We are going to compare the behavior of 3 different cache configurations on a benchmark program to better understand the impact of the cache configuration on the performance of the benchmark.

The first cache we will consider is a 64-line direct mapped cache.

The second is a 2-way set associative cache that uses the LRU, or least recently used, replacement strategy, and has a total of 64 lines.

The third is a 4-way set associative cache that uses the LRU replacement strategy, and also has a total of 64 lines.

Note that all three caches have the same capacity in that they can store a total of 64 words of data.

In a direct mapped cache any particular memory address maps to exactly one line in the cache.

Let's assume that our data is 32 bits, or 4 bytes wide.

This means that consecutive addresses are 4 bytes apart, so we treat the bottom two address bits as always being 00 so that our address is on a data word boundary.

Next, we want to determine which cache line this particular address maps to.

Since there are 64 lines in this cache, we need 6 bits to select one of the 64 lines.

These 6 bits are called the index.

In this example, the index is 000011, so this particular address maps to line 3 of the cache.

The data that gets stored in the cache is the tag portion of the address of the line plus the 32 bits of data.

The tag is used for comparison when checking if a particular address is in the cache or not.

It uniquely identifies the particular memory address.

In addition, each cache line has a valid bit that lets you know whether the data in the cache is currently valid or not.

This is important upon startup because without this bit there is no way to know whether the data in the cache is garbage or real data.

In a 2-way set associative cache, the cache is divided into 2 sets each with half the number of lines.

So we have two sets with 32 lines each.

Since there are only 32 lines, we now only need a 5 bit index to select the line.

However, any given index can map to two distinct locations, one in each set.

This also means that when the tag comparisons are done, two comparisons are required, one per set.

In a 4-way set associative cache, the cache is divided into 4 sets each with 16 lines.

The width of the index is now 4 bits to select the cache line.

Here, selecting a line identifies one of 4 words as possible locations for reading or writing the associated data.

This also implies that 4 tags need to be compared when trying to read from the cache to determine if the desired address is stored in the cache or not.

The test program begins by defining a few constants, J, A, B, and N.

J specifies the address where the program lives.

A is the starting address of data region 1, and B is the starting address of data region 2.

Finally, N specifies the size of the data regions.

Since one word consists of 4 bytes, 16 bytes of data mean that there are 4 data elements per region.

Next the assembler is told that the beginning of the program is at address 0x1000.

The green rectangle identifies the outer loop, and the yellow rectangle identifies the inner loop of the code.

Before entering the outer loop, a loop counter, which is stored in register R6 is initialized to 1,000.

Then each time through the outer loop, R6 is decremented by 1 and the loop is repeated as long as R6 is not equal to 0.

The outer loop also resets R0 to N each time through the loop.

R0 is used to hold the desired array offset.

Since the last element of the array is stored at location N – 4, the first step of the inner loop, is to decrement R0 by 4.

R1 is then loaded with the value at address A + N – 4 which is the address of A[3] because array indeces begin at 0.

R2 is loaded with B[3].

As long as R0 is not equal to 0, the loop repeats itself, each time accessing the previous element of each array until the first element (index 0) is loaded.

Then the outer loop decrements R6 and repeats the entire thing 1000 times.

Now that we understand the configuration of our three caches and the behavior of our test benchmark, we can begin comparing the behavior of this benchmark on the three caches.

The first thing we want to ask ourselves is which of the three cache configurations gets the highest hit ratio.

Here we are not asked to calculate an actual hit ratio, instead we just need to realize that there are 3 distinct regions of data in this benchmark.

The first holds the instructions, the second holds array A, and the third holds array B.

If we think about the addresses of each of these regions in memory, we see that the first instruction is at address 0x1000.

This will result in an index of 0 regardless of which cache you consider, so for all three caches the first instruction would map to the first line of the cache.

Similarly the first element of arrays A and B are at address 0x2000 and 0x3000.

These addresses will also result in an index of 0 regardless of which of the three caches you consider.

So we see that the 1st CMOVE, A[0], and B[0] would all map to line 0 of the cache.

Similarly, the 2nd CMOVE whose address is 0x1004 would map to line 1 of the cache as would array elements A[1] and B[1].

This tells us that if we use the direct mapped cache, or a 2-way set associative cache, then we will have cache collisions between the instructions, and the array elements.

Collisions in the cache imply cache misses as we replace one piece of data with another in the cache.

However, if we use a 4-way set associative cache then each region of memory can go in a distinct set in the cache thus avoiding collisions and resulting in 100% hit rate after the first time through the loop.

Note that the first time through the loop each instruction and data access will result in a cache miss because the data needs to initially be brought into the cache.

But when the loop is repeated, the data is already there and results in cache hits.

Now suppose that we make a minor modification to our test program by changing B from 0x3000 to 0x2000.

This means that array A and array B now refer to same locations in memory.

We want to determine, which of the cache's hit rate will show a noticeable improvement as a result of this change.

The difference between our original benchmark and this modified one is that we now have two distinct regions of memory to access, one for the instructions, and one for the data.

This means that the 2-way set associative cache will no longer experience collisions in its cache, so its hit rate will be significantly better than with the original benchmark.

Now suppose that we change our benchmark once again, this time making J, A and B all equal to 0, and changing N to be 64.

This means that we now have 16 elements in our arrays instead of 4.

It also means that the array values that we are loading for arrays A and B are actually the same as the instructions of the program.

Another way of thinking about this is that we now only have one distinct region of memory being accessed.

What we want to determine now, is the total number of cache misses that will occur for each of the cache configurations.

Let's begin by considering the direct mapped cache.

In the direct mapped cache, we would want to first access the first CMOVE instruction.

Since this instruction is not yet in the cache, our first access is a cache miss.

We now bring the binary equivalent of this instruction into line 0 of our cache.

Next, we want to access the second CMOVE instruction.

Once again the instruction is not in our cache so we get another cache miss.

This results in our loading the 2nd CMOVE instruction to line 1 of our cache.

We continue in the same manner with the SUBC instruction and the first LD instruction.

Again we get cache misses for each of those instructions and that in turn causes us to load those instructions into our cache.

Now, we are ready to execute our first load operation.

This operation wants to load A[15] into R1.

Because the beginning of array A is at address 0, then A[15] maps to line 15 of our cache.

Since we have not yet loaded anything into line 15 of our cache, this means that our first data access is a miss.

We continue with the second load instruction.

This instruction is not yet in the cache, so we get a cache miss and then load it into line 4 of our cache.

We then try to access B[15].

B[15] corresponds to the same piece of data as A[15], so this data access is already in the cache thus resulting in a data hit for B[15].

So far we have gotten 5 instruction misses, 1 data miss and 1 data hit.

Next we need to access the BNE instruction.

Once again we get a cache miss which results in loading the BNE instruction into line 5 of our cache.

The inner loop is now repeated with R0 = 60 which corresponds to element 14 of the arrays.

This time through the loop, all the instructions are already in the cache and result in instruction hits.

A[14] which maps to line 14 of our cache results in a data miss because it is not yet present in our cache.

So we bring A[14] into the cache.

Then as before, when we try to access B[14], we get a data hit because it corresponds to the same piece of data as A[14].

So in total, we have now seen 6 instruction misses and 2 data misses.

The rest of the accesses have all been hits.

This process repeats itself with a data miss for array element A[i], and a data hit for array element B[i] until we get to A[5] which actually results in a hit because it corresponds to the location in memory that holds the BNE(R0, R) instruction which is already in the cache at line 5.

From then on the remaining data accesses all result in hits.

At this point we have completed the inner loop and proceed to the remaining instructions in the outer loop.

These instructions are the second SUBC and the second BNE instructions.

These correspond to the data that is in lines 6 and 7 of the cache thus resulting in hits.

The loop then repeats itself 1000 times but each time through all the instructions and all the data is in the cache so they all result in hits.

So the total number of misses that we get when executing this benchmark on a direct mapped cache is 16.

These are known as compulsory misses which are misses that occur when you are first loading the data into your cache.

Recall that the direct mapped cache has one set of 64 lines in the cache.

The 2-way set associative has 2 sets of 32 lines each, and the 4-way set associative has 4 sets of 16 lines each.

Since only 16 lines are required to fit all the instructions and data associated with this benchmark, this means that effectively, only one set will be used in the set associative caches, and because even in the 4-way set associative cache there are 16 lines, that means that once the data is loaded into the cache it does not need to be replaced with other data so after the first miss per line, the remaining accesses in the entire benchmark will be hits.

So the total number of misses in the 2-way and 4-way set associative caches is also 16.