of the matrix, loop B starts loading new cache lines in the place of the ones currently in place. In other words, loop B uses only $\frac{1}{4}$ of the lines available in the cache. Loop A, on the other hand, loads the cache lines sequentially and accesses all four words within a line before accessing the next line. Only 1 in 4 memory accesses are misses when the length of a cache line is exceeded.

b

	1B Table 1-1: Matrix Multiply Without Blocking									
	Line 0		Line 1		Line 2		Line 3		Line 4	
0	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
1	$Y[0][0]$	$Y[0][1]$								
2			$Z[0][0]$	$Z[0][1]$						
3		HIT								
4					$Z[1][0]$	$Z[1][1]$				
5							$Y[0][2]$	$Y[0][3]$		
6									$Z[2][0]$	$Z[2][1]$
7								HIT		
8			$Z[3][0]$	$Z[3][1]$						
9	$X[0][0]$	$X[0][1]$								
10					$Y[0][0]$	$Y[0][1]$				
11									$Z[0][0]$	$Z[0][1]$
12						HIT				
13							$Z[1][0]$	$Z[1][1]$		
14			$Y[0][2]$	$Y[0][3]$						
15	$Z[2][0]$	$Z[2][1]$								
16				HIT						
17									$Z[3][0]$	$Z[3][1]$
18					$X[0][0]$	$X[0][1]$				

14 misses from 18 memory accesses means that the matrix multiply without blocking suffers from a $\frac{14}{18} = 77.78\%$ miss rate.

	1B Table 1-2: Matrix Multiply With Blocking									
	Line 0		Line 1		Line 2		Line 3		Line 4	
0	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
1	$Y[0][0]$	$Y[0][1]$								
2			$Z[0][0]$	$Z[0][1]$						
3		HIT								
4					$Z[1][0]$	$Z[1][1]$				
5							$X[0][0]$	$X[0][1]$		
6	HIT									
7				HIT						
8		HIT								
9						HIT				
10								HIT		
11									$Y[1][0]$	$Y[1][1]$
12			HIT							
13										HIT
14					HIT					
15	$X[1][0]$	$X[1][1]$								
16									HIT	
17				HIT						
18										HIT
19						HIT				
20		HIT								

Although the table extends to length 20, we will calculate the miss rate with respect to the first 18 accesses to memory. There are 6 of the first 18 accesses that are misses, for a miss ratio of $\frac{6}{18} = 33.33\%$ miss rate.

c

If we want the inner two loops of our matrix multiply code to run without cache misses (except for compulsory misses), we need to make sure there is enough space to hold the complete BxB block in Z, and the B-length rows in X and Y.

In other words, the data cache must be large enough to hold $B^2 + 2B$ matrix elements simultaneously to avoid (capacity) cache misses. Assuming Ben Bitdiddle has a 16 kB data cache and that matrix elements are 1 word each, Ben's maximum blocking factor is 63.

When Ben notices the poor performance of his code on the UltraSPARC-I with a TLB, his matrices are stored in row-major order within his virtual address space. However, due to the size of the matrix ($3000 * 4$bytes $= 12000$bytes per matrix row, which is greater than the page size of 8 KB) each row starts on a different page. For that matter, any element of any of the matrices is on a different page from the corresponding element (same column) in the previous and subsequent rows of the same matrix. In Ben's code (and using the maximum blocking factor from above), when the innermost loop is reached and executed it touches (in the best case for Ben) 63 separate pages from the data of Z and 1 page (at best) for the data from Y. At this point the TBL is full, containing 64 VPN-to-PPN mappings in its 64 lines. However, Ben's code then accesses and sets an entry of X. Since the TBL is governed by an LRU replacement policy, this causes the table entry correpsonding to $Z[kk, j]$ (the oldest Z page) to be removed from the TBL. In other words, the TBL is one line too short; every time Ben's code enters the innermost loop, it must reload **all** of the VPN-to-PPN mappings corresponding to Z and X into the TBL from memory. Therefore the hit rate is below 50% for the TLB. Thus the overhead connected to the TLB outweighs the benefit in this case. Consequently it leads to a slow down in performance.

d

The optimized `BlockMove` routine incorporates two main ideas:

1. The first was illustrated in the blocked code for the matrix multiplication in 1B: blocking to maximize the use of the cache in the reading of the matrix that's being copied.

2. The second and more important technique is the use of `dcbz` to avoid the penalty associated with the Macintosh's write-allocate cache behavior. If we weren't using the `dcbz` instruction, then every time we wrote to a new block of the destination memory array, we would incur a cache miss and a read from memory into cache that would be both costly and pointless (since the existing data is about to be written over anyway). Normally, the only way to bring something into the cache is to read it (the appropriate line) from memory. The `dcbz` instruction allows us to specify that a line is loaded in the cache without actually going through the hassle of reading the contents from memory, when we don't care about the contents. Reading the source array requires a cache load, as will writing the modified (copied) cache contents back out to memory when the program is finished with each cache line: those are costs that cannot be avoided. However, before writing to any memory location, the memory location is "read" into the cache through the use of the lightweight (1 cycle) `dcbz`, and **only then** is written to. We could think of `dcbz` as a data prefetch operation, although it ignores the values of the (destination) array that's being read.

In other words, the optimized `BlockMove` routine will access avoid spurious loads into the cache with `dcbz`, and use blocking so that all work on a cache line can be completed in a single pass once that cache line is loaded into the cache – there's no need for any flushed cache line to ever return to the cache.