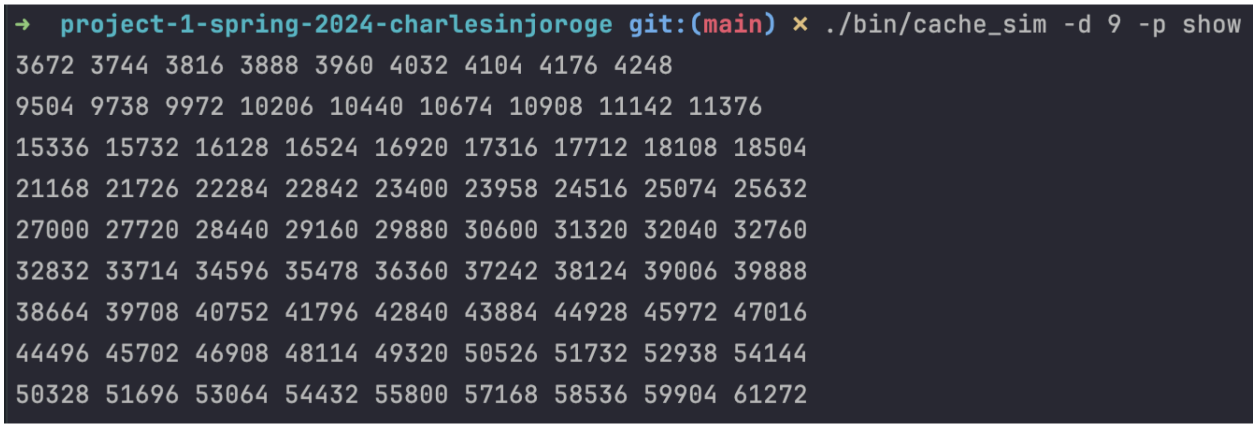
Cache Emulator

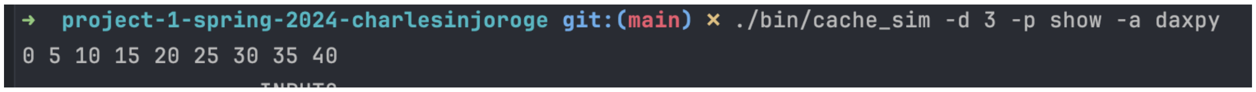
Correctness

Output of a 9 by 9 matrix

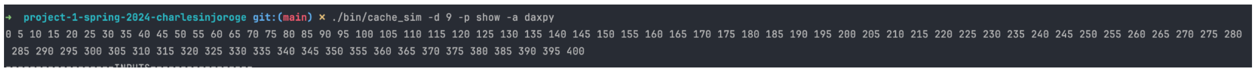


```
➜  project-1-spring-2024-charlesinjoroge git:(main) ✗ ./bin/cache_sim -d 9 -p show
3672 3744 3816 3888 3960 4032 4104 4176 4248
9504 9738 9972 10206 10440 10674 10908 11142 11376
15336 15732 16128 16524 16920 17316 17712 18108 18504
21168 21726 22284 22842 23400 23958 24516 25074 25632
27000 27720 28440 29160 29880 30600 31320 32040 32760
32832 33714 34596 35478 36360 37242 38124 39006 39888
38664 39708 40752 41796 42840 43884 44928 45972 47016
44496 45702 46908 48114 49320 50526 51732 52938 54144
50328 51696 53064 54432 55800 57168 58536 59904 61272
```

Output of daxpy. **Note:** the dimension parameter for daxpy is squared (just a design decision is all)



```
➜  project-1-spring-2024-charlesinjoroge git:(main) ✗ ./bin/cache_sim -d 3 -p show -a daxpy
0 5 10 15 20 25 30 35 40
```



```
➜  project-1-spring-2024-charlesinjoroge git:(main) ✗ ./bin/cache_sim -d 9 -p show -a daxpy
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175 180 185 190 195 200 205 210 215 220 225 230 235 240 245 250 255 260 265 270 275 280
 285 290 295 300 305 310 315 320 325 330 335 340 345 350 355 360 365 370 375 380 385 390 395 400
```

Associativity

Cache Associativity	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
1	446284800	211460889	11796711	5.28%	1413120	430080	23.33%
2	446284800	222252793	1004807	0.45%	1843200	0	0%
3	446284800	222658794	598806	0.27%	1843200	0	0%
4	446284800	222739200	518400	0.23%	1843200	0	0%
8	446284800	222739200	518400	0.23%	1843200	0	0%
16	446284800	222739200	518400	0.23%	1843200	0	0%
1024	446284800	222739200	518400	0.23%	1843200	0	0%

The choice to use 8-way set associativity is a reasonable choice because after 8 there is no real benefit other than the overhead of managing more sets. Depending on the implementation of LRU and other cache management techniques, it can significantly affect performance. Managing smaller sets is better.

Memory Block Size

Block Size	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
Init Included 8	470707200	207129600	27878400	11.86	13824000	691200	4.76
16	456883200	221068800	7027200	3.08	7257600	345600	4.54
32	449971200	222854400	1785600	0.79	3974400	172800	4.166
64	446745600	222051192	976008	0.43	2448000	86400	3.409
128	444902400	215407421	6698179	3.01	1569600	43200	2.67
256	443980800	209213667	12431133	5.6	731040	420960	36.541
512	443750400	109202640	112442160	50.73	680400	241200	26.17
1024	443750400	110012760	111632040	50.36	685800	235800	25.58
8	470246400	207129600	28108800	11.95%	13824000	0	0%
16	456422400	221184000	7142400	3.13%	6912000	0	0%
32	449510400	223027200	1843200	0.82%	3456000	0	0%
64	446284800	222252786	1004814	0.45%	1843200	0	0%
128	444441600	215623422	6712578	3.02%	921600	0	0%
256	443520000	209436868	12438332	5.61%	61440	399360	86.67%
512	443059200	109202640	112442160	50.73%	0	230400	100%
1024	443059200	110012760	111632040	50.37%	0	230400	100%

Workloads often exhibit what is known as spatial locality, where data elements that are close to each other are likely to be accessed around the same time. Larger block sizes exploit this by storing more contiguous data elements in a single cache block. Therefore, once a block is loaded into the cache due to a cache miss, subsequent accesses to nearby data (which is now in the cache) do not cause additional misses. By increasing the block size, each cache block can serve more potential data requests. This reduction in the miss rate is particularly noticeable in workloads where data access patterns are predictable or sequentially accessed, as is common in streaming data scenarios or large array traversals. Larger blocks mean that the overhead of fetching data from the next level of storage (whether that's a lower cache level or main memory) is amortized over more data. Each cache miss fetches more useful data, making the process more efficient in terms of time and energy per byte of data retrieved.

However, after a certain point, further increasing the block size can lead to an increase in cache misses, due to several factors. Larger blocks consume more cache space. This reduction in the number of blocks that can be stored in the cache at any given time can lead to an increase in capacity misses, especially if the working set size of the application is larger than the cache can effectively accommodate.If the block size exceeds the spatial locality footprint of the algorithm, much of the data brought into the cache may go unused. Replacing a large cache block is more costly than replacing a smaller one. If a cache miss occurs and the replacement policy dictates swapping out an existing large block, the penalty is higher due to the greater amount of data that needs to be written back to memory (if it is dirty) and the time taken to fetch a new large block. Especially in set-associative or direct-mapped caches, larger block sizes can exacerbate conflict misses. This occurs because there are fewer cache lines where data can be stored, increasing the likelihood of conflicts between different data items mapping to the same cache line.

Total Cache Size

Cache Size	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
4096	446976000	108550871	114706729	51.37	604800	1929600	76.13
8192	446976000	114528770	108728830	48.7	604800	1929600	76.136
16384	446976000	199878238	23379362	10.47	819840	1714560	67.65
32768	446976000	221105978	2151622	0.963	2225792	308608	12.17
65536	446976000	222252799	1004801	0.45	2448000	86400	3.4
131072	446976000	222495356	762244	0.341	2448000	86400	3.4
262144	446976000	222742887	514713	0.23	2448000	86400	3.4
524288	446976000	222847424	410176	0.18	2448000	86400	3.4
4096	446284800	108550873	114706727	51.38%	0	1843200	100%
8192	446284800	114528770	108728830	48.70%	0	1843200	100%
16384	446284800	199878277	23379323	10.47%	215040	1628160	88.33%
32768	446284800	221106025	2151575	0.96%	1620992	222208	12.06%
65536	446284800	222252778	1004822	0.45%	1843200	0	0%
131072	446284800	222495356	762244	0.34%	1843200	0	0%
262144	446284800	222742887	514713	0.23%	1843200	0	0%
524288	446284800	222847424	410176	0.18%	1843200	0	0%

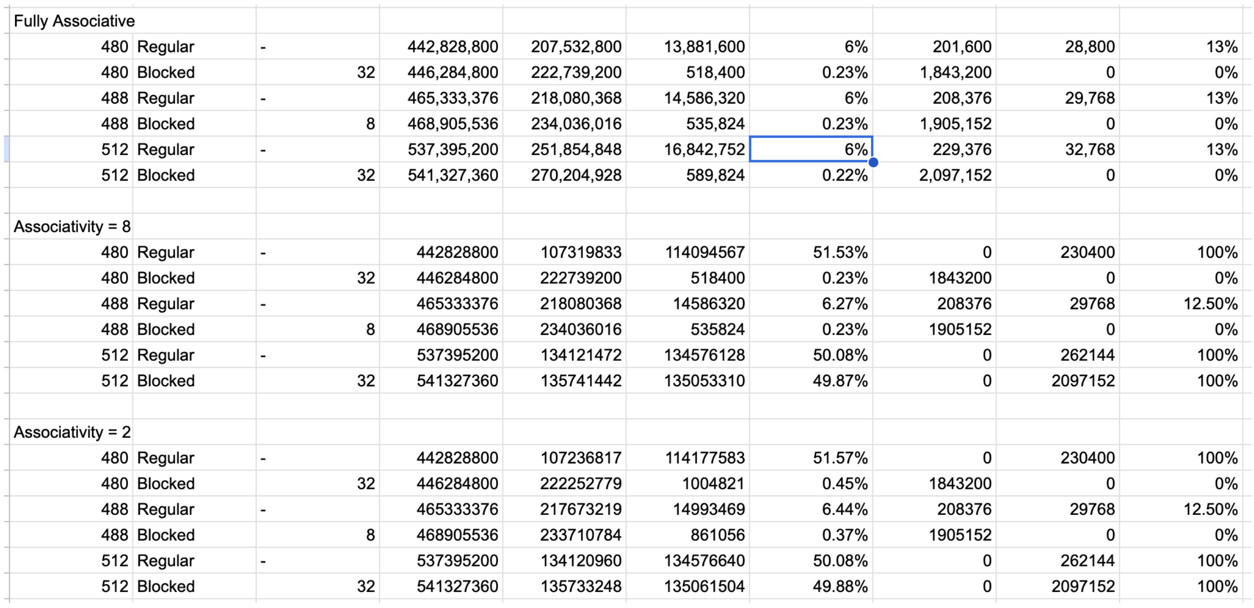
Given that the data I was able to collect indicates a requirement to reach a read miss rate of 0.5% or less, we can conclude that a cache size of 32768 bytes or more is needed. However, if the Skylake cache size of 32KB (32768 bytes) was insufficient to meet this target under some circumstances, there are a few architectural optimizations that could be considered:

- **Improve Spatial Locality:** Since cache effectiveness heavily depends on data locality, restructuring the data access pattern in the blocked matrix-matrix multiplication algorithm could help. Ensuring that the algorithm accesses contiguous memory locations to exploit the spatial locality could reduce miss rates.

- **Block Size:** Adjusting the block size within the matrix-matrix multiplication algorithm could also impact the cache hit rate, aligning the algorithm's block size with the cache line size to ensure full utilization of each cache line fetched.

- **Loop Tiling and Ordering:** Changing the loop ordering and tiling size within the matrix multiplication could also improve cache utilization by increasing temporal locality and reducing cache line eviction.

Problem Size and Thrashing



Fully Associative										
480	Regular	-		442,828,800	207,532,800	13,881,600	6%	201,600	28,800	13%
480	Blocked		32	446,284,800	222,739,200	518,400	0.23%	1,843,200	0	0%
488	Regular	-		465,333,376	218,080,368	14,586,320	6%	208,376	29,768	13%
488	Blocked		8	468,905,536	234,036,016	535,824	0.23%	1,905,152	0	0%
512	Regular	-		537,395,200	251,854,848	16,842,752	6%	229,376	32,768	13%
512	Blocked		32	541,327,360	270,204,928	589,824	0.22%	2,097,152	0	0%
Associativity = 8										
480	Regular	-		442828800	107319833	114094567	51.53%	0	230400	100%
480	Blocked		32	446284800	222739200	518400	0.23%	1843200	0	0%
488	Regular	-		465333376	218080368	14586320	6.27%	208376	29768	12.50%
488	Blocked		8	468905536	234036016	535824	0.23%	1905152	0	0%
512	Regular	-		537395200	134121472	134576128	50.08%	0	262144	100%
512	Blocked		32	541327360	135741442	135053310	49.87%	0	2097152	100%
Associativity = 2										
480	Regular	-		442828800	107236817	114177583	51.57%	0	230400	100%
480	Blocked		32	446284800	222252779	1004821	0.45%	1843200	0	0%
488	Regular	-		465333376	217673219	14993469	6.44%	208376	29768	12.50%
488	Blocked		8	468905536	233710784	861056	0.37%	1905152	0	0%
512	Regular	-		537395200	134120960	134576640	50.08%	0	262144	100%
512	Blocked		32	541327360	135733248	135061504	49.88%	0	2097152	100%

1. While observing the cache performance for the regular matrix-matrix multiply algorithm across different problem sizes, as detailed in Table 4, one might anticipate uniform performance given the minor variations in problem sizes. However, the data reveals significant discrepancies in cache performance, notably in read and write miss rates. These variations can primarily be attributed to the interaction between the problem size and the cache architecture, specifically the capacity and associativity. For instance, larger problem sizes lead to more frequent capacity misses as the data set exceeds the cache's capacity, causing data to be evicted and reloaded more often. Additionally, while increasing the cache's associativity typically reduces conflict misses and improves cache utilization by allowing more flexibility in data placement, it does not uniformly benefit all scenarios. For problem sizes close to or exceeding the cache capacity, even high associativity may not prevent the high rate of cache misses due to the sheer volume of data accesses and evictions, especially under a simple matrix multiplication algorithm that may not optimally exploit data locality.
2. For the 512 × 512 matrix problem size, the blocked matrix-matrix multiply algorithm, despite being designed to enhance cache efficiency, shows minimal improvement over the regular approach when analyzed based on a cache size of 65,536 bytes and a block size of 64 bytes, totaling 1,024 cache blocks for associativities 2 and 8. The read miss rates for regular and blocked methods are 50.08% and 49.87% respectively, indicating only a marginal reduction in read misses with the blocked approach. Both methods exhibit a 100% write miss rate, which underscores that neither approach effectively reduces write misses under the given cache configuration. This small difference suggests that the chosen block size and the total available cache blocks are not optimally utilized to fit the blocks of the matrix effectively within the cache lines or sets.. The data indicates that for the specific cache parameters and matrix size, further optimization of block size or a reevaluation of cache configuration may be necessary to

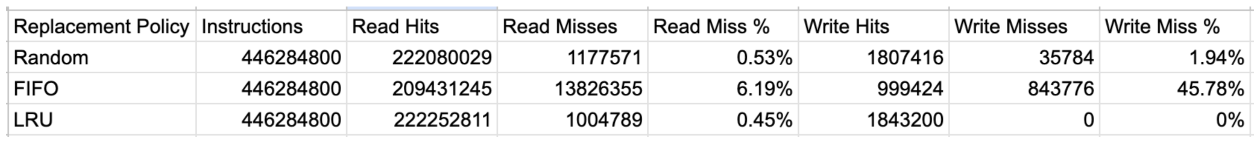
achieve the intended benefits of the blocked multiplication algorithm. Increasing the associativity to 1024 does show improvement for both.

3. Higher associativity in caches generally results in a substantial improvement in cache performance by reducing conflict misses, as observed in Tables 5 and 6. This improvement occurs because a higher or fully associative cache can store data in any of its lines, thereby drastically decreasing the chances that incoming data will evict other crucial data prematurely. However, while fully associative caches maximize the flexibility in data storage and can significantly enhance cache hit rates, they are not commonly used in most hardware designs due to several drawbacks. Firstly, fully associative caches are slower because every cache line must be checked for a hit, which increases the complexity and time of the cache lookup process. Additionally, the hardware required for fully associative caches is more complex and expensive, as it requires more sophisticated control logic and tag comparison circuits. This complexity also leads to higher power consumption and increased physical size, making fully associative caches less practical for most applications, especially where power efficiency and cost are critical considerations.
4. **Solutions**
    a. **Solution: Use Blocking/Tiling:** Divide matrices into smaller sub-matrices or blocks that fit well within the cache. This technique reduces the number of cache misses by ensuring that once data is loaded into the cache, it is utilized fully before being evicted.
    - **Implementation Strategy**: Choose a block size such that the entire block fits into the cache. Given a cache size of 65,536 bytes and a block size of 64 bytes, you can store up to 1,024 blocks in the cache. Since the cache is 8-way set associative, organizing data in blocks that match or are multiples of 8 could align with the cache's natural indexing to minimize conflict misses.
    - If each element of the matrix is a double (8 bytes), then each block can ideally be `(65,536 bytes / 8 bytes/element) / 8 (for associativity) = 1,024 elements per set`. A square root of 1,024 gives approximately 32, so blocks of 32x32 elements are a good choice. Each block is then `(32 elements x 32 elements x 8 bytes = 8,192 bytes)`, small enough to fit into one cache set without displacement.

    b. Solution 2:
        i. Access matrix elements in an order that follows their storage in memory (row-major or column-major), depending on the programming language, to improve spatial locality

    c. Solution 3: Insert prefetch instructions ahead of the primary use in the code. This is particularly beneficial when predictable patterns (like traversing matrix rows or columns) are evident.

Replacement Policy



Replacement Policy	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
Random	446284800	222080029	1177571	0.53%	1807416	35784	1.94%
FIFO	446284800	209431245	13826355	6.19%	999424	843776	45.78%
LRU	446284800	222252811	1004789	0.45%	1843200	0	0%

LRU was more efficient for this workload, however this may not be the case for all workloads. FIFO is not performant at all in comparison because it doesn't take into consideration the access structure of the data. Random may get lucky with its choices.