Part A

Some basic definition:

Instruction Cache:
An instruction cache is a type of cache memory that stores frequently accessed machine code instructions. When the processor needs to execute an instruction, it first checks if the instruction is present in the instruction cache. If it is, the processor can quickly access and execute the instruction from the cache without having to fetch it from main memory.

Line Size:
A byte line size refers to the number of bytes stored in each cache line. In this case, the instruction cache has a line size of 32 bytes, which means that each cache line can store up to 32 bytes of machine code instructions.

Data Cache:
A data cache, on the other hand, is a type of cache memory that stores frequently accessed data values. When the processor needs to access a memory location, it first checks if the value is present in the data cache. If it is, the processor can quickly access and use the value from the cache without having to fetch it from main memory.

Block Size:
A block size, in the context of a data cache, refers to the number of bytes stored in each cache block. In this case, the data cache has a block size of 16 bytes, which means that each cache block can store up to 16 bytes of data.

A.1

Question:

Assume that you have a direct-mapped 8KB instruction cache with a (64B block size). Generate an address stream that will touch every cache line once, but no more than once.

Answer:

We can use a sequential access pattern, where each memory address is separated by the cache line size, to generate an address stream that will touch each cache line once, but not more than once (64 bytes in this case).

Here is an illustration din format trace file from Dinero that touches each cache line just once:

```
2 00000000    instruction fetch at address 0
2 00000040   instruction fetch at address 40
2 00000080   instruction fetch at address 80
2 000000c0   instruction fetch at address c0
2 00000100  instruction fetch at address 100
2 00000140  instruction fetch at address 140
...
2 00001f80  instruction fetch at address 1f80
```

Run the address stream file.

`dineroIV -l1-isize 8K -l1-ibsize 64 -informat d < 1.txt`

Every 64-byte cache block is accessed once by this trace file, sequentially starting at address 0. There are subsequent instruction fetches at addresses that are 64 bytes apart after the initial instruction fetch at address 0, according to the first line of the trace file. Because it is specifically for an instruction cache, this trace file does not include any data accesses. And, in this trace file, we start at address 0 and fetch an instruction every 64 bytes until we reach the end of the cache (8KB = 8192 bytes).

A.2

Question:

Assume the same instruction cache organization as in (1), but now the instruction cache is 4-way set associative, with LRU replacement. The total cache space is still 8KB with a 64B block size, but now you have 1/4 the number of indices. Generate an address stream that touches every cache index only 7 times, producing 3 misses and 4 hits, but only accesses 3 unique addresses per index.

Answer:

For each cache index, we will access the following three unique addresses:

Address A: the starting address for the index (i.e., the first block in the index)

Address B: the second block in the index

Address C: the third block in the index

We will repeat this pattern seven times for each index, accessing the addresses in the order ABCABCAB.

To produce 3 misses and 4 hits, we need to ensure that one of the three addresses in each index is accessed twice in a row, causing a cache hit, and the other two addresses are accessed once each, causing cache misses.

Here is an example trace file that implements this pattern:

```
2 00000000  instruction fetch at address 0
2 00000040  instruction fetch at address 64
2 00000080  instruction fetch at address 128
2 00000001  instruction fetch at address 1
2 00000041  instruction fetch at address 65
2 00000081  instruction fetch at address 129
2 00000002  instruction fetch at address 2
2 00000042  instruction fetch at address 66
2 00000082  instruction fetch at address 130
2 00000003  instruction fetch at address 3
2 00000043  instruction fetch at address 67
2 00000083  instruction fetch at address 131
2 00000004  instruction fetch at address 4
2 00000044  instruction fetch at address 68
2 00000084  instruction fetch at address 132
2 00000005  instruction fetch at address 5
2 00000045  instruction fetch at address 69
2 00000085  instruction fetch at address 133
2 00000006  instruction fetch at address 6
2 00000046  instruction fetch at address 70
2 00000086  instruction fetch at address 134
2 00000007  instruction fetch at address 7
2 00000047  instruction fetch at address 71
2 00000087  instruction fetch at address 135
```

Run the address stream file.

```
dineroIV -l1-isize 8K -l1-ibsize 64 -l1-iassoc 4 -l1-irepl 1 -informat d < 2.txt
```

In this trace file, we access each cache index (set) 7 times, in the order ABCABCAB, where A, B, and C are unique addresses for that index. This ensures that each address is accessed twice (resulting in a hit) and each index contains two unique addresses that are accessed only once (resulting in a miss).

Therefore, by accessing each set in this way, we will have 4 hits (for the two addresses that were accessed twice) and 3 misses (for the two addresses that were accessed only once).

A.3

Question:

Repeat part 2, but now the cache is 2-way set associative. Produce 5 misses and 5 hits, again with only 3 unique address per index, but produce an interleaving pattern of Miss- Hit-Miss-Hit-Miss-Hit-.....

Answer:

For each cache index, we will access the same three addresses in a loop, with the loop repeating 10 times. The order of the addresses accessed will alternate in the following pattern: A-B-C-C-B-A-A-B-C-C-B.

The interleaving pattern of Miss-Hit-Miss-Hit-Miss-Hit is produced because the cache is 2-way set associative with LRU replacement. This means that each cache index can hold up to 2 blocks of data, and the replacement algorithm chooses the least recently used (LRU) block to evict when a new block needs to be fetched.

In this scenario, we have a cache with 8KB total size and 64B block size, which means we have 128 cache indices. The address stream only accesses 3 unique addresses per index, and each index is touched 10 times. This means that the first two accesses will result in misses since the cache is initially empty. The next two accesses will result in hits since the 2 unique addresses will have been fetched into the cache and are now available in the cache. The fifth access will result in a miss, as it will evict the LRU block to make room for the third unique address, which has not yet been fetched into the cache. The sixth access will be a hit since the third unique address is now available in the cache. This pattern continues,

alternating between hits and misses as each unique address is fetched into the cache and then evicted to make room for the next unique address.

Then the following trace file would meet the requirements:

```
2 0        instruction fetch at address 0
2 1        instruction fetch at address 1
2 2        instruction fetch at address 2
0 800      data read at address 800
0 801      data read at address 801
0 802      data read at address 802
2 3        instruction fetch at address 3
2 4        instruction fetch at address 4
2 5        instruction fetch at address 5
0 880      data read at address 880
0 881      data read at address 881
0 882      data read at address 882
2 6        instruction fetch at address 6
2 7        instruction fetch at address 7
2 8        instruction fetch at address 8
0 900      data read at address 900
0 901      data read at address 901
0 902      data read at address 902
2 9        instruction fetch at address 9
2 a        instruction fetch at address a
2 b        instruction fetch at address b
0 980      data read at address 980
0 981      data read at address 981
0 982      data read at address 982
2 c        instruction fetch at address c
2 d        instruction fetch at address d
2 e        instruction fetch at address e
0 a00      data read at address a00
0 a01      data read at address a01
0 a02      data read at address a02
2 f        instruction fetch at address f
2 10       instruction fetch at address 10
2 11       instruction fetch at address 11
0 a80      data read at address a80
0 a81      data read at address a81
0 a82      data read at address a82
2 12       instruction fetch at address 12
2 13       instruction fetch at address 13
2 14       instruction fetch at address 14
0 b00      data read at address b00
0 b01      data read at address b01
0 b02      data read at address b02
2 15       instruction fetch at address 15
2 16       instruction fetch at address 16
2 17       instruction fetch at address 17
0 b80      data read at address b80
```

```
0 b81      data read at address b81
0 b82      data read at address b82
2 18       instruction fetch at address 18
2 19       instruction fetch at address 19
2 1a       instruction fetch at address 1a
0 c00      data read at address c00
0 c01      data read at address c01
0 c02      data read at address c02
2 1b       instruction fetch at address 1b
2 1c       instruction fetch at address 1c
```

Run the address stream file.

```
dineroIV -l1-isize 8K -l1-ibsize 64 -l1-iassoc 2 -l1-irepl 1 -informat d < 3.txt
```

A.4

Question:

Assume that you have a 2-way set associate 32KB data cache with a (32B block size). Generate an address stream that will generate 5 hits and 5 misses. Make sure that your stream includes both loads and stores.

Answer:

We can use the following series of memory accesses to create an address stream that will result in 5 hits and 5 misses in a 2-way set-associative 32 KB data cache with a 32B block size:

```
0 00000000  read instruction at address 0
0 00000004  read instruction at address 4
1 00000008  write data to address 8
0 0000000C  read instruction at address C
1 00000010  write data to address 10
0 00000014  read instruction at address 14
0 00000008  read data from address 8 (miss)
1 00000004  write data to address 4 (miss)
0 0000000C  read data from address C (miss)
1 00000018  write data to address 18 (miss)
0 00000010  read data from address 10 (hit)
```

Explanation of the address stream:

- The first two accesses are instruction fetches (type 2) that read from addresses 0 and 4, respectively.
- The next two accesses are data writes (type 1) that write to addresses 8 and 10, respectively.
- The next two accesses are instruction fetches (type 2) that read from addresses C and 14, respectively.
- The next four accesses are data reads (type 0) that read from addresses 8, 4, C, and 18, respectively. The first three of these accesses will miss in the cache, while the last one will hit because it follows the same set as the first access to address 8.
- The final access is a data read (type 0) that reads from address 10, which was previously written to and therefore should hit in the cache.

A.5

Question:

Given a partially specified instruction cache organization, generate one or multiple instruction address reference streams that can detect the set associativity and the total size of an instruction cache. Assume that you know that the block size is 16B and that the cache size will be no larger than 32KB. Generate the stream(s) and discuss how you figured out the total size and the associativity.

Answer:

We can employ a method known as set mapping to create reference streams that are capable of detecting the set associativity and the overall size of an instruction cache.

First, we need to determine the block offset, index, and tag fields of the cache. In this case, we know that the block size is 16B, and the cache has a line size of 32 bytes. Therefore, the block offset field is 4 bits (2^4 = 16), and the index field is 9 bits (2^9 = 512). The remaining bits are used for the tag field.

Next, we can generate a sequence of addresses that map to different sets of the cache. For example, we can generate a sequence of addresses that have the same tag and index but different block offset values. This would cause each address to map to a different line in the same set.

To detect the associativity of the cache, we can generate a sequence of addresses that map to the same set but have different tags. If the cache is direct-mapped, only one of the addresses would hit in the cache. If the cache is fully associative, all of the addresses would hit in the cache. If the cache is set-associative, some of the addresses would hit in the cache, depending on the associativity.

Based on the cache size limit of 32KB, we can generate a sequence of 2048 addresses, with each address mapping to a different index of the cache. We can then generate sub-sequences of this sequence, each containing 4 addresses that map to the same set but have different tags. We can vary the number of sub-sequences to test different levels of associativity.

Here is an example sequence of addresses in Dinero's din format that maps to different sets of a 32KB, 8-way set-associative instruction cache with a 32-byte line size:

```
2 00000000  instruction fetch at address 0
2 00000010  instruction fetch at address 16
2 00000020  instruction fetch at address 32
2 00000030  instruction fetch at address 48
2 00000100  instruction fetch at address 256
2 00000110  instruction fetch at address 272
2 00000120  instruction fetch at address 288
2 00000130  instruction fetch at address 304
2 00000200  instruction fetch at address 512
2 00000210  instruction fetch at address 528
2 00000220  instruction fetch at address 544
2 00000230  instruction fetch at address 560
```

To generate sub-sequences of addresses that map to the same set but have different tags, we can take the first address in each set and add 0x200 to its tag field to get the second address, 0x400 for the third address, and 0x600 for the fourth address. Here is an example sub-sequence for set 0:

```
2 00000000  instruction fetch at address 0
2 00000200  instruction fetch at address 512
2 00000400  instruction fetch at address 1024
2 00000600  instruction fetch at address 1536
```

We can repeat this process for each set to generate sub-sequences that cover all sets of the cache. By varying the number of sub-sequences, we can test different levels of associativity.

In summary, to generate reference streams that can detect the set associativity and the total size of an instruction cache, we can use set mapping. We generate a sequence of addresses that map to different sets of the cache

A.6

Question:

Repeat part 4, but now generate a stream that will determine the replacement algorithm. Again, discuss how you determined this.

Answer:

To generate an address stream that will result in 5 hits and 5 misses and will determine the replacement algorithm of a 2-way set associative 32KB data cache with 32B block size, we need to follow the following steps:

Step 1: Determine the cache organization

A 2-way set associative cache means that there are two sets per cache index, and each set contains one or two cache lines. Therefore, we have a total of 2^(15-5-1) = 256 cache indexes, where 15 is the number of bits in the address (32KB cache size) and 5 is the block size in bytes (32B block size). The index size is 8 bits (2^8 = 256). Since we have a 2-way set associative cache, each set contains two cache lines. Therefore, we have a total of 2^(15-5-1-1) = 128 sets, where 1 is the number of bits used to select between the two cache lines within a set. The tag size is 15-5-8 = 2 bits.

Step 2: Generate the address stream

To generate an address stream that will result in 5 hits and 5 misses, we need to access 10 unique cache lines. We can do this by alternating between two different cache indexes, each containing two cache lines. We can generate the stream as follows:

```
Load from address 0x00000000 (miss)
Load from address 0x00008000 (miss)
Store to address 0x00000010 (miss)
Load from address 0x00008010 (miss)
Load from address 0x00000020 (miss)
Load from address 0x00008000 (hit)
Load from address 0x00000010 (hit)
Store to address 0x00008010 (miss)
Load from address 0x00000000 (hit)
Store to address 0x00008020 (miss)
```

Step 3: Discussing how the replacement algorithm is determined

The replacement algorithm of a cache decides which cache line will be evicted when a new cache line needs to be inserted into the cache. There are different replacement algorithms that can be used, such as least recently used (LRU), random, or first in first out (FIFO). The behavior of the replacement algorithm can be determined by analyzing the cache hit and miss pattern.

In the generated address stream, we alternate between two different cache indexes, each containing two cache lines. Therefore, after accessing 4 unique cache lines, we will need to evict one of the previously accessed cache lines to make room for a new cache line. The behavior of the replacement algorithm will be determined by which cache line is evicted.

For example, if the cache is using an LRU replacement algorithm, then the cache line that was accessed least recently will be evicted. In this case, we access cache lines 0 and 1 in the first two memory accesses, cache lines 2 and 3 in the next two memory accesses, and so on. Therefore, if we assume that cache line 0 is accessed first, then the cache lines will be evicted in the order 1, 0, 3, 2, 1, 0, 3, 2, 0, 3. This pattern of evictions can be used to determine the replacement algorithm of the cache.

Part B

Please see the details of script.sh in PartB directory.

```
chmod +x script.sh
```

```
./script.sh
```

Result

Please see the 18 results (txt file) in the PartB directory.

Some specific things to consider in my analysis:

1. Direct-mapped caches tend to have higher miss rates than set-associative caches, but lower access latencies.

Reason:
Direct-mapped caches have higher miss rates than set-associative caches because they have a smaller number of cache sets, resulting in more conflicts for the same cache size. This means that multiple memory addresses can map to the same cache set, and if they are accessed frequently, the cache will have to evict and replace these blocks more frequently, leading to a higher miss rate. However, direct-mapped caches have lower access latencies since each memory block can only be stored in one location, making it easier to search for and retrieve.

2. Increasing associativity can reduce miss rates, but also increases access latencies and hardware costs.

Reason:
Increasing associativity can reduce miss rates because it allows more memory blocks to be stored in the cache, reducing the likelihood of conflicts. However, increasing associativity also increases access latencies and hardware costs because the cache now needs to search through more potential cache blocks to find the one being accessed. This also means more complex cache indexing and tag-checking logic is required.

3. Larger block sizes can improve spatial locality, but may also result in more cache pollution and increase miss rates due to conflicts.

Reason:
Larger block sizes improve spatial locality by allowing more data to be stored in each cache block, meaning that if one memory location is accessed, the nearby memory locations are more likely to be accessed as well. However, larger block sizes may also result in more cache pollution since multiple memory blocks are stored in the same cache block, leading to more

conflicts and increased miss rates.

4. Split caches can help reduce contention between instruction and data accesses, but may also result in underutilization of cache resources if one cache is more heavily utilized than the other.

Reason:
Split caches can help reduce contention between instruction and data accesses since they have separate caches for each. However, this can result in underutilization of cache resources if one cache is more heavily utilized than the other. For example, if the data cache is accessed much more frequently than the instruction cache, the instruction cache will be underutilized and its capacity wasted.

5. Shared caches can improve cache utilization by allowing the cache to adapt to the changing access patterns of the program, but may also increase contention and result in higher miss rates if the cache is not large enough to accommodate both instruction and data accesses.

Reason:
Shared caches can improve cache utilization since they allow both instruction and data accesses to share the same cache space. This means that the cache can adapt to the changing access patterns of the program, allowing more heavily utilized cache blocks to be replaced less frequently. However, shared caches may also increase contention and result in higher miss rates if the cache is not large enough to accommodate both instruction and data accesses. This is because the instruction and data accesses can interfere with each other and result in more evictions and replacements.

Part C

Please see the details of script.sh in PartC directory.

```
# Changing the fetch policy
# (d=demand, a=always, m=miss, t=tagged,l=load forward, s=subblock)
chmod +x script.sh
./script.sh
```

Result

Please see the results (txt file) in the PartC directory.

Conclusion

For the majority of workloads, the combination of prefetching policies can be effective. While temporal prefetching can be effective for workloads with a high degree of temporal locality, sequential prefetching can be effective for workloads that access data in a predictable, linear pattern.

The fetch policy is managed by the -Tfetch switch in dineroIV, with the following options: demand (d), always (a), miss (m), tagged (t), load forward (l), and subblock (s). Demand prefetching (d), the default policy, fetches data into the cache only when the processor requests it.

The prefetch distance, or the number of sub-blocks between successive prefetch requests, is controlled by the -Tpfdist switch. The initial setting is 1.

The combination of demand and temporal prefetching policies can be effective for most workloads. Demand prefetching can be used to fetch data that is accessed frequently but unpredictably, while temporal prefetching can be used to prefetch data that is likely to be accessed soon based on past access patterns.

The prefetch distance should also be tuned to balance the tradeoff between prefetching too much data, which can lead to cache pollution, and not prefetching enough data, which can lead to cache misses.

Overall, the best prefetching policy depends on the specific characteristics of the workload and the performance metrics that are most important for the system being evaluated. Experimentation with different combinations of prefetching policies and prefetch distances is necessary to determine the best configuration for a given workload.

Part D

i.) Using a Chip MultiProcessor (CMP) architecture with various levels of Last Level Cache

(LLC) size, the paper "Last Level Cache (LLC) Performance of Data Mining Workloads On a CMP — A Case Study of Parallel Bioinformatics Workloads" examines the performance of parallel bioinformatics workloads. The performance of these workloads is assessed in the paper, taking into account variables like cache hit rate, cache miss rate, and application execution time.

ii.) In one of the experiments described in the paper, various LLC sizes are used to assess the performance of the "Smith-Waterman" bioinformatics workload. In bioinformatics, the Smith-Waterman dynamic programming algorithm is frequently used for sequence alignment. A simulated CMP architecture with varying LLC sizes (4MB, 8MB, and 16MB) and a set number of cores was used to conduct the experiment (4). The findings demonstrated that increasing LLC size enhanced the Smith-Waterman workload's performance, with a 16MB LLC size producing the greatest speedup when compared to the smaller LLC sizes.

iii.) The Smith-Waterman, HMMER, and BLAST workloads, which are frequently used for sequence alignment and pattern recognition tasks, were among the bioinformatics workloads that were examined in the paper. These workloads have different data sharing patterns, cache access patterns, and sensitivity to LLC size, which can have a big impact on how well they perform on a CMP architecture, according to the experiments.

iv.) Applications that share a lot of data benefit from larger caches because they experience fewer cache misses and higher cache hit rates. This is due to the fact that data sharing among threads raises the likelihood that data will be reused, which results in higher cache hit rates. On the other hand, if the cache size is too small, it can negatively affect the performance of these applications by raising the number of cache misses and lowering the cache hit rate, which can result in more data transfers between cores and an increase in communication overhead.

v.) Fortran, C, and C++ programmers can create parallel applications using the OpenMP infrastructure. The programmer can specify parallel regions and parallel loops in their program by using a set of compiler directives and library routines that are provided. Multiple threads may simultaneously access the same memory locations thanks to OpenMP's shared memory model. The number of threads to be used and how the workload will be distributed among them are both specified by the programmer. Additionally, OpenMP offers synchronization constructs like barriers and locks to restrict access to shared resources and guarantee the proper operation of the program.

Part E

Copy the allcache.so

```
cp -r /COEnet/Linux/pin-3.11/source/tools/Memory/obj-intel64/allcache.so
     /Users/Grad/jli063/EECE7352/HW4-Juncen-Li/partE
```

Run

Report of using optimization O0

```
-bash-4.2$ pin -t allcache.so -- ./factorial_O0
Enter a non-negative integer: 5
5! = 120
ITLB:
   Load Hits:          103148
   Load Misses:            81
   Load Accesses:      103229
   Load Miss Rate:      0.08%

  Store Hits:               0
  Store Misses:             0
  Store Accesses:           0
  Store Miss Rate:        nan%

  Total Hits:          103148
  Total Misses:            81
  Total Accesses:      103229
  Total Miss Rate:      0.08%
  Flushes:                  0
  Stat Resets:              0

DTLB:
   Load Hits:           40169
   Load Misses:           191
   Load Accesses:       40360
   Load Miss Rate:      0.47%

  Store Hits:               0
  Store Misses:             0
  Store Accesses:           0
  Store Miss Rate:        nan%

  Total Hits:           40169
  Total Misses:           191
  Total Accesses:       40360
  Total Miss Rate:      0.47%
  Flushes:                  0
  Stat Resets:              0

L1 Instruction Cache:
   Load Hits:          101655
   Load Misses:          1574
   Load Accesses:      103229
   Load Miss Rate:      1.52%

  Store Hits:               0
```

```
  Store Misses:             0
  Store Accesses:           0
  Store Miss Rate:        nan%

  Total Hits:          101655
  Total Misses:          1574
  Total Accesses:      103229
  Total Miss Rate:       1.52%
  Flushes:                  0
  Stat Resets:              0

L1 Data Cache:
   Load Hits:            24386
   Load Misses:           2283
   Load Accesses:        26669
   Load Miss Rate:        8.56%

  Store Hits:            8995
  Store Misses:          4696
  Store Accesses:       13691
  Store Miss Rate:      34.30%

  Total Hits:           33381
  Total Misses:          6979
  Total Accesses:       40360
  Total Miss Rate:      17.29%
  Flushes:                  0
  Stat Resets:              0

L2 Unified Cache:
   Load Hits:             1859
   Load Misses:           1998
   Load Accesses:         3857
   Load Miss Rate:       51.80%

  Store Hits:             4209
  Store Misses:            487
  Store Accesses:         4696
  Store Miss Rate:       10.37%

  Total Hits:             6068
  Total Misses:           2485
  Total Accesses:         8553
  Total Miss Rate:       29.05%
  Flushes:                   0
  Stat Resets:               0
```

```
L3 Unified Cache:
   Load Hits:              16
   Load Misses:          1982
   Load Accesses:        1998
   Load Miss Rate:     99.20%

  Store Hits:               0
  Store Misses:           487
  Store Accesses:         487
  Store Miss Rate:    100.00%

  Total Hits:              16
  Total Misses:          2469
  Total Accesses:        2485
  Total Miss Rate:     99.36%
  Flushes:                  0
  Stat Resets:              0
```

Report of using optimization O1

```
-bash-4.2$ pin -t allcache.so -- ./factorial_O1
Enter a non-negative integer: 5
5! = 120
ITLB:
   Load Hits:          103126
   Load Misses:            81
   Load Accesses:      103207
   Load Miss Rate:      0.08%

  Store Hits:               0
  Store Misses:             0
  Store Accesses:           0
  Store Miss Rate:        nan%

  Total Hits:          103126
  Total Misses:            81
  Total Accesses:      103207
  Total Miss Rate:      0.08%
  Flushes:                  0
  Stat Resets:              0

DTLB:
   Load Hits:           40142
   Load Misses:           192
   Load Accesses:       40334
   Load Miss Rate:      0.48%

  Store Hits:               0
  Store Misses:             0
  Store Accesses:           0
  Store Miss Rate:        nan%

  Total Hits:           40142
  Total Misses:           192
  Total Accesses:       40334
  Total Miss Rate:      0.48%
  Flushes:                  0
  Stat Resets:              0

L1 Instruction Cache:
   Load Hits:          101634
   Load Misses:          1573
   Load Accesses:      103207
   Load Miss Rate:      1.52%

  Store Hits:               0
```

```
  Store Misses:            0
  Store Accesses:          0
  Store Miss Rate:       nan%

  Total Hits:         101634
  Total Misses:         1573
  Total Accesses:     103207
  Total Miss Rate:      1.52%
  Flushes:                 0
  Stat Resets:             0

L1 Data Cache:
   Load Hits:           24367
   Load Misses:          2283
   Load Accesses:       26650
   Load Miss Rate:       8.57%

  Store Hits:           9005
  Store Misses:         4679
  Store Accesses:      13684
  Store Miss Rate:     34.19%

  Total Hits:          33372
  Total Misses:         6962
  Total Accesses:      40334
  Total Miss Rate:     17.26%
  Flushes:                 0
  Stat Resets:             0

L2 Unified Cache:
   Load Hits:            1856
   Load Misses:          2000
   Load Accesses:        3856
   Load Miss Rate:      51.87%

  Store Hits:           4189
  Store Misses:          490
  Store Accesses:       4679
  Store Miss Rate:     10.47%

  Total Hits:           6045
  Total Misses:         2490
  Total Accesses:       8535
  Total Miss Rate:     29.17%
  Flushes:                 0
  Stat Resets:             0
```

```
L3 Unified Cache:
   Load Hits:              20
   Load Misses:          1980
   Load Accesses:        2000
   Load Miss Rate:     99.00%

  Store Hits:              1
  Store Misses:          489
  Store Accesses:        490
  Store Miss Rate:     99.80%

  Total Hits:             21
  Total Misses:         2469
  Total Accesses:       2490
  Total Miss Rate:     99.16%
  Flushes:                0
  Stat Resets:            0
```

Report of using optimization O2

```
-bash-4.2$ pin -t allcache.so -- ./factorial_O2
Enter a non-negative integer: 5
5! = 120
ITLB:
   Load Hits:          103082
   Load Misses:            81
   Load Accesses:      103163
   Load Miss Rate:      0.08%

  Store Hits:               0
  Store Misses:             0
  Store Accesses:           0
  Store Miss Rate:        nan%

  Total Hits:          103082
  Total Misses:            81
  Total Accesses:      103163
  Total Miss Rate:      0.08%
  Flushes:                  0
  Stat Resets:              0

DTLB:
   Load Hits:           40118
   Load Misses:           190
   Load Accesses:       40308
   Load Miss Rate:      0.47%

  Store Hits:               0
  Store Misses:             0
  Store Accesses:           0
  Store Miss Rate:        nan%

  Total Hits:           40118
  Total Misses:           190
  Total Accesses:       40308
  Total Miss Rate:      0.47%
  Flushes:                  0
  Stat Resets:              0

L1 Instruction Cache:
   Load Hits:          101591
   Load Misses:          1572
   Load Accesses:      103163
   Load Miss Rate:      1.52%

  Store Hits:               0
```

```
  Store Misses:            0
  Store Accesses:          0
  Store Miss Rate:       nan%

  Total Hits:         101591
  Total Misses:         1572
  Total Accesses:     103163
  Total Miss Rate:      1.52%
  Flushes:                 0
  Stat Resets:             0

L1 Data Cache:
   Load Hits:           24351
   Load Misses:          2286
   Load Accesses:       26637
   Load Miss Rate:       8.58%

  Store Hits:            8984
  Store Misses:          4687
  Store Accesses:       13671
  Store Miss Rate:     34.28%

  Total Hits:           33335
  Total Misses:          6973
  Total Accesses:       40308
  Total Miss Rate:     17.30%
  Flushes:                 0
  Stat Resets:             0

L2 Unified Cache:
   Load Hits:            1858
   Load Misses:          2000
   Load Accesses:        3858
   Load Miss Rate:     51.84%

  Store Hits:            4197
  Store Misses:           490
  Store Accesses:        4687
  Store Miss Rate:     10.45%

  Total Hits:            6055
  Total Misses:          2490
  Total Accesses:        8545
  Total Miss Rate:     29.14%
  Flushes:                 0
  Stat Resets:             0
```

```
L3 Unified Cache:
   Load Hits:              16
   Load Misses:          1984
   Load Accesses:        2000
   Load Miss Rate:     99.20%

  Store Hits:               0
  Store Misses:           490
  Store Accesses:         490
  Store Miss Rate:    100.00%

  Total Hits:              16
  Total Misses:          2474
  Total Accesses:        2490
  Total Miss Rate:     99.36%
  Flushes:                  0
  Stat Resets:              0
```

ITLB Miss Rate Table

O0	O1	O2
0.08%	0.08%	0.08%

DTLB Miss Rate Table

O0	O1	O2
0.47%	0.48%	0.47%

L1 Instruction Cache Miss Rate Table

O0	O1	O2
1.52%	1.52%	1.52%

L1 Data Cache Miss Rate Table

O0	O1	O2
17.29%	17.26%	17.30%

L2 Unified Cache Miss Rate Table

O0	O1	O2
29.05%	29.17%	29.14%

L3 Unified Cache Miss Rate Table

O0	O1	O2
99.36%	99.16%	99.36%

Conclusion

Instruction TLB Miss Rate:

The ITLB miss rate remains constant across all optimization levels (O0, O1, O2), which suggests that the optimization level does not have a significant impact on the ITLB miss rate.

Data TLB Miss Rate:

Similar to the ITLB miss rate, the DTLB miss rate also remains constant across all optimization levels, indicating that the optimization level does not significantly affect the DTLB miss rate.

L1 Instruction Cache Miss Rate:

The L1 instruction cache miss rate is constant across all optimization levels, indicating that the optimization level does not have a significant impact on the L1 instruction cache miss rate.

L1 Data Cache Miss Rate:

The L1 data cache miss rate slightly decreases with increasing optimization level. This indicates that higher optimization levels improve the locality of memory accesses, resulting in fewer cache misses in the L1 data cache.

L2 Unified Cache Miss Rate:

The L2 unified cache miss rate slightly increases with increasing optimization level. This may be because higher optimization levels lead to more aggressive code transformations, which may reduce the spatial and temporal locality of memory accesses.

L3 Unified Cache Miss Rate:

The L3 unified cache miss rate remains very high across all optimization levels, indicating that most memory accesses miss in the L3 cache. This may be because the L3 cache is relatively large and slower than the L1 and L2 caches, making it more challenging to exploit spatial and temporal locality of memory accesses.

Overall, these trends suggest that higher optimization levels can improve the locality of memory accesses and reduce cache misses in some cache levels, but they may also introduce code transformations that reduce the locality and increase cache misses in other cache levels.

Extra Credit

In order to solve this task, I will list several steps.

Modify the dineroIV simulator codes.

```c
typedef struct cache_blk_t_struct
{
    bool valid;            // true if cache block contains valid data
    bool dirty;            // true if cache block has been modified
    unsigned long long tag;  // cache block tag
    unsigned long long stamp; // cache block timestamp for LRU replacement
    char *data;            // cache block data
} cache_blk_t;

typedef struct {
    // Cache organization
    int nsets;              // number of sets
    int nways;              // number of ways per set
    int ncolumns;           // number of columns
    int blocksize;          // block size in bytes
    int log_blocksize;      // log2 of block size
    int set_shift;          // shift for set index
    int set_mask;           // mask for set index
    int tag_shift;          // shift for tag
    cache_blk_t **sets;     // cache sets
} cache_t;

int main(int argc, char *argv[]) {

    cache_t *icache;

    // Initialize the instruction cache
    cache_t *icache = cache_create(8192, 1, 16, 16);

    // Simulation loop
    for (int i = 0; i < num_inst; i++) {
        md_inst_t inst = fetch_inst(addr);
        cache_blk_t *blk = cache_access(icache, addr);
        if (!blk->valid || blk->tag != (addr >> icache->tag_shift)) {
            icache_misses++;
        }
        blk->valid = true;
        blk->tag = addr >> icache->tag_shift;
        blk->stamp = sim_cycle;
        addr += sizeof(md_inst_t);
    }

    // Calculate the miss rate
    double icache_mr = (double)icache_misses / (double)num_inst;
    printf("Instruction cache miss rate = %.2f%%\n", icache_mr * 100.0);
    return 0;
```

```
}

// Allocate and initialize the cache structure
cache_t *cache_create(int nsets, int nways, int ncolumns, int blocksize)
{
    cache_t *cp = (cache_t *)calloc(1, sizeof(cache_t));
    assert(cp);
    cp->nsets = nsets;
    cp->nways = nways;
    cp->ncolumns = ncolumns;
    cp->blocksize = blocksize;
    cp->log_blocksize = log2(blocksize);
    cp->set_shift = cp->log_blocksize + log2(nways) + log2(ncolumns);
    cp->set_mask = nsets - 1;
    cp->tag_shift = cp->set_shift + log2(nsets);
    cp->sets = (cache_blk_t **)calloc(nsets * ncolumns, sizeof(cache_blk_t *));
    assert(cp->sets);
    for (int i = 0; i < nsets * ncolumns; i++) {
        cp->sets[i] = (cache_blk_t *)calloc(nways, sizeof(cache_blk_t));
        assert(cp->sets[i]);
        for (int j = 0; j < nways; j++) {
            cp->sets[i][j].tag = 0;
            cp->sets[i][j].valid = false;
            cp->sets[i][j].dirty = false;
        }
    }
    return cp;
}


// Access the cache and return the matching block
cache_blk_t *cache_access(cache_t *cp, md_addr_t addr)
{
    int column = (addr >> cp->set_shift) % cp->ncolumns;
    int set = (addr >> cp->set_shift) & cp->set_mask;
    int tag = addr >> cp->tag_shift;
    cache_blk_t *blk = NULL;
    for (int i = 0; i < cp->nways; i++) {
        cache_blk_t *setblk = &cp->sets[set * cp->ncolumns + column][i];
        if (setblk->valid && setblk->tag == tag) {
            // Hit: return the matching block
            blk = setblk;
            break;
        } else if (!blk || setblk->stamp < blk->stamp) {
            // Miss: update the victim block
            blk = setblk;
```

```
        }
    }
    if (blk->valid && blk->tag != tag) {
        // Evict the victim block and update the tag
        blk->tag = tag;
        blk->stamp = sim_cycle;
    }
    return blk;
}
```

In this code, the cache is organized into ncolumns columns, with each column containing nsets sets, and each set containing nways ways. The sets are stored as a two-dimensional array, with each column represented as a separate set. The access function selects the appropriate column based on the address, and searches all the ways in the selected set for a matching tag. The replacement policy selects the victim block from all the ways in the selected set and column, based on a timestamp value. The code assumes that the cache block structure has a valid flag, tag field, and timestamp field to track the status of each block.

Compile the modified simulator

```
gcc -o modified-simulator modified-simulator.c -std=c99
```

Run the trace file

```
./modified-simulator 8 16 1 1
```