Texas A&M University
Computer Science 434
Compiler Design

Assignment Seven

David Cox

December 9, 2015

Part One

Deliverables

1. Create at least three test runs for evaluation. Document what you did for each test run and why you chose to perform the test run in that particular way. These test runs should perform sufficient work to allow you to differentiate between cache settings.

2. Provide a theoretical model of how each execution should perform for Direct Mapped, Fully Associative, and N-Way Associative placement schemes, and MRU, LRU, and Random replacement schemes.

3. Test your models, providing results that prove your model. Explain how your model is correct and explain any exceptions or inconsistencies.

Test Data

Both spatial and temporal locality play a large role in cache performance. To adequately test the various placement and replacement schemes, it is necessary to generate test runs that exhibit variations in these features.

Low spatial locality was simulated by running `sort.decaf` on a reverse-ordered, 100 element list. By using an input array that is large (relative to our cache size), we ensure that the entire array cannot be cached at once. This causes a significant number of cache misses on each iteration of the sorting function.

Reverse-ordered 50 and 20 element lists were used to simulate a moderate and high degree of spatial locality, respectively.

Theoretical Performance Models

In direct mapped cache, each memory address maps directly to a single slot in the cache, with multiple addresses mapping to the same cache slot. Performance issues arise when data gets repeatedly loaded and evicted to the same slot. This can happen even when the cache is relatively empty because slots are assigned with the use of a hashing function rather than availability. Associative cache addresses this issue by mapping memory addresses to a set of slots. When moving data into an associative cache, hardware logic determines which slot within a set to place the data. The added hardware logic increases the physical size and complexity of the cache, making large fully associative caches impractical.

N-Way associative caches represent the spectrum of caches between direct mapping and full associativity. They retain a relatively small silicon footprint while gaining some degree of flexibility in where data is stored.

From this, we can infer that the simulations of fully associative cache will fair much better than direct mapping, with N-way associativity falling somewhere in the middle.

The Most Recently Used (MRU) replacement scheme evicts the most recently used cache line upon sufficiently high memory pressure. This is in contrast to the Least Recently Used (LRU) replacement scheme, which evicts the least recently used. MRU is ideally used when a high turnover of cache data is desired (low temporal locality), such as when infrequently accessing many items an equal number of times for each. LRU results in frequently used data residing in the cache longer, and making it suited for situations where many items are being accessed at different frequencies relative to each other. Random replacement gives no bias to last access time and serves as a middle ground between MRU and LRU.

The selection sort algorithm in `sort.decaf` exhibits spatial locality with regards to how it traverses the input list. Temporal locality starts low and increases with each iteration – the last few elements are accessed in quick succession. Due to the variation in element access frequency, LRU is likely the optimal replacement scheme, followed by random replacement.

Performance Evaluation Setup

A total of 72 tests were performed in attempt to verify our theoretical models. The provided cache simulator does not allow for input from `stdin`, which initially limited the number of tests that could be performed, as manual input to the program is very tedious and error prone. To solve this issue, I used keyboard macros. An initial recording was done with the utility `xmacrorec2`, which served as a template for scripting use later. Unfortunately execution of the macros could not be scripted, as there was no way to ensure completion of the cache simulator. Even using the `sleep` command, this still did not work. All 72 tests had to be started manually.

This semi-automated testing method allowed all possible combinations of replacement and placement schemes to be tested for both four and eight block cache sizes, resulting in a more complete picture than the alternative of several, hand-picked tests. The cache length (number of block) for both tests was kept artificially small to stress the replacement and placement schemes. Comparing two different sized caches helps account for any variations in hit rate due to cache size.

Results

The full test results are shown in the table below and represented graphically in the appendix section. From this data, two trends are majorly apparent. The MRU replacement algorithm performs far worse than LRU and Random removal, unless the Direct Mapped placement scheme is being used, in which case all three replacement algorithms perform exactly the same. It appears that performance of the Direct Mapped placement scheme is hindered most by the nature of the scheme itself, rather than the replacement algorithm being used. This is further verified by the significant performance difference between direct mapping and 2-way associative mapping – providing one degree of "flexibility" in data placement increases the hit rate by approximately 20% on average.

It's worth noting here that a small increase in cache hit rate results in a tremendous performance improvement, as each cache miss causes a delay of hundreds of clock cycles.

		Direct Mapped	Fully Associative	2-Way
Sort[100..1] Cache Accesses: 148485	MRU	0.4139	0.1300	0.1509
	LRU	0.4139	0.5595	0.5150
	Random	0.4139	0.4797	0.4846
Sort[50..1] Cache Accesses: 39335	MRU	0.4256	0.1522	0.1589
	LRU	0.4256	0.5885	0.5352
	Random	0.4256	0.5094	0.5029
Sort[20..1] Cache Accesses: 7445	MRU	0.4484	0.3032	0.1778
	LRU	0.4484	0.6770	0.6700
	Random	0.4484	0.5852	0.5472
Sort[2..1] Cache Accesses: 407	MRU	0.7371	0.6118	0.3563
	LRU	0.7371	0.8403	0.8329
	Random	0.7371	0.8059	0.7568

Hit Ratios for a 4 Block Cache

		Direct Mapped	Fully Associative	2-Way
Sort[100..1] Cache Accesses: 2296	MRU	0.6730	0.1768	0.1217
	LRU	0.6730	0.8604	0.7626
	Random	0.6730	0.8714	0.7129
Sort[50..1] Cache Accesses: 282	MRU	0.6612	0.2110	0.1361
	LRU	0.6612	0.8785	0.7767
	Random	0.6612	0.8816	0.7412
Sort[20..1] Cache Accesses: 7445	MRU	0.6203	0.3588	0.2411
	LRU	0.6203	0.9268	0.8379
	Random	0.6203	0.9076	0.7449
Sort[2..1] Cache Accesses: 282	MRU	0.7961	0.6634	0.4717
	LRU	0.7961	0.9410	0.9017
	Random	0.7961	0.8993	0.8206

Hit Ratios for an 8 Block Cache

Conclusion

Our theoretical models lined up fairly well with the results of the testing. It was surprising to see the Direct Mapped placement algorithm produce exactly the same result for each replacement algorithm (discussed above), however upon consideration it makes sense. The difference in hit rates between 2-way and fully associative schemes was slightly less than expected, and is likely not enough to justify the much greater costs and size limitations of fully associative memory in anything other than a very small cache.

The most interesting and perhaps "useful" observation (if you're designing hardware and we're assuming the simulator is accurate, which it likely isn't), is that direct mapped

placement with 8 blocks consistently outperforms both the 2-way and fully associative schemes. This would be an important discovery for a hardware manufacturer – they might favor using larger, direct mapped caches over smaller, more complex (expensive) n-way associative caches.

Part Two

Deliverables

1. You will generate at least four test runs for evaluation. You may create more test runs to provide more information as needed. The four minimum test runs should illustrate a small grid size with a low generational number, small grid with many generations, a large grid with few generations, and a large grid with many generations. Examination of the decaf code should assist in determining appropriate values.

2. Find the single optimal cache configuration that provides the best average performance. Explain and justify your definition of optimal, then show that this configuration fulfills this requirement better than any other configuration. Note and explain any deviations or inconsistencies.

Test Data

The test data for this section was generated by running `life.decaf` with input matrices of size 5x5 and 15x15 for 1 and 30 iterations, each. The Game of Life involves checking the state of a given cell's neighboring cells to determine if the cell dies. This operation exhibits good spatial locality on small grids, as cells in rows below the current cell are more likely to reside in the same page in memory. Increasing the grid size reduces the spatial locality. Temporal locality is also better for small matrices, as there are less cells that must be evaluated each round.

Ideal Result

A good cache has a high average hit rate, a reasonable size, and is likely not fully associative. As previously stated, fully associative caches are impractical due to the physical space required for the logic that performs tag comparisons. Cache lines must also be of a reasonable size. Doubling the length of a cache line slightly decreases the odds of a cache miss, but twice as much data must be loaded each miss. Similarly, shorter cache lines require less data to be loaded in the event of a miss, but theoretically will result in more frequent misses.

Performance Evaluation

The first four tests were used as a baseline for future work. The cache simulator's default settings were used to evaluate the cache performance of each execution trace described

above. Manual, iterative tests were then done using the trace with the lowest hit ratio. This trace represents the worst case performance that will be encountered by our cache (in this series of tests). Iterative improvements upon the worst case performance are likely to have a beneficial, or neutral effect on the best case. Testing all four cases each iteration would be too time consuming for the constraints of this assignment. Each iteration consisted of a performance evaluation and a single modification, favoring configurations that were "economical" both well performing. The final result of this was saved for later comparison.

Using the keyboard macros from earlier, I was able to automate a fairly comprehensive set of tests for all combinations of cache sizes from 2 – 64 blocks and cache lines of size 1 – 64 words. All tests used the LRU replacement scheme and 2-way associative mapping. This amounted to 42 automated tests, and 23 manual (iterative) tests.

Results

Manual optimization resulted in a configuration of 128 blocks with 4 word cache lines. Using short cache lines reduces the amount of data transferred upon a cache miss. Performance of this configuration is compared against the benchmark and shown below.

	Benchmark	Optimized Configuration
	Cache Hit Rate	Cache Hit Rate
5x5: 1 iteration	0.9266	0.9807
5x5: 30 iterations	0.9228	0.9821
15x15: 1 iteration	0.8938	0.9524
15x15: 30 iterations	0.8904	0.9517

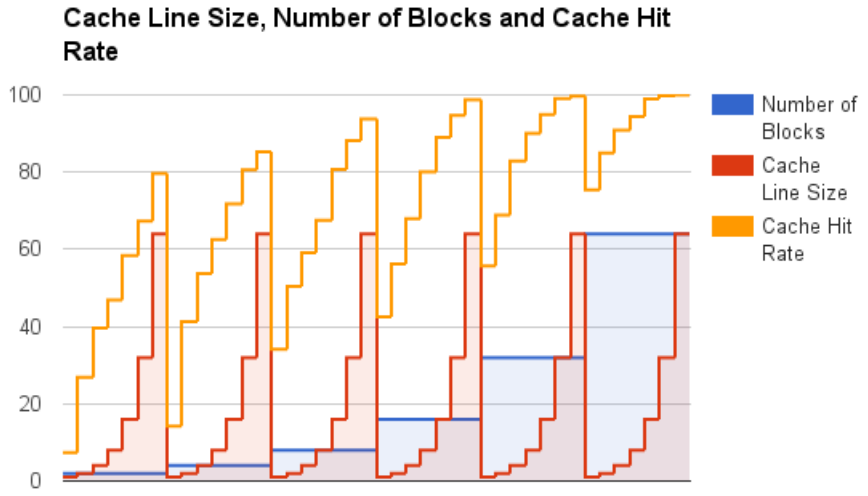
Benchmark vs Optimized Configuration

As in the previous section, being able to automate some of the testing was tremendously useful. Running 42 tests by hand would have been very error prone and tedious. From the table on the next page, we see that our configuration is fairly close to ideal. The appendix contains several graphical plots of this data.

Number of Blocks	2	4	8	16	32	64
Cache Line Size						
1	0.74	0.1421	0.3414	0.4252	0.5569	0.754
2	0.2685	0.4126	0.5037	0.5625	0.6891	0.8494
4	0.3965	0.5371	0.5911	0.6798	0.8285	0.9086
8	0.4693	0.626	0.6754	0.8013	0.9005	0.9445
16	0.5844	0.7182	0.807	0.8904	0.9489	0.9901
32	0.673	0.806	0.8817	0.9471	0.9905	0.9975
64	0.7962	0.8525	0.9375	0.9879	0.9967	1.000

Hit rate for various cache configurations – 15x15: 30 iterations

The following chart is my best effort at trying to display this data. It's a bit "busy" but does a nice job of illustrating the direct correlation between cache size and hit rate (up until the last couple points when we see diminishing returns).



The appendix has two different views of a three dimensional plot for the data as well. I found it helpful to first look at these, then look back at the table above. The gradient in cache hit rate becomes much more apparent.

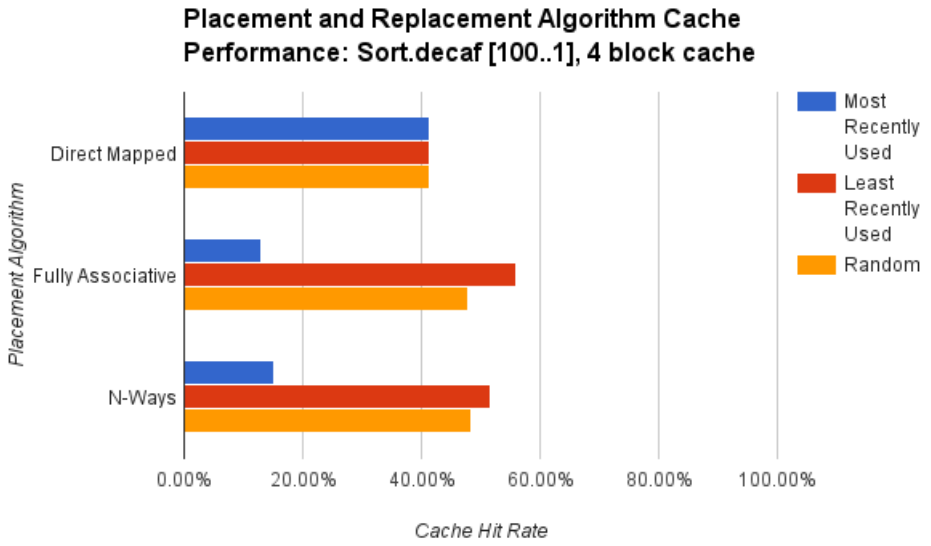
Conclusion

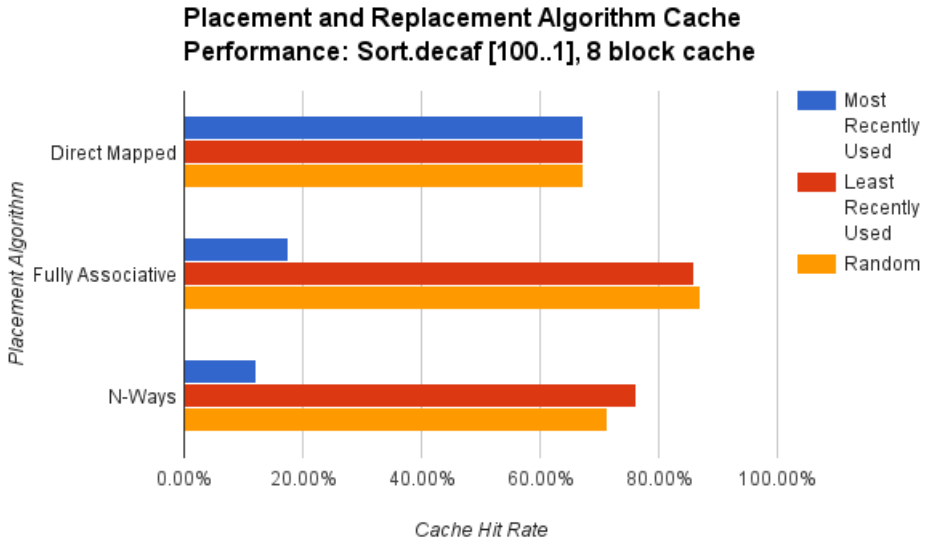
The task of finding "the single optimal cache configuration" is deceptively open ended, as the definition of optimal changes both with the hardware limitations of the cache as well as the intended usage situation. For example, a TLB makes use of fully associative memory, but can only do so due to its small size, making this sort of design impractical for other cache devices.

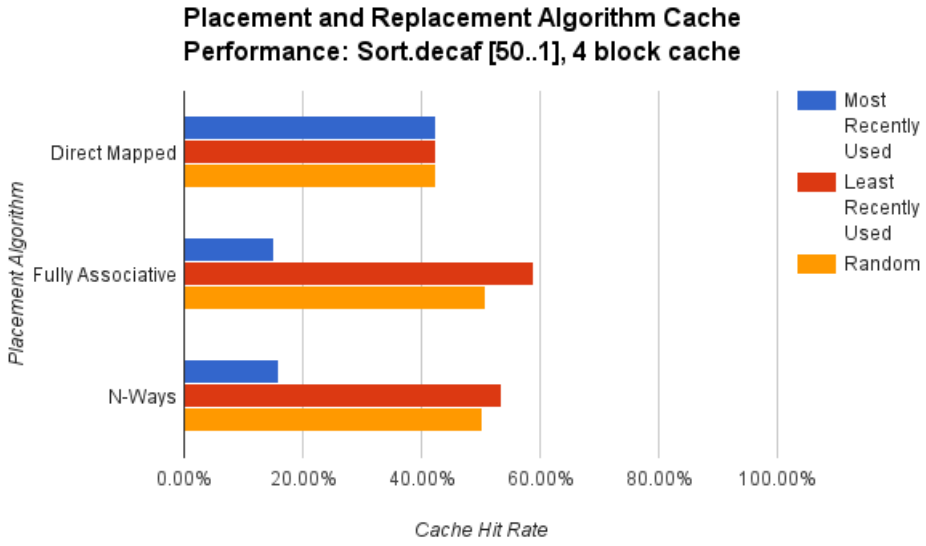
The final cache design resulted in a very close to ideal hit ratio – the only likely design changes might be for economic reasons (too many blocks used, too large for 2-way associative, etc.). If I were to extend upon this project in the future, I think it would be interesting to extend the automated testing to support an algorithmic approach to cache design, as this sort of problem seems like it would be well suited to genetic programming.

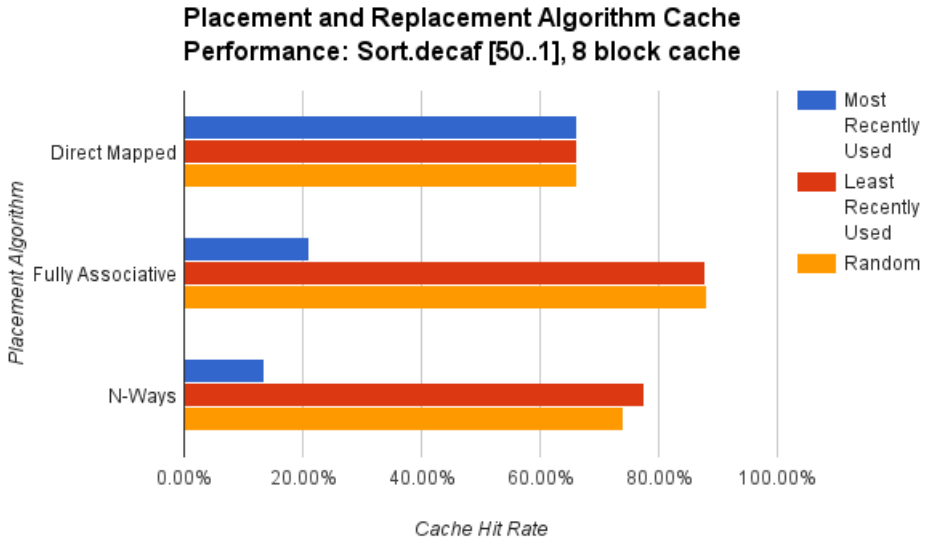
Appendix

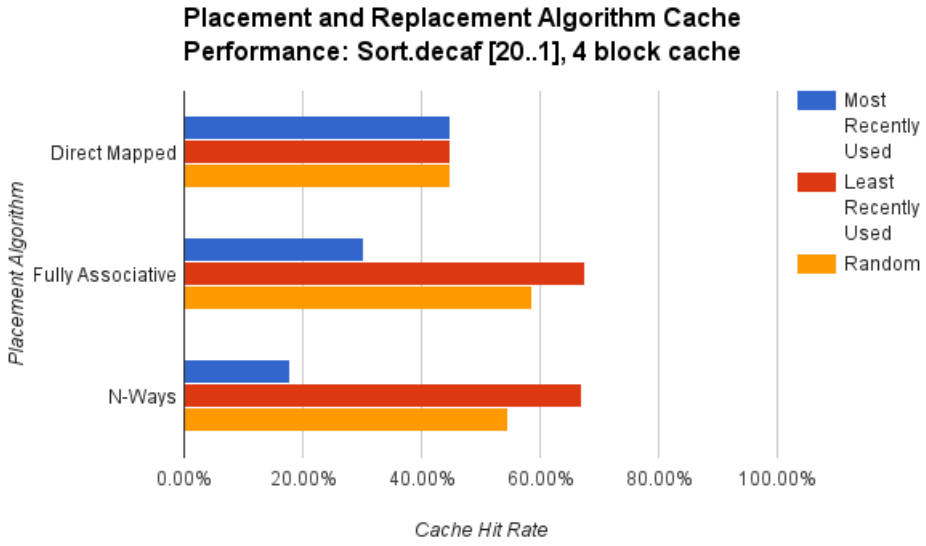
Part One Figures

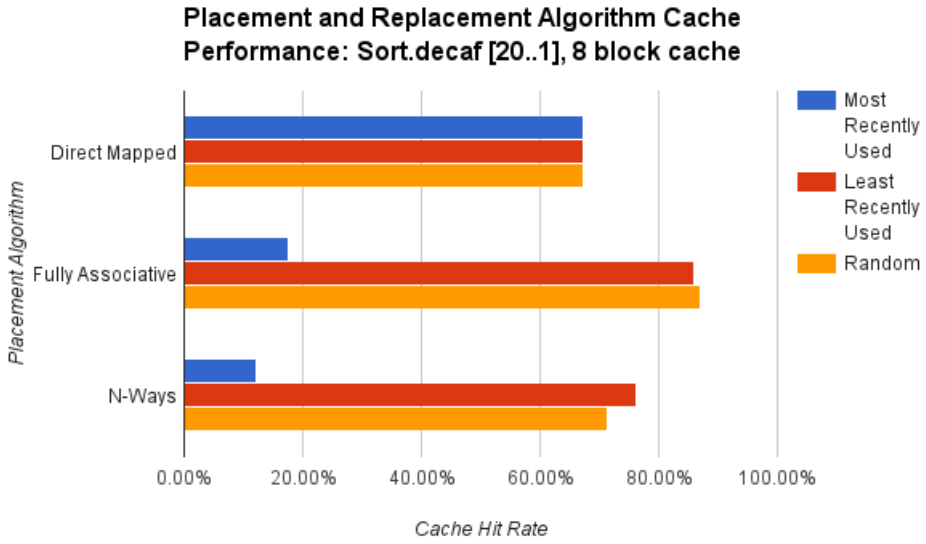


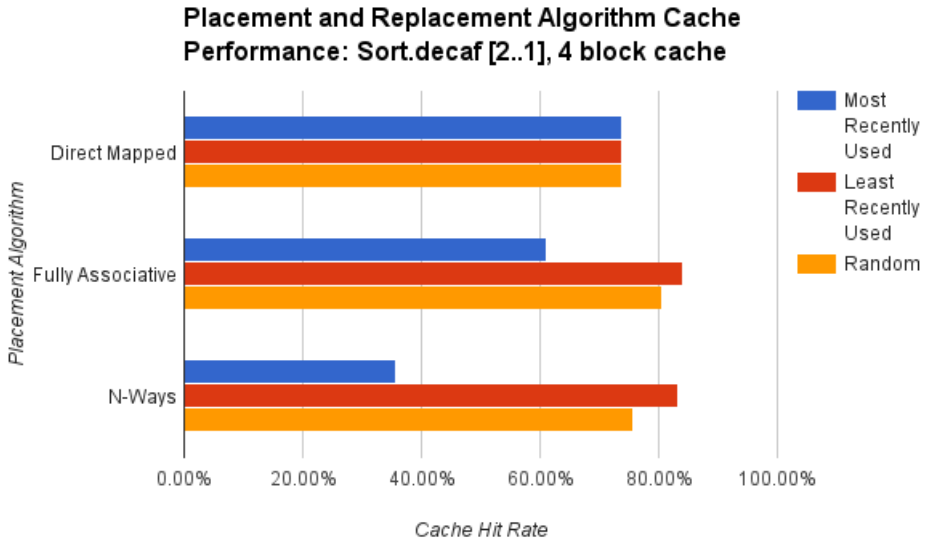


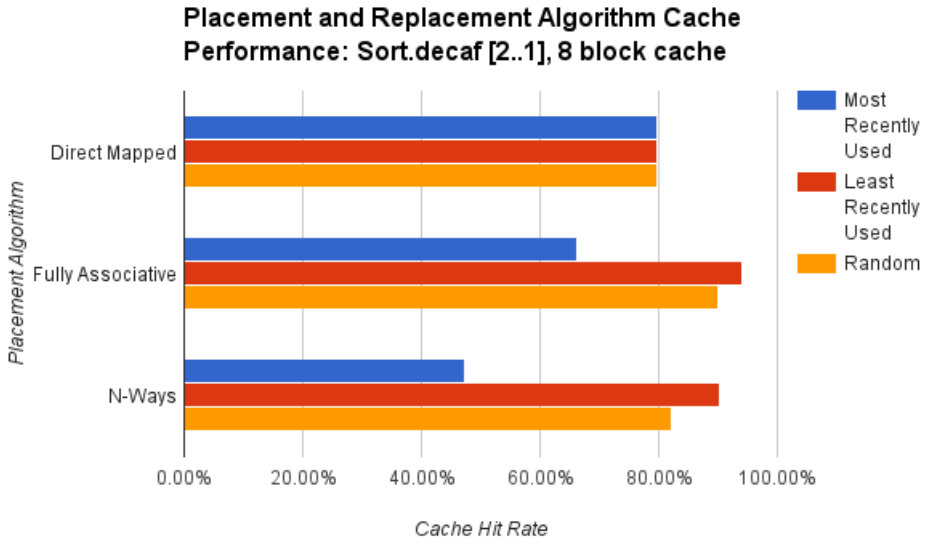

Placement and Replacement Algorithm Cache Performance: Sort.decaf [50..1], 4 block cache
Direct Mapped
Fully Associative
N-Ways
Placement Algorithm
0.00%
20.00%
40.00%
60.00%
80.00%
100.00%
Cache Hit Rate
Most Recently Used
Least Recently Used
Random


Placement and Replacement Algorithm Cache Performance: Sort.decaf [50..1], 8 block cache
Direct Mapped
Fully Associative
N-Ways
Placement Algorithm
0.00%
20.00%
40.00%
60.00%
80.00%
100.00%
Cache Hit Rate
Most Recently Used
Least Recently Used
Random

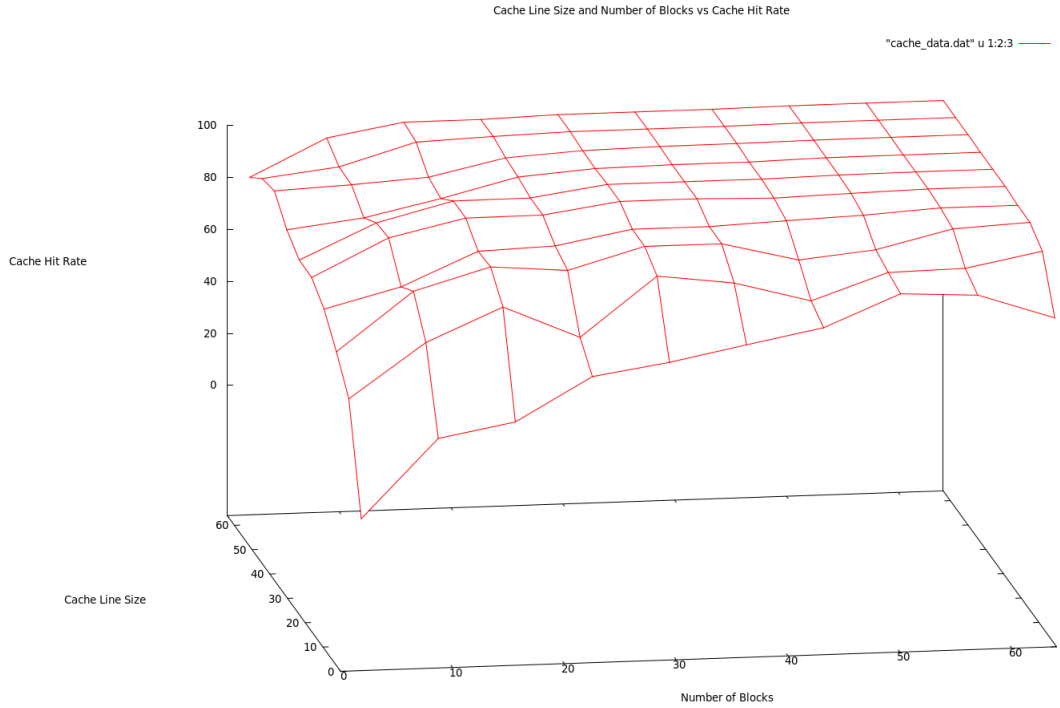

Placement and Replacement Algorithm Cache Performance: Sort.decaf [20..1], 4 block cache
Direct Mapped
Fully Associative
N-Ways
Placement Algorithm
0.00%
20.00%
40.00%
60.00%
80.00%
100.00%
Cache Hit Rate
Most Recently Used
Least Recently Used
Random

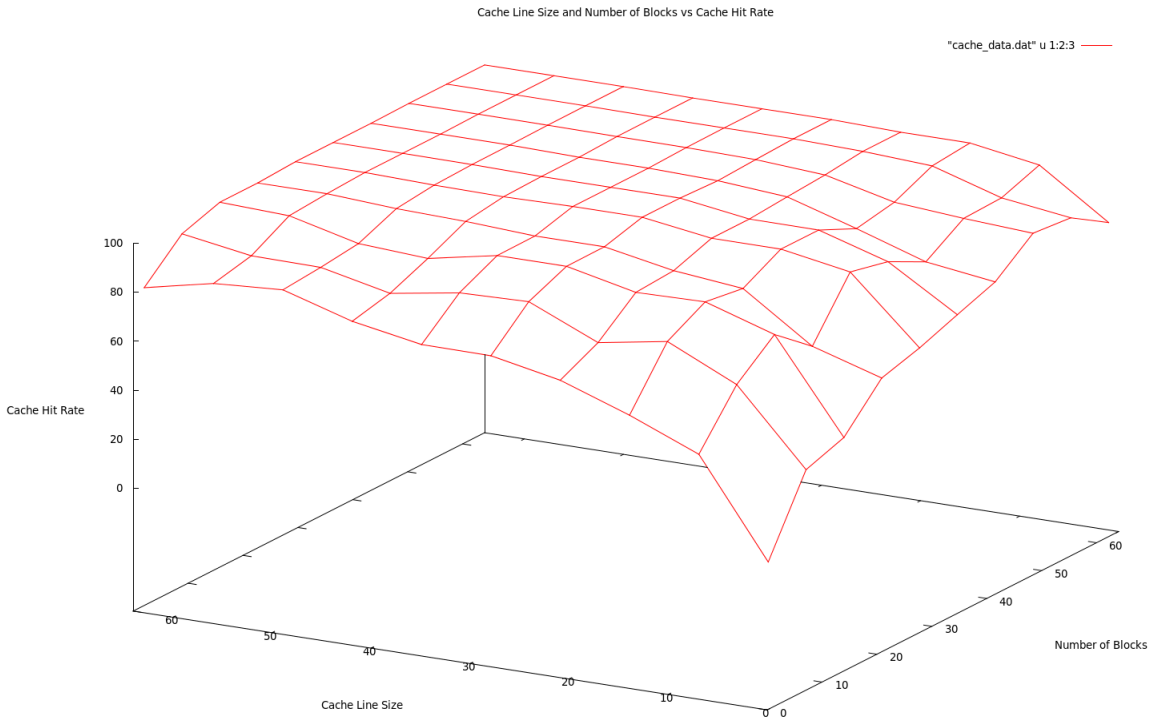

Placement and Replacement Algorithm Cache Performance: Sort.decaf [20..1], 8 block cache
Direct Mapped
Fully Associative
N-Ways
Placement Algorithm
0.00%
20.00%
40.00%
60.00%
80.00%
100.00%
Cache Hit Rate
Most Recently Used
Least Recently Used
Random


Placement and Replacement Algorithm Cache Performance: Sort.decaf [2..1], 4 block cache
Direct Mapped
Fully Associative
N-Ways
Placement Algorithm
0.00%
20.00%
40.00%
60.00%
80.00%
100.00%
Cache Hit Rate
Most Recently Used
Least Recently Used
Random


Placement and Replacement Algorithm Cache Performance: Sort.decaf [2..1], 8 block cache
Direct Mapped
Fully Associative
N-Ways
Placement Algorithm
0.00%
20.00%
40.00%
60.00%
80.00%
100.00%
Cache Hit Rate
Most Recently Used
Least Recently Used
Random

Part Two Figures





References

"Set Associative Cache." Set Associative Cache. UMD, n.d. Web. 09 Dec. 2015.