18344 Lab 2: Implement a Memory Hierarchy

Eli Wirth-Apley (ewirthap)
Chalisa Udompanyawit (cudompan)

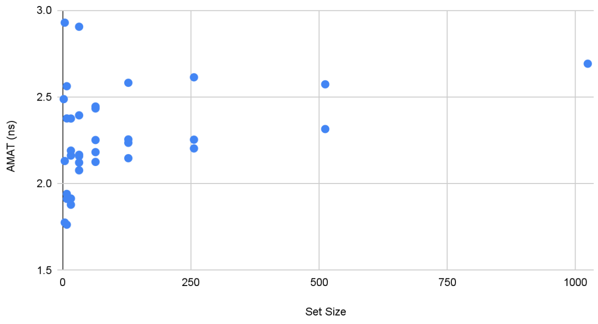
Task 1: Implementing a set associative cache

In this task, we found the Average Memory Access Time (AMAT) values that were simulated across 4 cache sizes (1KB, 4KB, 16KB, 262KB, and 4MB). The simulated cache loads were produced by the gcc_s benchmark. Throughout all tasks in this lab, we are working entirely from the gcc_s benchmark, primarily due to time constraints. The four cache sizes were simulated from direct-mapped (one-way associativity) up to fully-mapped (fully-associativity) with varying block sizes. However, we found out that Destiny only allows the simulation for certain configurations. Destiny does now allow for caches with associativity greater than the square root of the total cache size. The block size also limits whether a cache is able to be simulated by Destiny. Therefore, the data that we will be representing is being simulated with a maximum associativity of 128 lines per set.

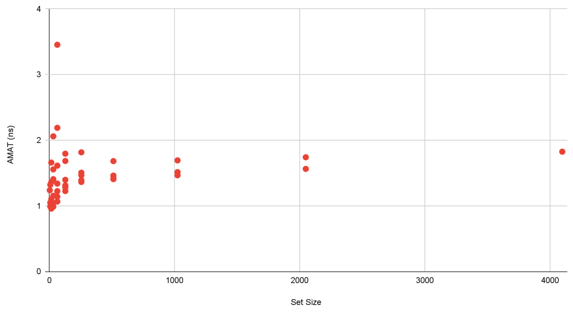
We have run the sensitivity study in this task, and the results are shown by comparing the AMAT vs the various cache parameters per each cache size. In the graphs below, we show AMAT vs Set Size, AMAT vs Associativity and AMAT vs Block Size. From the data, we see that there are clusters of optimal values based on minimal AMAT.

From the graphs we can see that the AMAT is pretty high on the edge case values, especially in the associativity graph. We can see that the trend is like a parabola, that the curve bottom lies around the lower end of the x-axis. We also suspect that there are data points that have comparatively low AMAT for increasing set sizes, but not fully associative. This cannot be confirmed due to limitations in Destiny. From our data sheet, we see that the trial in which there is a balance between the number of sets and lines mostly produces lower average access rate.

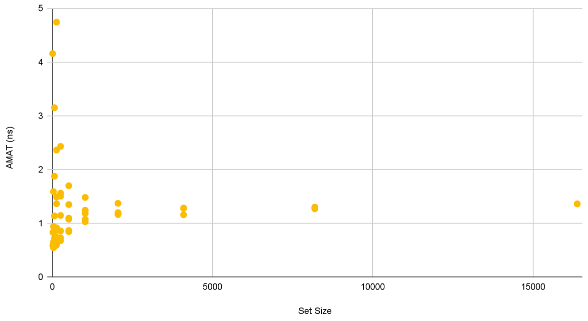
AMAT vs. Set Size 1024B Cache



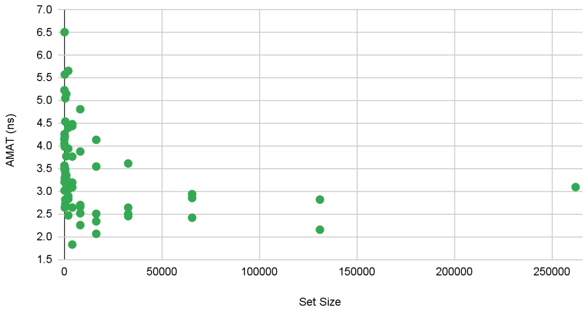
AMAT vs. Set Size 4096B Cache



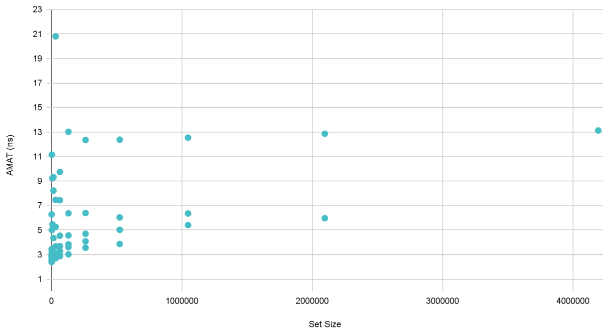
AMAT vs. Set Size 16384B Cache

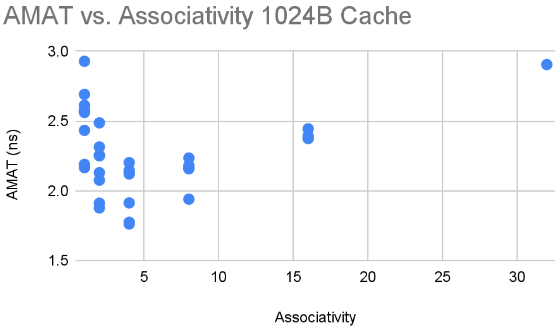


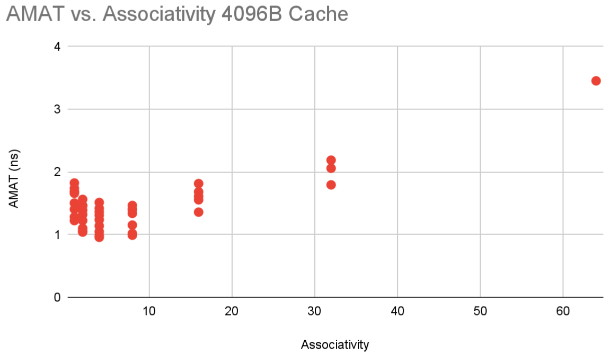
AMAT vs. Set Size 262144B Cache

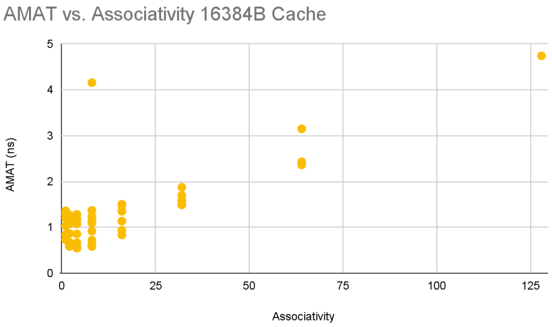


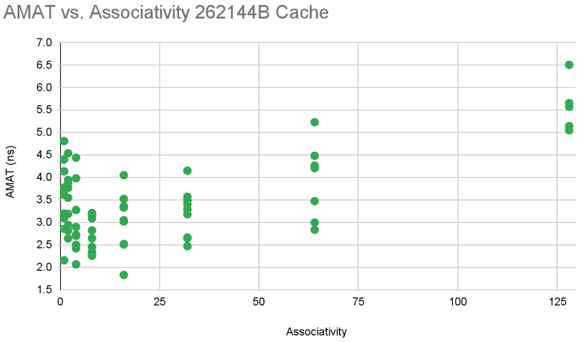
AMAT vs. Set Size 4194304B Cache

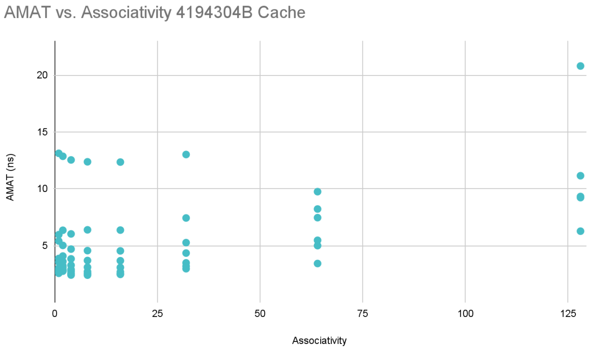



AMAT vs. Associativity 1024B Cache

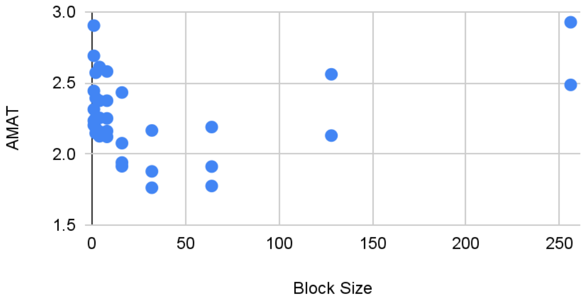

AMAT vs. Associativity 4096B Cache


AMAT vs. Associativity 16384B Cache

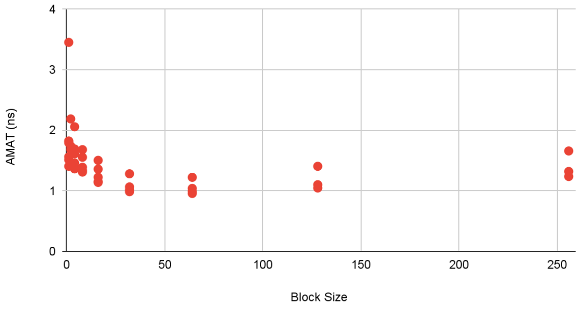

AMAT vs. Associativity 262144B Cache


AMAT vs. Associativity 4194304B Cache

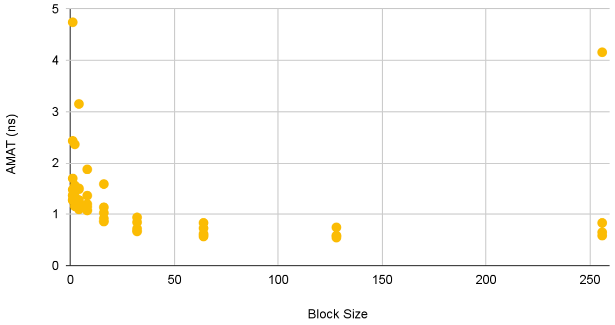
AMAT vs. Block Size 1024B Cache



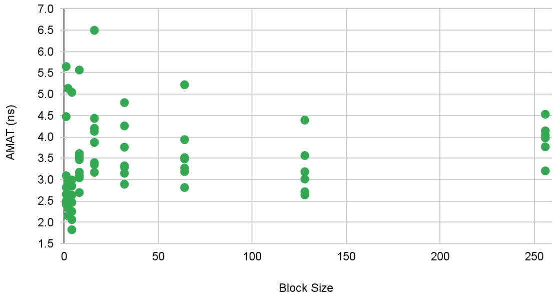
AMAT vs. Block Size 4096B Cache



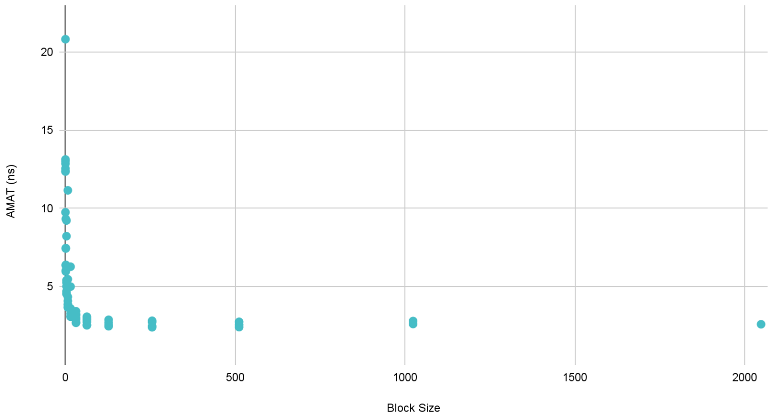
AMAT vs. Block Size 16384B Cache



AMAT vs. Block Size 262144B Cache



AMAT vs. Block Size 4194304B Cache

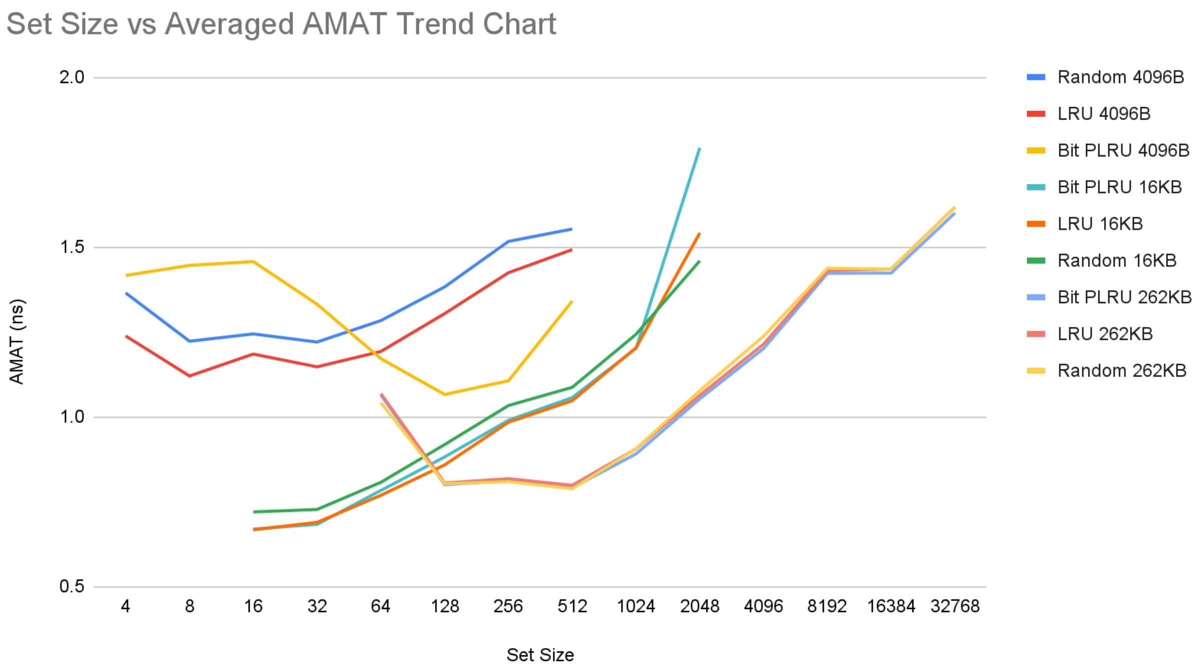


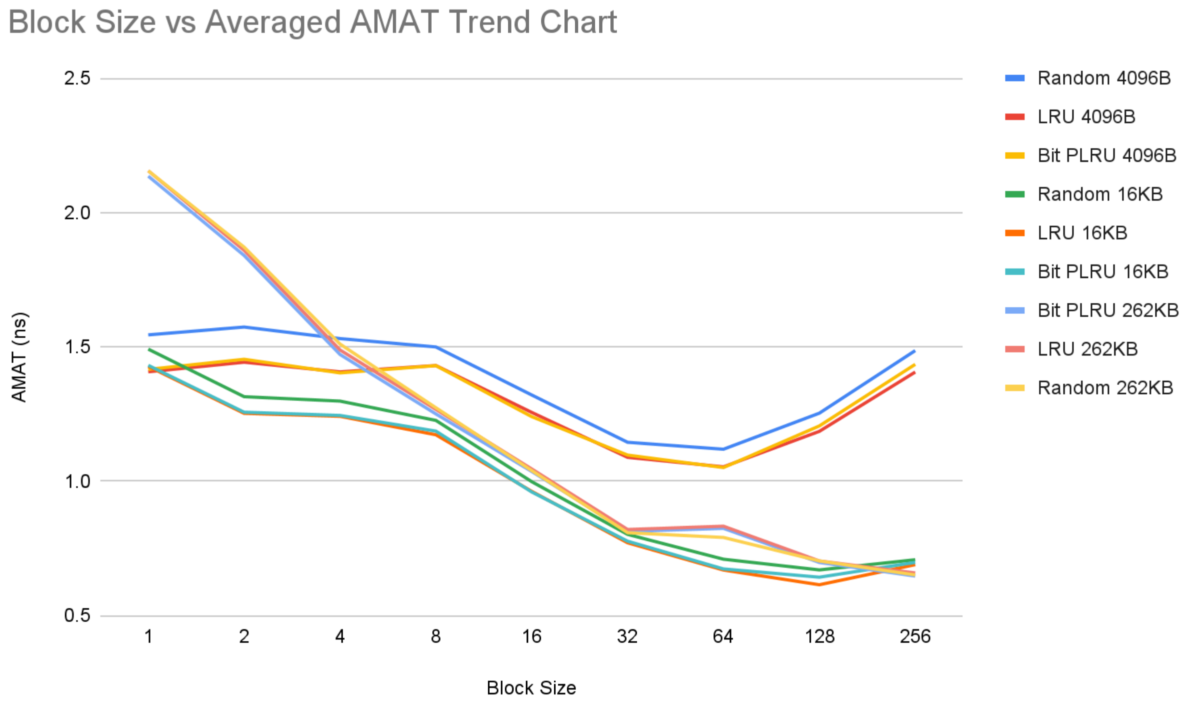
Task 2: Different Replacement Policies

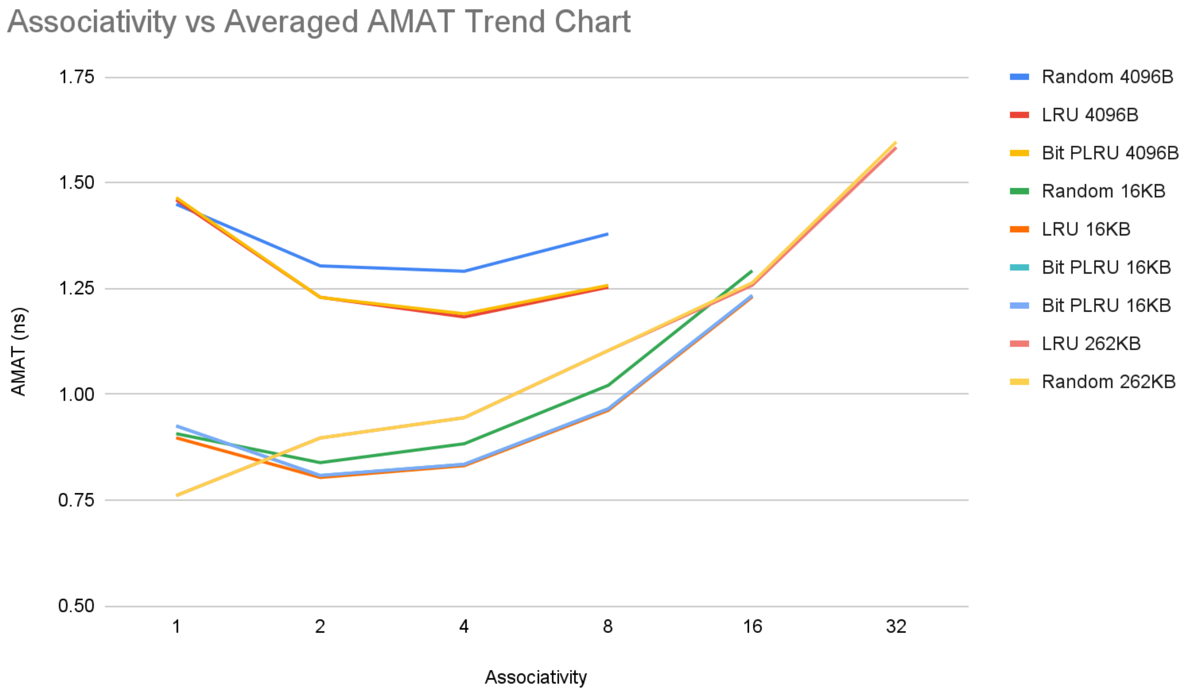
For this task, we have implemented three different replacement policies: random, LRU and bit-PLRU. We have chosen an associativity and set range based on our parameter sweep in the first task. To find these ranges, we chose clusters of points with minimal AMATs and decided on maximum and minimum viable associativities and sets to ensure that we are working from data which has reasonable access times with an eye towards minimizing AMAT generally. The values we have chosen are as the following:

Total cache size	**Associativity range**	**Set range**
4096	1-8 ways	4-512 sets
16384	1-16 ways	16-2048 sets
262144	1-32 ways	64-32768 sets

We obtained the simulation results of the different cache policies and computed their AMATs. Then we made nine separate graphs (clearly displayed in our task 2 data sheet). These graphs contained averaged AMAT values from different runs depending on associativity, block or set size again per cache size (three cache sizes times three cache parameters). From these nine graphs, we condensed all the different cache sizes into three graphs. These graphs, included below, show AMAT against the three cache parameters, associativity, block size and set size for different policies per cache size. From the graphs, we can see that LRU and bit-PRLU generally reduce the average access time compared to the random replacement bit policy. From our results, we see that the associativity of 4 or 8 tends to produce the better results.

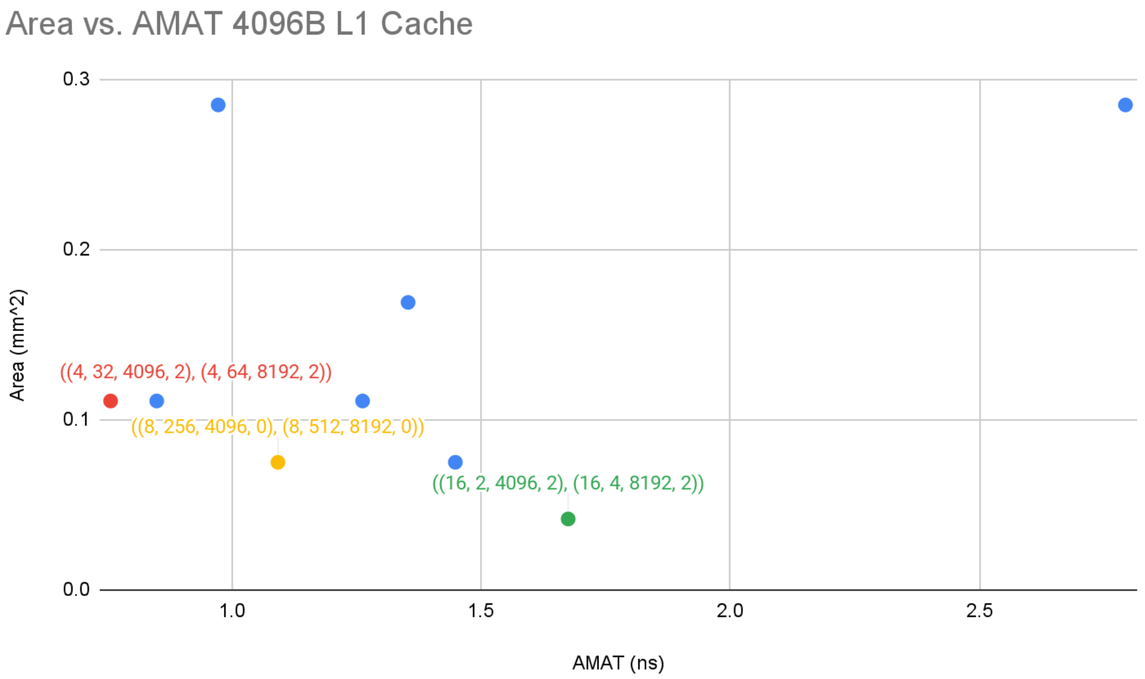

Set Size vs Averaged AMAT Trend Chart
AMAT (ns)
2.0
1.5
1.0
0.5
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
Set Size
Random 4096B
LRU 4096B
Bit PLRU 4096B
Bit PLRU 16KB
LRU 16KB
Random 16KB
Bit PLRU 262KB
LRU 262KB
Random 262KB

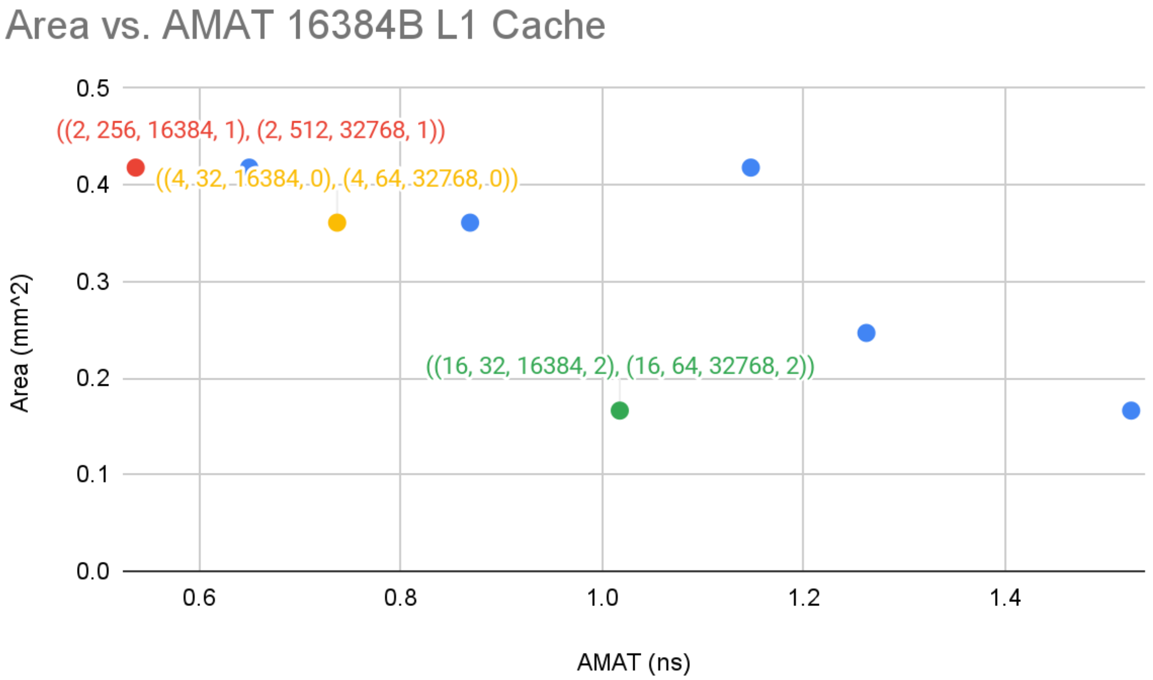

Block Size vs Averaged AMAT Trend Chart
AMAT (ns)
2.5
2.0
1.5
1.0
0.5
1
2
4
8
16
32
64
128
256
Block Size
Random 4096B
LRU 4096B
Bit PLRU 4096B
Random 16KB
LRU 16KB
Bit PLRU 16KB
Bit PLRU 262KB
LRU 262KB
Random 262KB

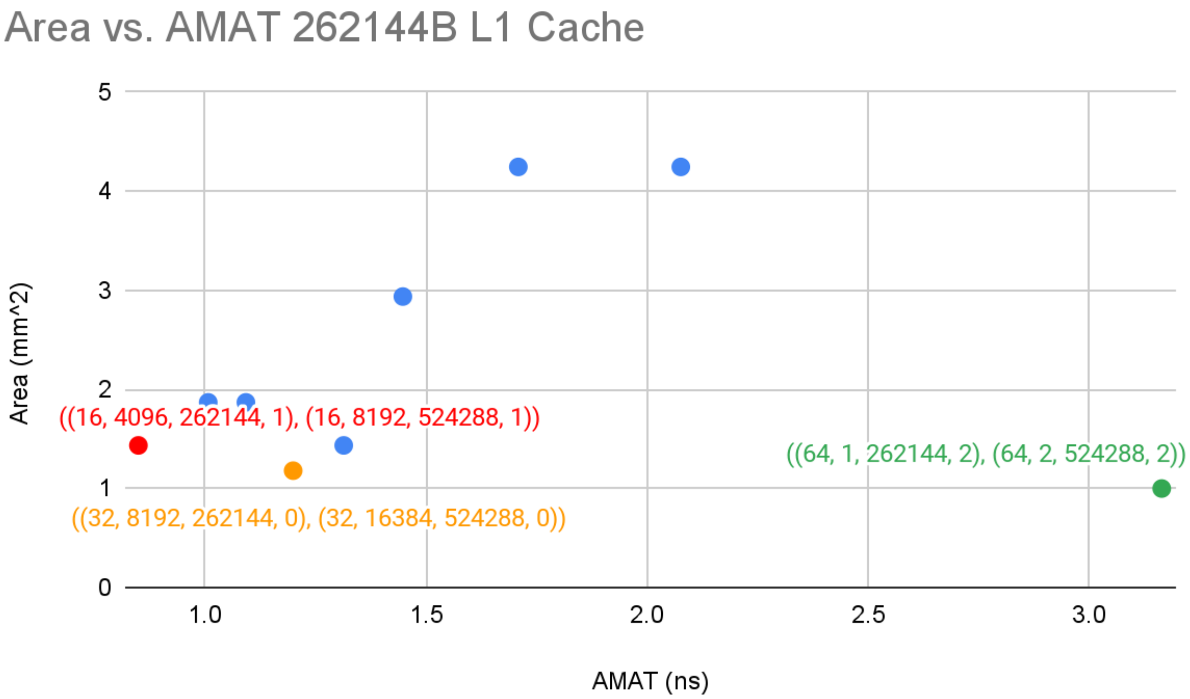


Task 3-4 : 2-Level Cache and Pareto Optimal

For this task, we have implemented a 2-level cache with specifications given in the following Cache Simulator Description section. We wrote an iteration tool to automatically find Pareto Optimal cache configurations. The tool works by first reading in the output of our run.py for our various simulated cache configurations (pinOut-<L1 cache size>.csv). Then the tool reads in a generated library of cache configuration specific statistics. This library, designated as destinyLog.csv is generated by a python script we wrote, runDestiny2.py, which calls Destiny and formats the output nicely. The tool iterates over all configurations of our simulated cache, finding corresponding Destiny values for each, computing AMAT for each, and outputting a spread of tuples to be graphed to find Pareto Optimal configurations. The tool sorts all the configurations in order of increasing AMAT, then returns a spread of those including the shortest AMAT and longest AMAT. The associated area is included with each tuple after AMAT. See our GitHub and the included readme for more information. We have included our Pareto Optimal graphs for our three cache sizes with optimal values highlighted and their tuples included in the formatting: ((L1 associativity, L1 block size, L1 size, L1 replacement), (L2 associativity, L2 block size, L2 size, L2 replacement)). For reference, our random policy is indicated as 0, LRU as 1 and bit PLRU as 2. Again, due to time constraints, we only simulated L2 caches with the same parameters as L1 cache and double the L1 cache size. We again see that a balance between block size and set size with a relatively small associativity tends to produce optimal results.


Area vs. AMAT 4096B L1 Cache
0.3
0.2
0.1
0.0
Area (mm^2)
((4, 32, 4096, 2), (4, 64, 8192, 2))
((8, 256, 4096, 0), (8, 512, 8192, 0))
((16, 2, 4096, 2), (16, 4, 8192, 2))
1.0
1.5
2.0
2.5
AMAT (ns)


Area vs. AMAT 16384B L1 Cache
0.5
0.4
0.3
0.2
0.1
0.0
Area (mm^2)
((2, 256, 16384, 1), (2, 512, 32768, 1))
((4, 32, 16384, 0), (4, 64, 32768, 0))
((16, 32, 16384, 2), (16, 64, 32768, 2))
0.6
0.8
1.0
1.2
1.4
AMAT (ns)


Area vs. AMAT 262144B L1 Cache
Area (mm^2)
AMAT (ns)
0
1
2
3
4
5
1.0
1.5
2.0
2.5
3.0
((16, 4096, 262144, 1), (16, 8192, 524288, 1))
((32, 8192, 262144, 0), (32, 16384, 524288, 0))
((64, 1, 262144, 2), (64, 2, 524288, 2))

Cache Simulator Description

Our cache simulator accepts the following parameters: L1 cache size, L1 associativity, L1 number of sets, L1 replacement policy, L2 cache size, L2 associativity, L2 number of sets and L2 replacement policy. First, our simulator uses the L1 parameters to split a passed 64 bit address of a load or a store into tag bits, set bits and offset bits which are used for indexing into our cache. Then using the set bits, the set of the load/store is traversed until a block with a matching tag is found. If the replacement policy is not random, the LRU/MRU indicator(s) are updated. A hit is recorded regardless of replacement policy. During this traversal, an index of a block for replacement is looked for in case there isn't a hit. In the LRU policy, the index of the block with the least recently used valid block is recorded, and the index of any empty block. In the bit-PLRU, the index of a block without its PLRU bit set is recorded and the total number of blocks with set PLRU bits is recorded. Once the traversal is completed and a load/store results in a miss, each replacement policy either fills a valid or an invalid block.

1.) If a valid block is being replaced, the L2 cache is first invoked to see if the block is already in the L2 cache[1]. If the block is already in L2, nothing is overwritten in the L1 cache and a miss is recorded (in code, a miss and an eviction are recorded, but this is fixed in data by subtracting recorded L2 hits from L1 evictions). If the block is valid and isn't in L2, a miss and an eviction is recorded.
2.) If an invalid block is being replaced, the block is overwritten and a miss is recorded.

In the random replacement policy, a random int from 0 to total number of ways - 1 is found, and the corresponding block is replaced according to the validity. If the LRU replacement policy is being used, if an empty block was found that is filled first. If no empty block, indexing is based on the least recently used block found earlier and replaced according to validity. The bit-PLRU policy is similar to the LRU policy. First, if all the PLRU bits are set, the first block in a set is replaced according to validity (and all PLRU bits are unset). Otherwise the recorded block with an unset PLRU bit is replaced according to validity.

[1] The L2 cache (and its replacement policies) functions in the exact same way as the L1 cache, except on replacement, if a valid block is being replaced, the block is simply overwritten in L2 and an L2 eviction is recorded rather than interacting with another layer of memory. The L2 cache also has its own independent hit, miss, eviction counts and parameters.