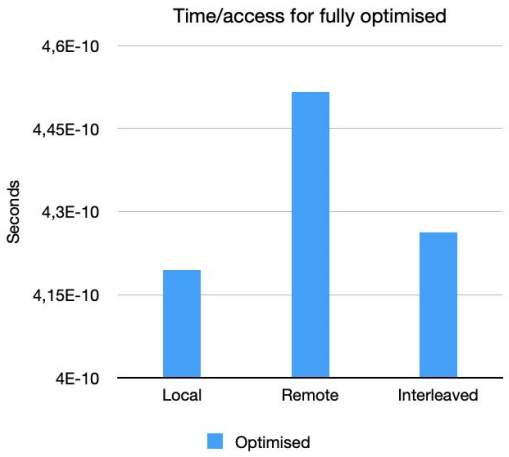
Part 1

Before we deoptimised the numa.c program, we obtained the following results :

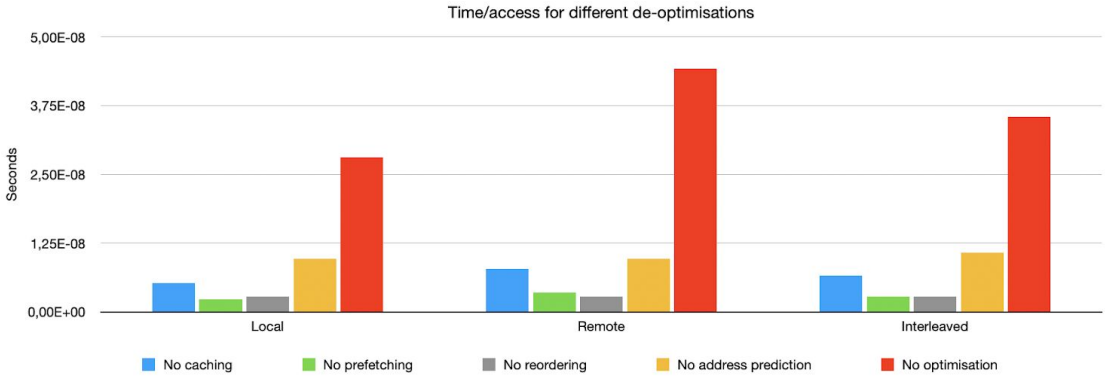


As we can see, the shortest time per access occurs when opting for the local policy, followed by the interleaved policy, and finally, the longest, the remote policy. Those results can be explained by the time it takes to retrieve data from a remote memory location to a local memory location. For the local policy, the memory is always allocated on the current node, so there is no “remote-fetching” delay. For the interleaved, some of the memory is allocated on the current node, some is allocated on a remote node, so it makes sense that the time is longer. Finally, for the remote policy, the entirety of the data is allocated on a remote node, when fetching data we always get some “remote-fetching” delay. Once again, it makes sense that the time is longer for the remote than for the interleaved.

For the second part of this exercise we tried four different de-optimisations:

- Access one element per cache block in order not to benefit from caching.
- Jump two caches blocks every time to avoid line prefetching.
- Calculate the new address using data read in the array to prevent reordering instructions.
- Use random number to break pattern in our memory accesses

Below are the results obtained with each one of thoses de-optimisation methods.



It globally runs faster when the program memory is allocated on the node executing the program (local) which actually seems intuitive since the path to memory is the shortest in this configuration. As explained in the handout the fully optimised version is able to hide most of the numa latency, but as we de-optimise, local tends to be the fastest, followed by interleaved and finally remote Numa memory allocation policy. As soon as cores do not benefit from caching, differences appear in execution times of different numa memory policies. It shows at which point memory locality is important. Disabling address prediction is the most effective de-optimisation.

Part 2

In this part we will focus on two sections of the code, `thread1Func` and `thread2Func`. Here are the assembly code in order:

```
thread1Func:
      movl    $1, X(%rip)           // Write 1 into X
      movl    Y(%rip), %eax         // Read Y into eax
      movl    %eax, r1(%rip)        // Write  eax into r1

thread2Func:
      movl    $1, Y(%rip)           // Write 1 into Y
      movl    X(%rip), %eax         // Read X into eax
      movl    %eax, r2(%rip)        // Write reax into r2
```

As we learned in class, an optimisation technique used by hardware is to let unrelated reads be executed before writes to reduce the memory latency. Here is the assembly code we would get if we let within-thread-data-race-free reads be executed before writes.

```
thread1Func:
      movl    Y(%rip), %eax         // Read Y into eax
      movl    $1, X(%rip)           // Write 1 into X
      movl    %eax, r1(%rip)        // Write  eax into r1
thread2Func:
      movl    X(%rip), %eax         // Read X into eax
      movl    $1, Y(%rip)           // Write 1 into Y
      movl    %eax, r2(%rip)        // Write reax into r2
```

Now if we simulate an execution with both threads running concurrently, here is a possible instructions order we could get:

```
Thread N°
    1 movl    Y(%rip), %eax       // Read Y into eax
    2       movl    X(%rip), %eax       // Read X into eax
    1       movl    $1, X(%rip)         // Write 1 into X
    1       movl    %eax, r1(%rip)     // Write eax into r1
    2       movl    $1, Y(%rip)         // Write 1 into Y
    2       movl    %eax, r2(%rip)     // Write reax into r2
```

With this execution order we would end up with both r1 and r2 being equal to zero since both reads to X and Y happen before writes to the same locations.

We ran our `execute_order.sh` script 6 times and we got a mean value of 99% reorderings when the two explicit threads are on the same socket, and a mean value of 74% reorderings when the two explicit threads are on different sockets. When placing the two threads on different sockets, we clearly see that the number of reorderings goes down. That can be explained by the fact that bringing a cache block from a remote socket takes time, and the time spent fetching those blocks lowers the chance of having concurrent executions between the two threads, which then lowers the number of reorderings.

To force the required ordering, we need to prevent both reads to be executed before the corresponding writes. To achieve that, we must insert execution fences between the writes and the reads to X and reads to Y. Those fences will make sure that the code preceding them has been executed before moving on to the code that is following. Thus in both threads, the read will be blocked by the fence into the ROB until the write is executed. When executing the order program with the added fences we observe that no memory reordering has been made, and we get the following assembly code :

```
thread1Func:
          movl    $1, X(%rip)           // Write 1 into X
          mfence                        // Fence
          movl    Y(%rip), %eax         // Read Y into eax
          movl    %eax, r1(%rip)        // Write eax into r1

thread2Func:
          movl    $1, Y(%rip)           // Write 1 into Y
          mfence                        // Fence
          movl    X(%rip), %eax         // Read eax into X
          movl    %eax, r2(%rip)        // Write eax into r2
```

We can see that the fences we added in the C code have been translated into `mfence` instructions in the assembly code, which ensure that the instructions will not get reordered across the fences.