COL216: Assignment 3
Cache Simulator

Mihir Kaskhedikar 2021CS10551
Rishabh Verma 2021CS10581

11 May 2023

1 Code Explanation

We have represented the L1 and L2 caches by a vector of vector of pairs called `L1Tag` and `L2Tag`. We have only maintained the tag array for each of the caches and not the data array since we are not concerned with the data values but only need to know if they have been updated while they were in the cache or not. Any pair contained in `L1Tag[idx]` has its first entry as a boolean variable and the second entry as the tag index `tag`. The boolean variable denotes if the block corresponding to index bits `idx` and tag bits `tag` is dirty or not (if the boolean variable is true then the dirty bit is on else off). Similar considerations hold for `L2Tag`. We have created the following simple functions that get used many times in the main functions of our code:

- **MemoryBlock**: Converts the byte address to its corresponding block
- **L1sets** and **L2sets**: Finds the set corresponding to a block in the cache
- **L1tag** and **L2tag**: Finds the tag corresponding to a block in the cache.

The following variables were made global:

- `L1ReadMiss`, `L1WriteMiss`, `L1ReadHit`, `L1WriteHit`, `L1update`, `WriteBackFromL1`
- `L2ReadMiss`, `L2WriteMiss`, `L2ReadHit`, `L2WriteHit`, `L2update`, `WriteBackFromL2`
- `MemoryRead`, `MemoryWrite`

For implementing the LRU policy we are ensuring that for any index `idx`, both `L1Tag[idx]` and `L2Tag[idx]` vectors contain the tag values in the order of least recent to most recent. So the first entry of the vector contains the least recently accessed tag while the last entry of the vector contains the most recently accessed tag (either by **read** or **write** access to the block corresponding to that tag and set values). We have implemented the following main functions:

- **ReadL1**: We call this function when we encounter a r instruction in the trace file. If the block is found, we increment `L1ReadHit` and update the LRU ordering of the set and return. Else we increment `L1ReadMiss` and make a call to **ReadL2** function to bring the block in L1.
- **WriteL1**: We call this function when we encounter a w instruction in the trace file. If the block is found, we increment `L1WriteHit` and update the LRU ordering of the set and return. Else we increment `L1WriteMiss` and make a call to **ReadL2** function to bring the block in L1. **After that we write on the block that was brought by turning its dirty bit on**.
- **AddL1**: We call this function when we require to add a block to L1. We increment `L1update` at the beginning of the function. So suppose the block is added to set x of L1 and it has tag y. If size of `L1Tag[x]` $<$ `L1Assoc`, we simply append {`false,y`} to `L1Tag[x]` (to maintain LRU ordering). Otherwise, we remove the first pair of `L1Tag[x]`, shift the pairs ahead by one position left, and then append {`false,y`} to `L1Tag[x]`. Then we check if the first entry of the removed pair is true(i.e. if the dirty bit corresponding to this tag is on). If it is, we increment `WriteBackFromL1` and call `WriteL2` function with the input parameter as the block index corresponding to the set x and the tag of the removed pair(which is its second entry).
- **ReadL2**: We call this function when we need to find some block in L2 that needs to be brought in L1 due to a **read/write** miss of that block in L1. If the block is found, we increment `L2ReadHit`, update the LRU ordering of the set and call **AddL1** function with argument as this memoryblock. Else we increment `L2ReadMiss` and make a call to **ReadMemorytoBoth** function **to bring the block to both L1 and L2 caches**.

- **WriteL2**: We call this function when we had evicted some block in L1 that was dirty. If the block is found, we increment `L2ReadHit`, overwrite this block by **turning its dirty bit on** and update the LRU ordering of the set and return. Else we increment `L2WriteMiss`and make a call to **ReadMemorytoL2** function with argument as this block to **bring the block to only L2** and then overwrite it by **turning its dirty bit on**.
- **AddL2**: We call this function when we require to add a block to L2. We increment `L2update` at the beginning of the function. So suppose the block is added to set x of L2 and it has tag y. If size of `L2Tag[x]` $<$ `L2Assoc`, we simply append {`false,y`} to `L2Tag[x]` (to maintain LRU ordering). Otherwise, we remove the first pair of `L2Tag[x]`, shift the pairs ahead by one position left, and then append {`false,y`} to `L2Tag[x]`. Then we check if the first entry of the removed pair is true(i.e. if the dirty bit corresponding to this tag is on). If it is, we increment `WriteBackFromL2` and call `WriteMemory` function with the input parameter as the block index corresponding to the set x and the tag of the removed pair(which is its second entry).
- **WriteMemory**: We call this function when we need to write back some block in memory. Since we are not concerned with data values, we just increment `MemoryWrite` here.
- **ReadMemoryToL2**: This function is called when we had encountered a **read/write** miss in L1 and **read** miss in L2. So here we increment `MemoryRead` and call **AddL2** function.
- **ReadMemoryToBoth**: This function is called when **we had evicted some dirty block in L1 and encountered a write miss for this block in L2**. So here we increment `MemoryRead`, **call AddL2 function first and then call AddL1 function**.

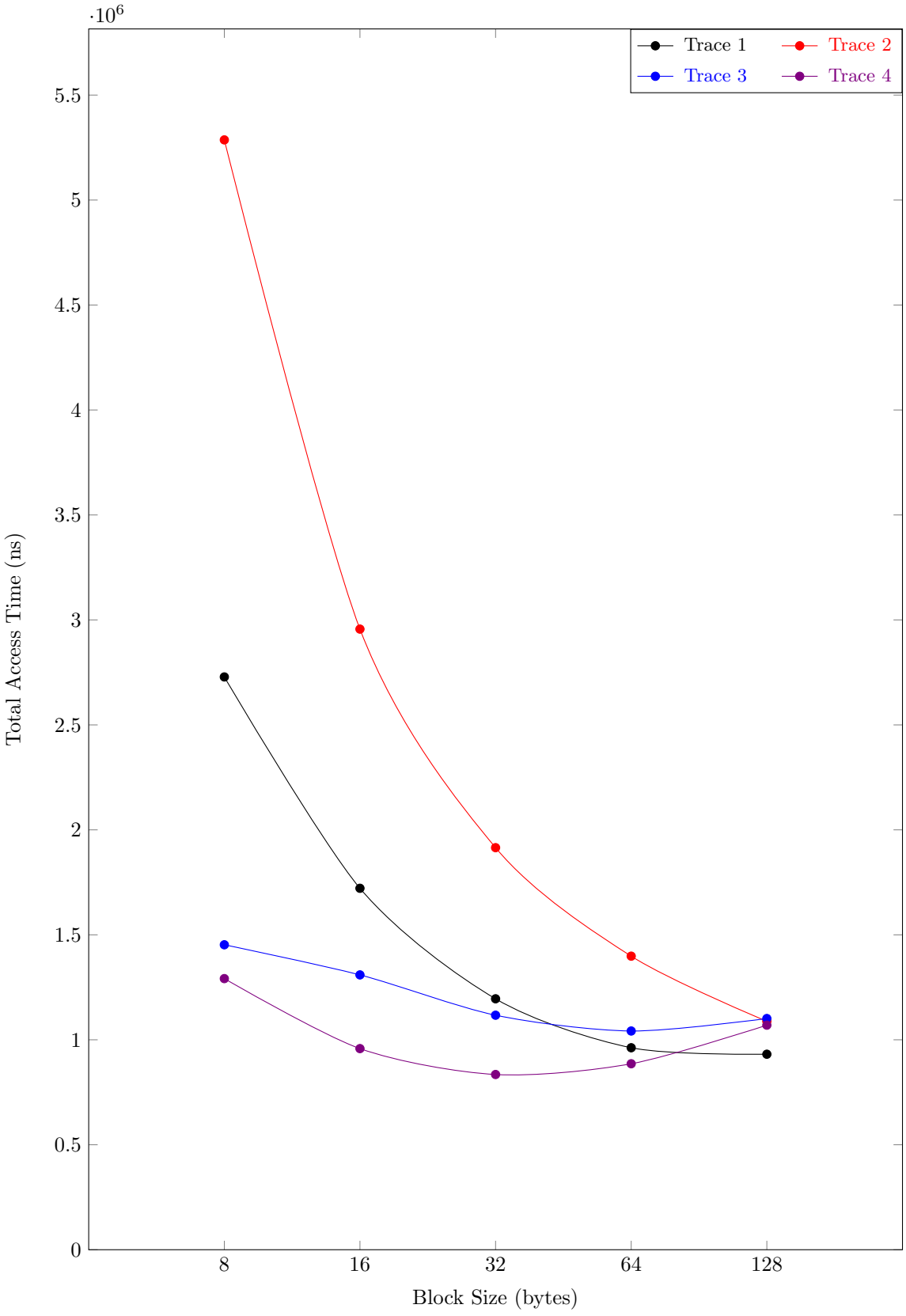
2 Relation between Output Parameters and Code Variables

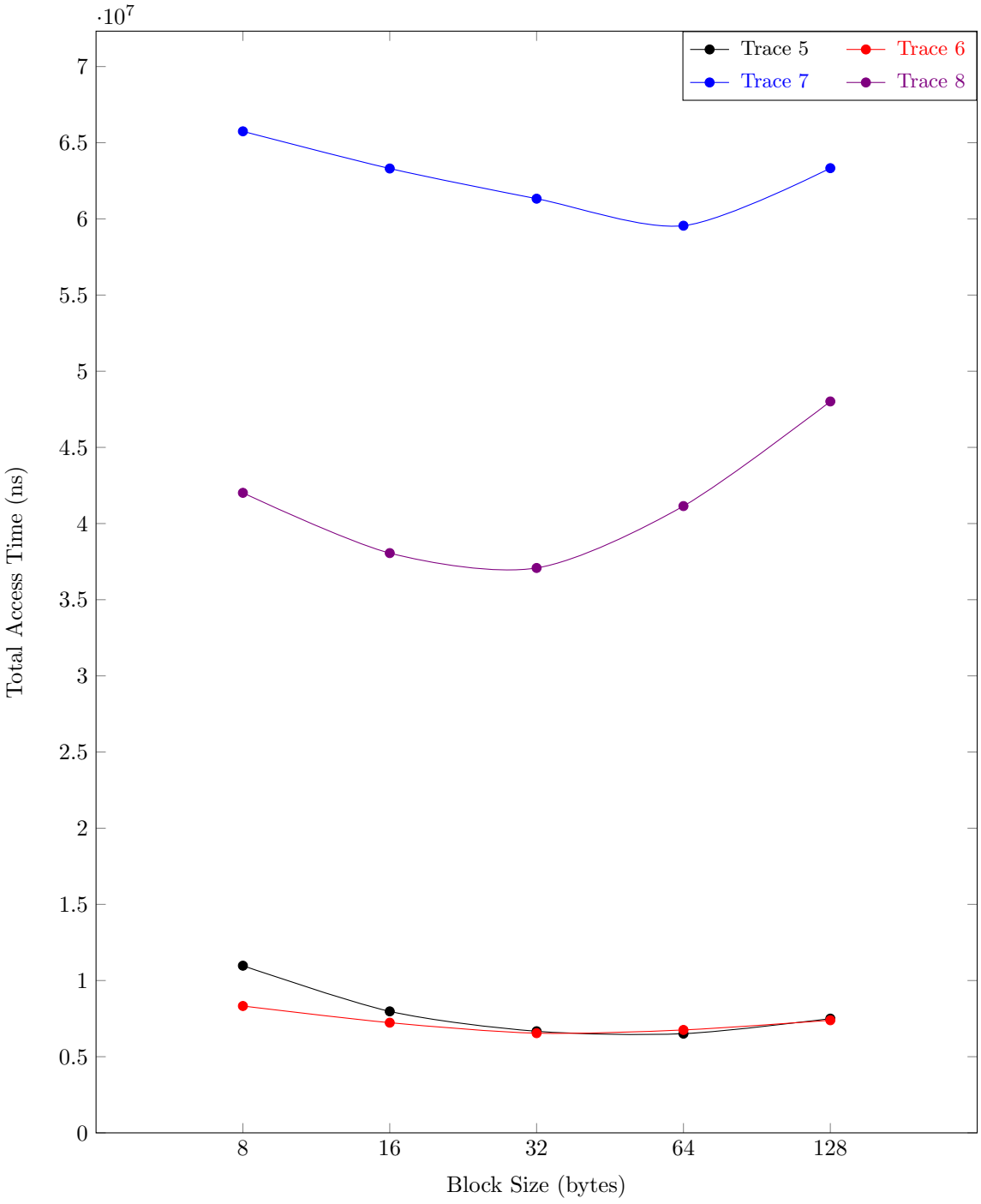
- Number of L1 Reads = `L1ReadHit` + `L1ReadMiss`
- Number of L1 Read Misses = `L1ReadMiss`
- Number of L1 Writes = `L1WriteHit` + `L1WriteMiss`
- Number of L1 Write Misses = `L1WriteMiss`
- L1 Miss Rate = (`L1ReadMiss` + `L1WriteMiss`)/(`L1ReadMiss` + `L1ReadHit` + `L1WriteMiss` + `L1WriteHit`)
- Number of WriteBacks from L1 = `WriteBackFromL1`
- Number of L2 Reads = `L2ReadHit` + `L2ReadMiss`
- Number of L2 Read Misses = `L2ReadMiss`
- Number of L2 Writes = `L2WriteHit` + `L2WriteMiss`
- Number of L2 Write Misses = `L2WriteMiss`
- L2 Miss Rate = (`L2ReadMiss` + `L2WriteMiss`)/(`L2ReadMiss` + `L2ReadHit` + `L2WriteMiss` + `L2WriteHit`)
- Number of WriteBacks from L2 = `WriteBackFromL2`
- Total Time Taken (in ns) = 1 × (Number of L1 Reads + Number of L1 Writes + `L1update`) + 20 × (Number of L2 Reads + Number of L2 Writes + `L2update`) + 200 × (`MemoryRead`+`MemoryWrite`).

3 Analysing total access time with varying parameters

3.1 Block Size vs Total Access Time

3.1.1 Graphs



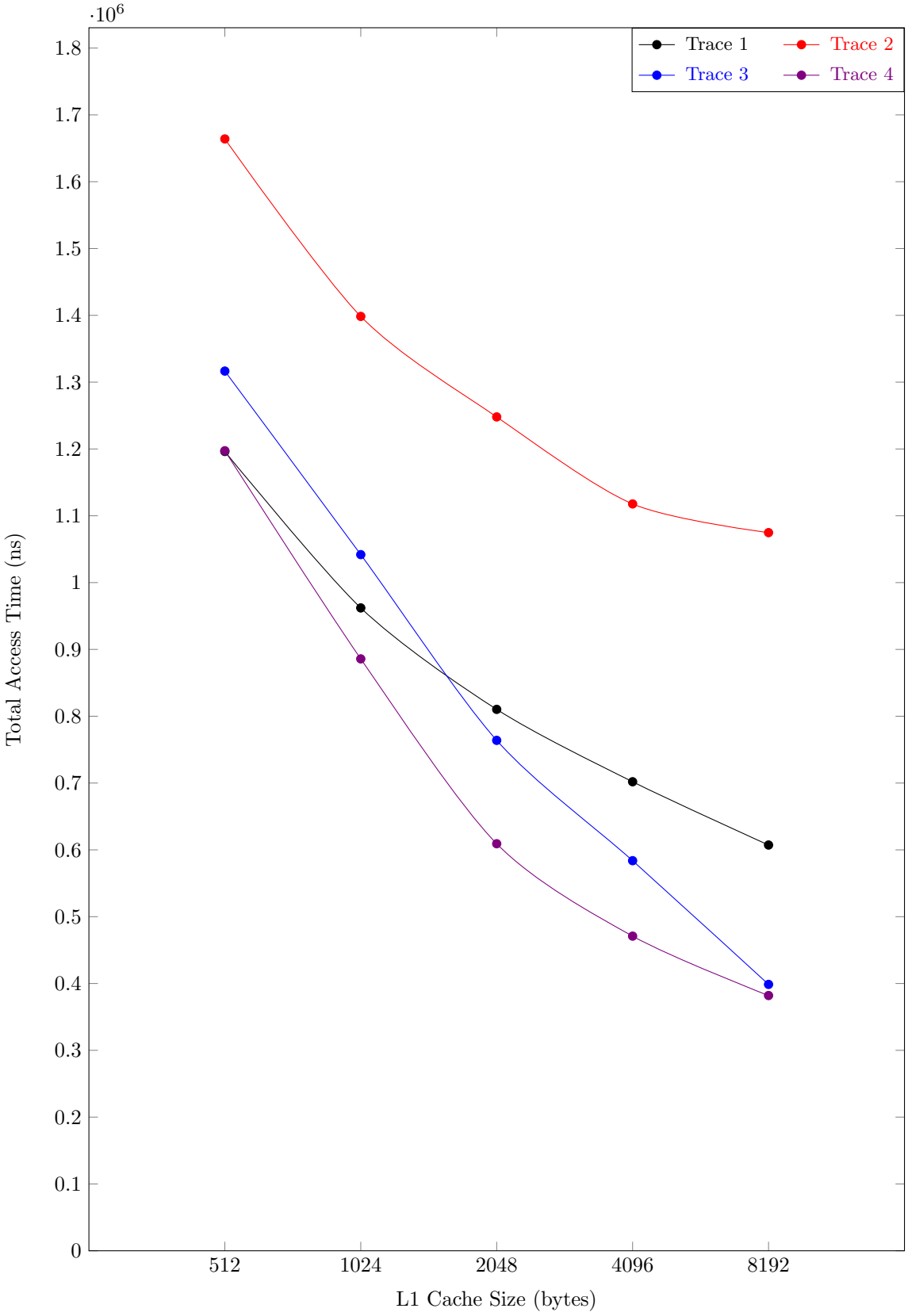


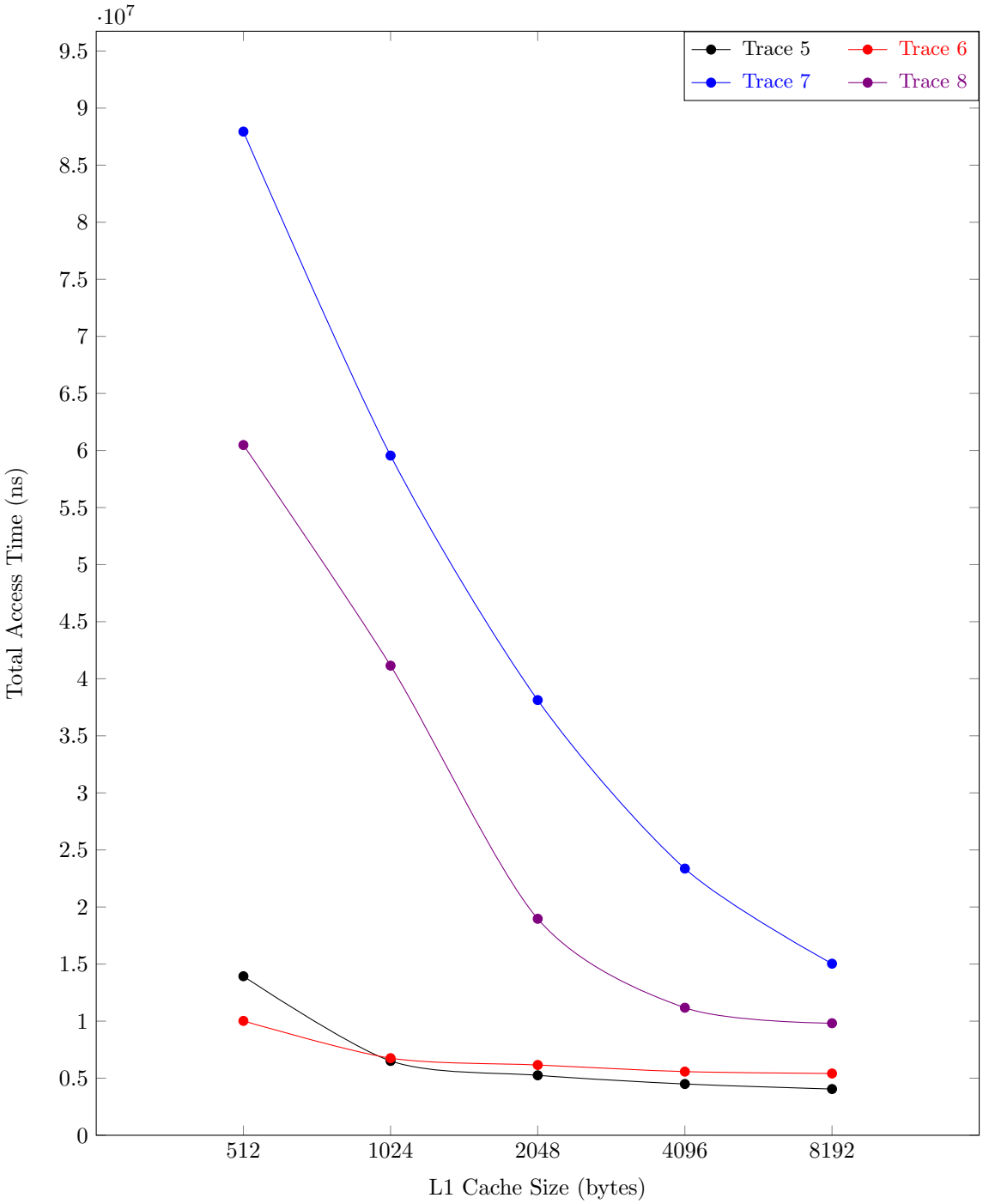
3.1.2 Observations

- We can infer that the total access time decreases up to a certain point when increasing the block size. After which, it increases because the number of sets in the cache decrease thereby **increasing the number of conflict misses**.
- In traces 1 and 2, increasing the block size from 8 to 16, 16 to 32, 32 to 64 bytes drastically reduces the total access time. So we can say that traces 1 and 2 involved more accesses to byte addresses that were close to each other, thereby decreasing time by a large margin when the block size was increased.
- In traces 3, 4, 5 and 6, the total access time first decreases but increases when the block size is increased from 64 to 128 bytes for traces 3 & 5 and 32 to 64 bytes for traces 4 & 6. This reflects the fact that large block sizes do not increase the efficiency of the cache.
- In traces 7 and 8, the total access time increases by a big margin when the block size is increased from 64 to 128 bytes for trace 7 and 32 to 64 bytes for trace 8 due to more conflict misses (large files).

3.2 L1 Cache Size vs Total Access Time

3.2.1 Graphs



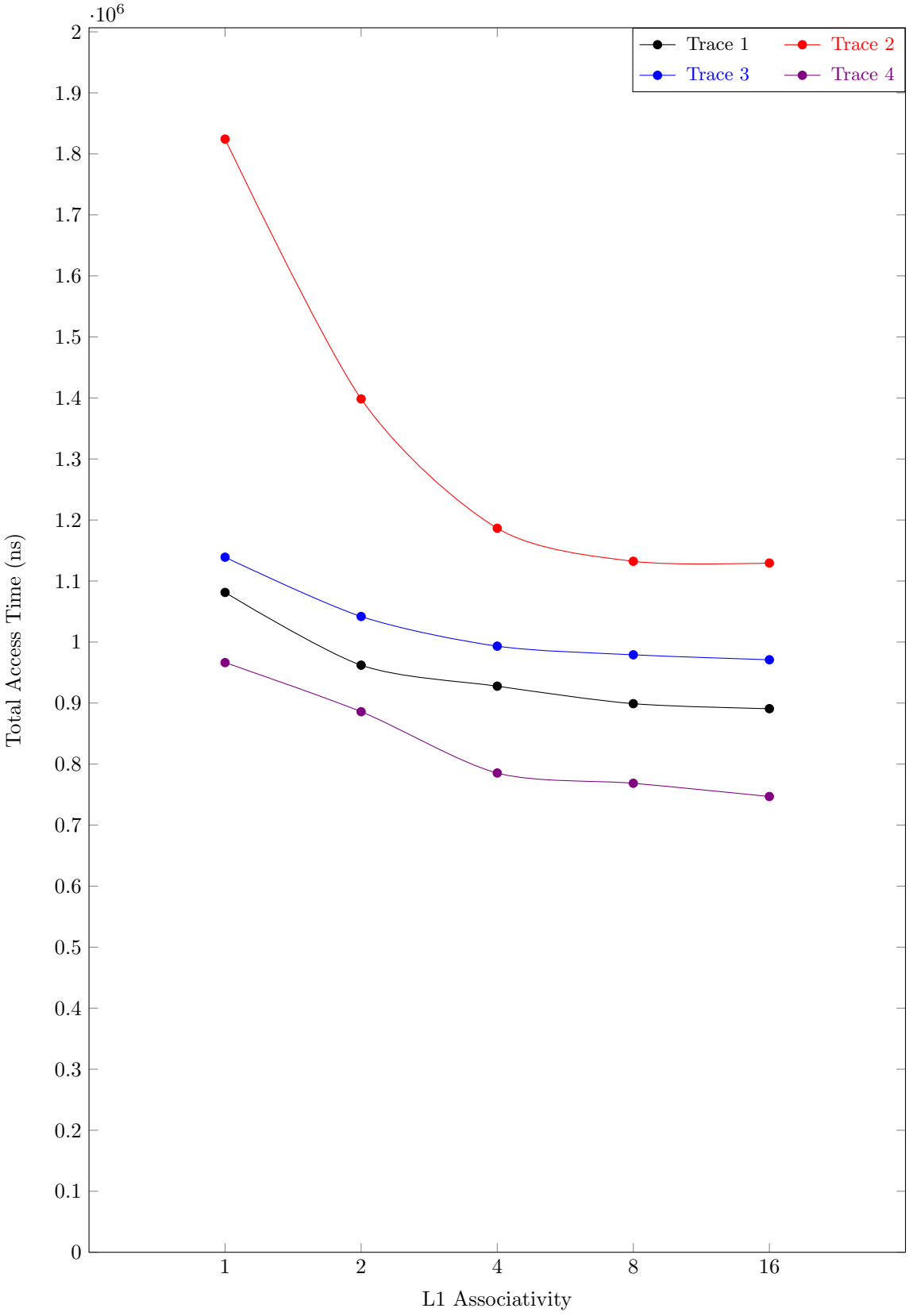


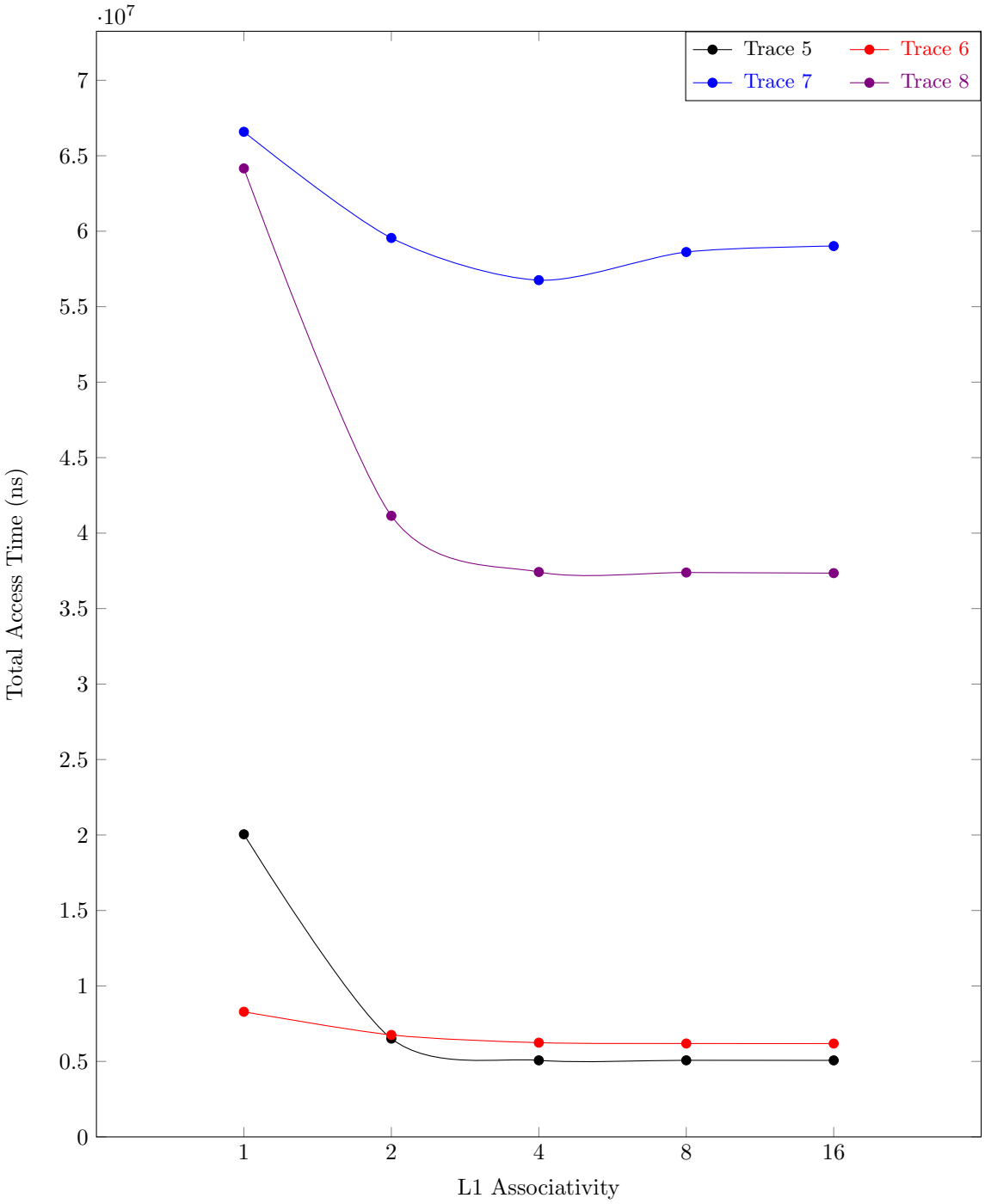
3.2.2 Observations

- We can infer that total access time decreases when increasing L1 cache size because the number of sets increase with the cache size **reducing the number of conflict misses in L1**.
- In traces 1, 2, 3 and 4, increasing L1 cache size drastically reduces total access time as the effect of increase in number of sets is dominant.
- In traces 5 and 6, increasing L1 cache size reduces total access time slightly because the instructions in these traces are reading from or writing to a small range of addresses. This means that in these traces, increasing number of sets **does not lead to more hits in L1**.
- In traces 7 and 8, total access time decreases drastically when L1 cache size is increased due to less conflict misses(large files).

3.3 L1 Associativity vs Total Access Time

3.3.1 Graphs



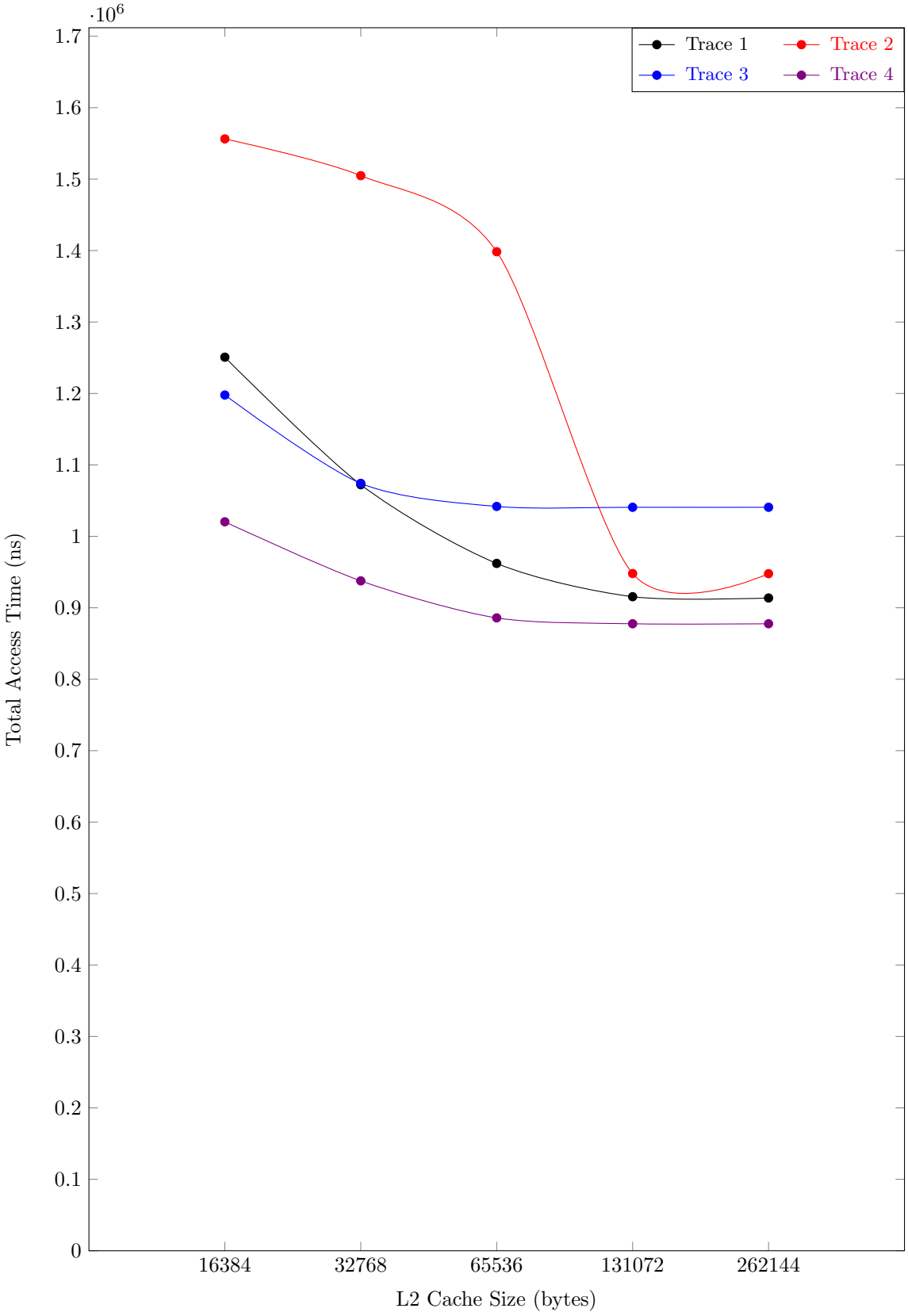


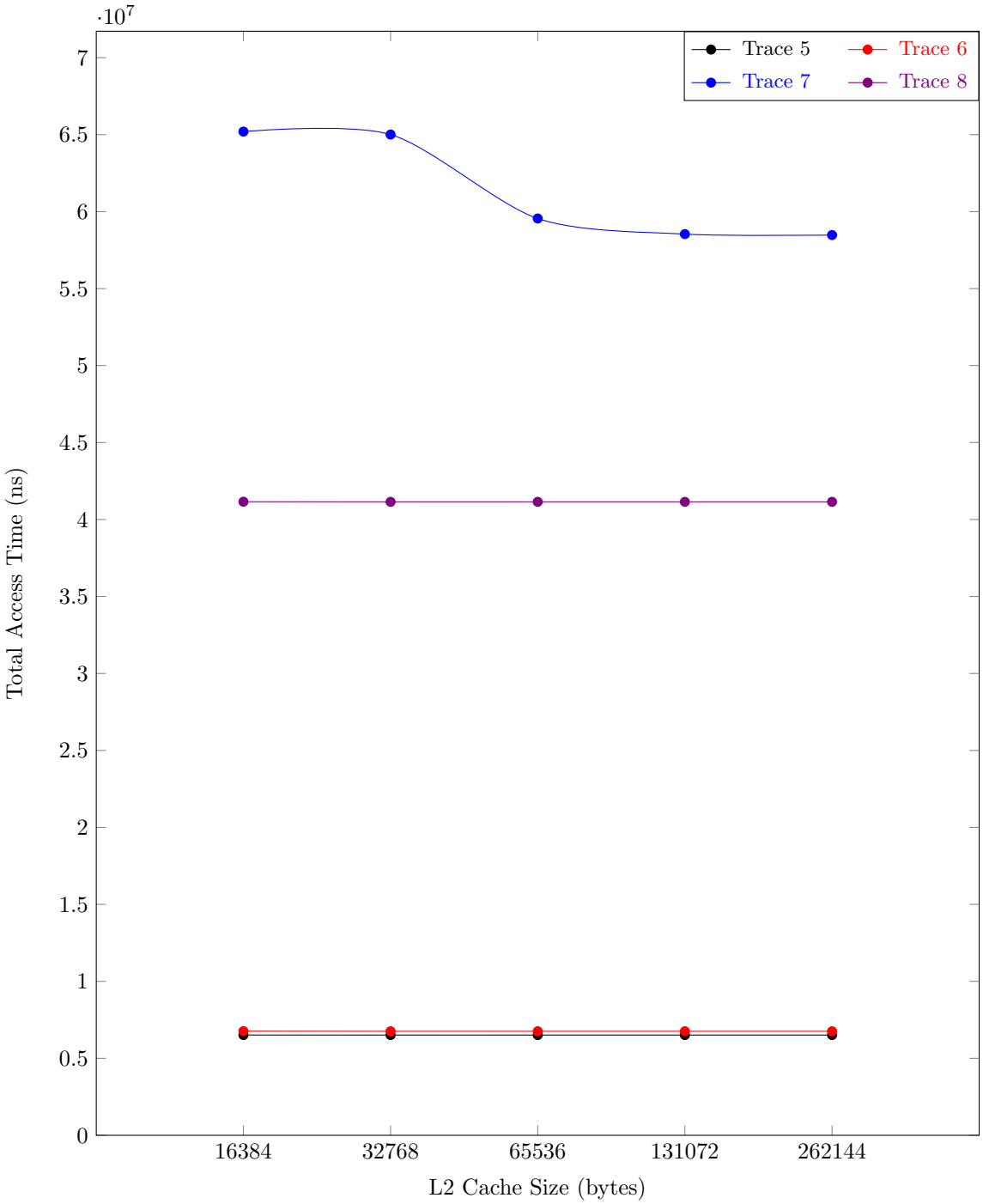
3.3.2 Observations

- We can infer that the total access time decreases up to a certain point when increasing the L1 associativity. After which, it increases because the number of sets in the cache decrease thereby increasing the number of conflict misses.
- In traces 2, 5 and 8, increasing the L1 associativity reduces the total access time significantly as the instructions in these traces are reading from or writing to a **small range of addresses**. This means that with high associativity, there are **more hits in L1**.
- In traces 1, 3, 4 and 6, increasing the L1 associativity causes a slight decrease in the total access time initially but it remains constant after that as the instructions in these traces are reading from or writing to a **decent range of addresses**.
- To explain the trend in trace 7, refer to the Example section.

3.4 L2 Cache Size vs Total Access Time

3.4.1 Graphs



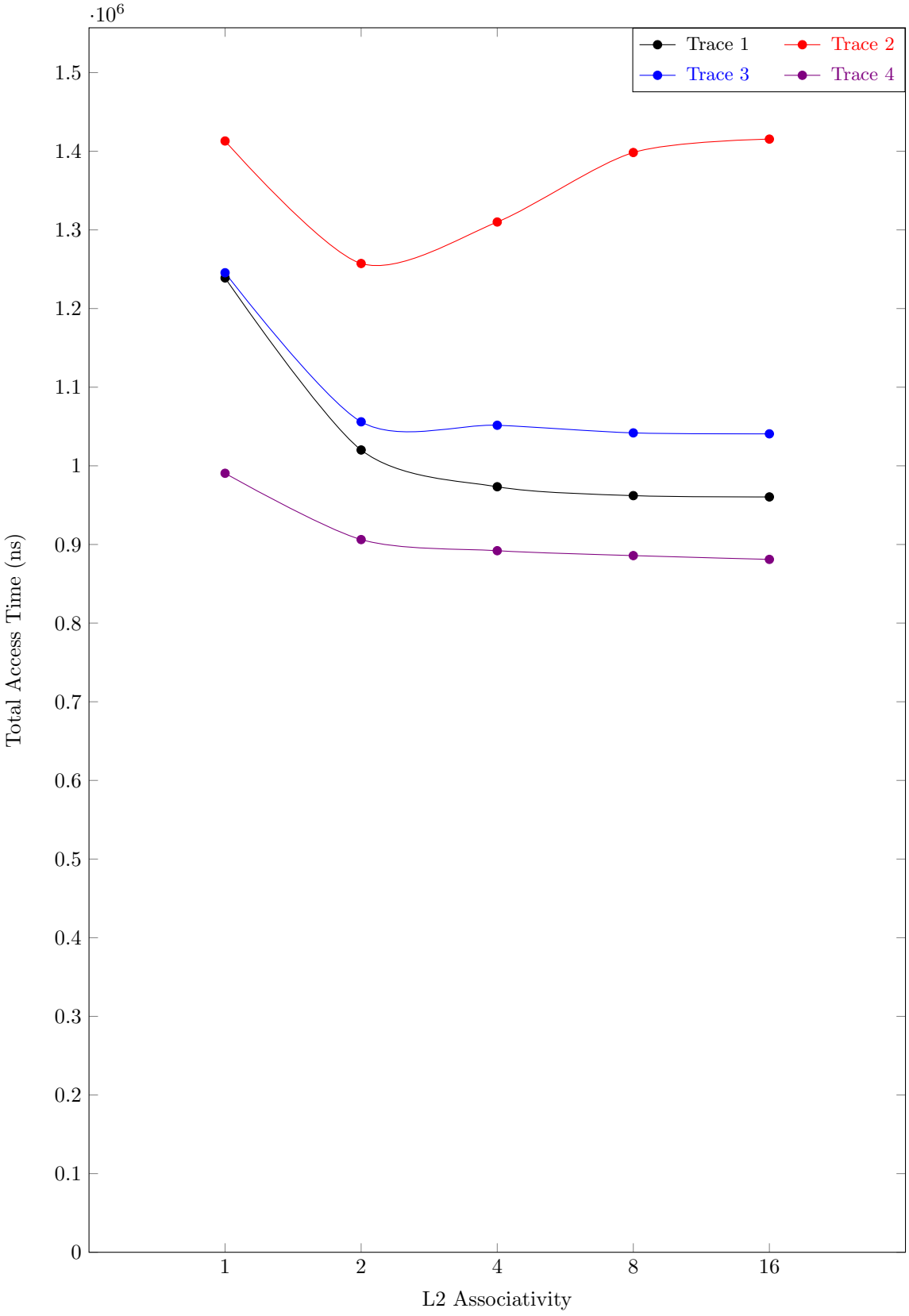


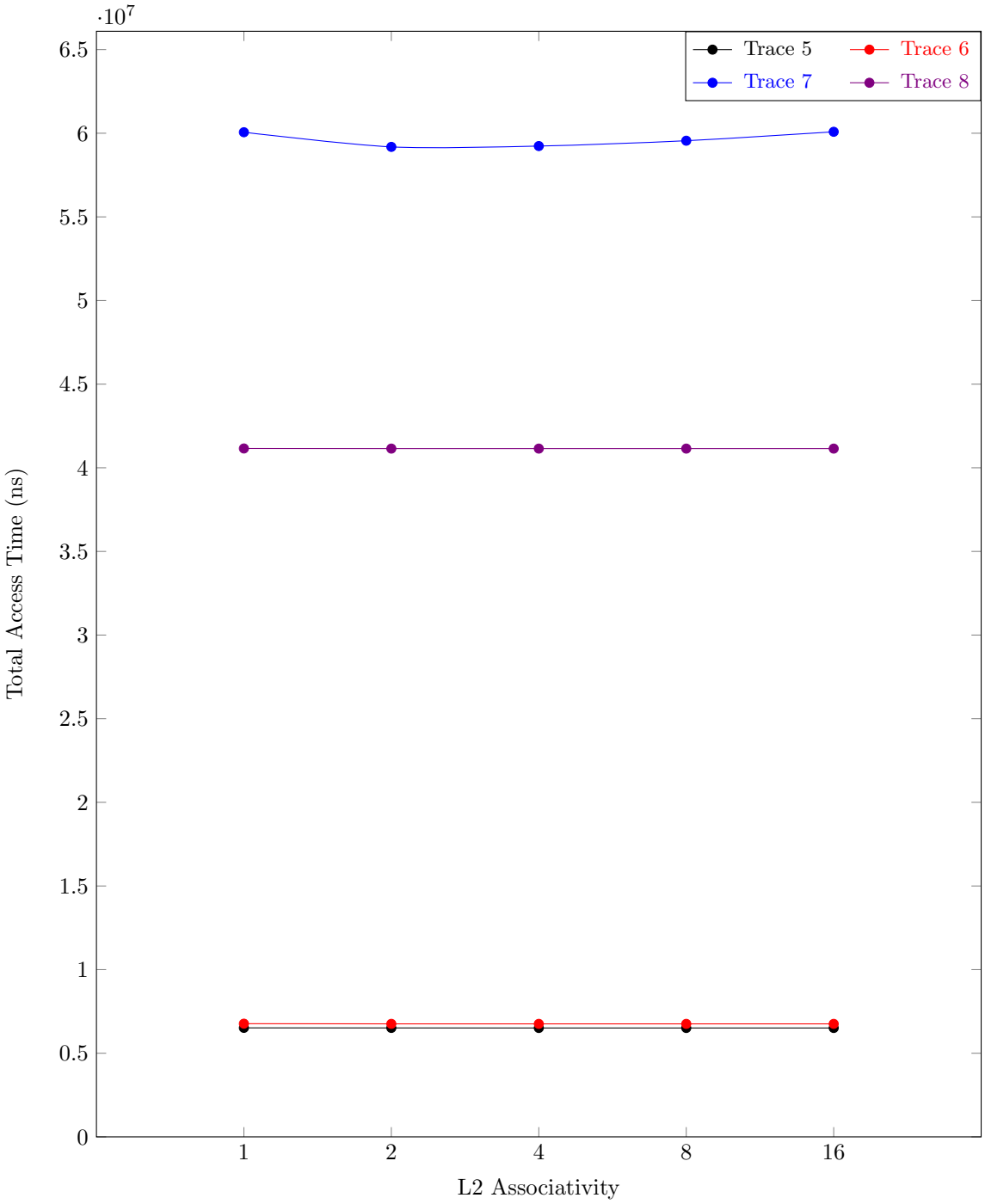
3.4.2 Observations

- We can infer that the total access time decreases up to a certain point when increasing L2 cache size. The decrease is due to the fact that number of sets in L2 cache increase with the size. But the total access time remains constant after a certain point because **increase in number of sets in L2 may not cause a significant increase in the number of hits in L2**.
- In traces 1, 3 and 4, increasing L2 cache size reduces the total access time as the instructions in these traces are reading from or writing to a **decent range of addresses**, which means that the increase in number of sets results in **more hits in L2**.
- In traces 5, 6 and 8, the total access time does not change when L2 cache size is increased as the instructions in these traces are reading from or writing to a **small range of addresses**.
- In traces 2 and 7, total access time reduces significantly with increase in L2 cache size as the instructions in these traces are reading from or writing to a **wide range of addresses**.

3.5 L2 Associativity vs Total Access Time

3.5.1 Graphs





3.5.2 Observations

- We can infer that the total access time decreases up to a certain point when increasing the L2 associativity. After which, it increases because the number of sets in the cache decrease thereby increasing the number of conflict misses.
- In traces 1, 3 and 4, increasing the L2 associativity reduces the total access time as the instructions in these traces are reading from or writing to a **decent range of addresses**. This means that with high associativity, there are **more hits in L2**.
- In traces 5, 6 and 8, increasing the L2 associativity causes a slight decrease in the total access time initially but it remains constant after that as the instructions in these traces are reading from or writing to a **small range of addresses**.
- To explain the increase observed in traces 2 and 7, refer to the Example section.

4 Example

Suppose you have two caches: C_1 in which there are 4 sets S_0,S_1,S_2 and S_3 and 2 ways and C_2 in which there is 1 set S and 8 ways. Now consider a repeated sequence of instructions of the form:

```
r X_1
r B_1
r B_2
r B_3
r X_2
r B_1
r B_2
r B_3
r X_3
.
.
.
.
.
r X_n
r B_1
r B_2
r B_3
```

where block X_i are all distinct and correspond to set S_0 in C_1 $\forall$ i and B_i corresponds to S_i in C_1 for $i \in \{1, 2, 3\}$ (obviously, all of them correspond to S in C_2). So we see that in C_1, every 1 out of 4 instructions would be a miss roughly (to make space for X_{i+1} in S_0, we would need to evict X_{i-1}), while in C_2 the blocks B_1,B_2 and B_3 would have to be evicted from S and again be brought to S periodically to make space for the blocks X_i that also get added to set S of C_2. Thus in C_2 we are **wasting time in removing blocks that would be used again in the future**. So for such a set of instructions, increasing associativity increases time.

5 Work Split

Mihir Kaskhedikar (2021CS10551)	**Rishabh Verma (2021CS10581)**
50/100	50/100