Bresia Prudente, bprude2

Lab Report

For this experiment, different combinations of parameters for the cache simulator were run. By doing so, this helps test various set associative and direct mapped caches.
The simulator provides a graphing utility that's enabled by using ssh –Y which helps visualize how caching works. The data tables were the automatically generated ones from Microsoft Word, while the graphs were created using bash scripts and gnuplot.

Histogram:

Associativity	Block Size	Cache Size	Miss Rate(%)	Hit Time	Miss Penalty	Total Time
128	256	128	6.6	264	2660	638160
64	256	128	6.6	136	2660	452560
64	512	128	6.6	136	5220	693100

Based on the tables above, it shows the smallest miss rates out of three trials. It can be assumed that the miss penalty increases as the block size increases. This could be because of the time it takes to fetch data from a slower drive.

Block Size vs Miss Penalty:

```
12000 ++-----------+------------+------------+------------+------------+++
+ + + + + ****** |
| *******
10000 ++ ********* ++
| ********* |
| ********** |
8000 ++ ********** ++
| ********** |
6000 ++ *********** ++
| *********** |
| ************ |
4000 ++ *********** ++
| *********** |
| ************ |
2000 ++ ************ ++
| ********* |
+******* + + + + + |
0 ***----------+------------+------------+------------+------------+++
0 200 400 600 800 1000
```

The graph shows a linear correlation between the miss penalty and block size. This shows a linear correlation between the miss penalty and the block size; the greater the miss penalty, the greater the block size.

Associativity vs Hit Penalty

```
2500 ++---------+-----------+----------+----------+-----------+---------++
 + ****** +
 | |
 2000 ++ ******* ++
 | ********* * |
 | ********* * |
 | ********** * |
 1500 ++ **** ***** * ++
 | **** ***** * |
 | **** ***** * |
 1000 ++ *********** **************** ++
 | ***** ****** ********************* |
 | ***** ************************** |
 | **************************** |
 500 ++ ************************ ++
 | ******************** |
 +*************** + + + + +
 0 ********---+-----------+----------+----------+-----------+---------++
 0 200 400 600 800 1000 1200
```

This graph shows that increasing the associativity of the graph increases hit time as well. The block must be scanned sequentially to find the correct match, hence the increase of the associativity. The empty spaces within the graph is assumed to be parameters which produce an invalid cache.

Sort

```
#include "inst_legible.h"
#include <iostream>
using namespace std;

const int listSize = 10000;
int l[listSize];
void XSort(int list[], int n);
```

```
void QSort(int list[], int lo, int hi);
int Partition(int list[], int lo, int hi);

int main()
{
	int i, tmp, i1, i2;

	// Generate a list with no repeated numbers
	for (i=0; i<listSize; i++) l[i] = i;

	// Scramble the numbers thoroughly
	for (i=0; i<listSize; i++) {
		i1 = rand() % listSize; i2 = rand() % listSize;
		tmp = l[i1]; l[i1] = l[i2]; l[i2] = tmp;
	}

	QSort(l, 0, listSize-1);
	return 0;
}//end main

void XSort(int list[], int n)
{
	// Exchange sort
	int min, tmp, i, j, min_j;

	// Scan the list from the left to the right
	for (i=0; i<n-1; i++) {
		// Remember the item at position i
		INST_R(list[i]);
		min = list[i]; min_j = i;

		//Check the list to the right of position i for any smaller items
		for (j=i+1; j<n; j++) {
			INST_R(list[j]);
			if (list[j] < min) {

			// Find where the smaller item is and remember it
			INST_R(list[j]);
			min = list[j]; min_j = j;
			}
		}
		// Swap the item at position i with the smallest item found to the right
		INST_R(list[i]);
		INST_R(list[min_j]);
		INST_W(list[i]);
		tmp = list[i]; list[i] = list[min_j];
```

```
            INST_W(list[min_j]);
            list[min_j] = tmp;
        }
} //end void XSort

//Assumes no repeated items on the list
void QSort(int list[], int lo, int hi)
{
        int k;
        if (lo < hi) {

            // Partition the list into two sub-lists
            k = Partition(list, lo, hi);

            // Now every item left of position k is smaller than the item at k,
            // while every item right of position k is larger than the item at k
            QSort(list, lo, k-1); // sort the sublist to the left of k
            QSort(list, k+1, hi); // sort the sublist to the right of k
            }
}//end void

//Partitions the function for quicksort
int Partition(int list[], int lo, int hi)
{
        int x, tmp;
        // Pick an arbitrary key, say half way through the list
        INST_R(list[(lo+hi)/2]);
        x = list[(lo+hi)/2];

        // Now swap items until every item to the left of the key is smaller than
        // the key, and every item to the right of the key is larger than the key
        while (lo < hi) {
            // Scan from the right until we find an item smaller than the key
            while ( (lo < hi) && (x < list[hi]) ){
                INST_R(list[hi]);
                hi--;
            }
            // Scan from the left until we find an item larger than the key
            while ( (lo < hi) && (x > list[lo]) ){
                INST_R(list[lo]);
                lo++;
            }

            // Swap the two items we've discovered on the wrong side of the key
            INST_R(list[hi]);
            tmp = list[hi];
```

```
        INST_R(list[lo]);
        INST_W(list[hi]);
        list[hi] = list[lo];
        INST_W(list[lo]);
        list[lo] = tmp;
    }
 return lo; // this is where the key is now
}//end int Partition
```

Q_Sort

Associativity	Block Size	Cache Size	Miss Rate(%)	Hit Time	Miss Penalty	Total Time
1	1	32	23.4	10	110	357840
1	1	64	23.4	10	110	357840
1	1	32	23.4	10	110	357840
1	1	64	23.4	10	110	357480
1	1	16	24.2	10	110	366310
1	1	8	24.2	10	110	366310
1	1	16	24.2	10	110	366310
1	1	8	24.2	10	110	366310
1	1	4	26.2	10	110	388530
1	1	4	26.2	10	110	388530
1	2	32	26.3	10	120	415720
1	2	64	26.3	10	120	415720

The table shows success of the top 12 caches sorted by the smallest rate. It worked well because of direct mapped caching since it’s the only parameter that remains constant. This is the best method since the data in the array will be accessed in sequence (therefore disregarding the increase/decrease of block size).

However, having plenty of blocks in the cache is necessary since the code calls for arbitrary memory access. By having a larger index, there could be more memory chunks retained in the cache. This renders the replacement method useless as cache blocks are replaced by memory addresses.

X-Sort

Associativity	**Block Size**	**Cache Size**	**Miss Rate(%)**	**Hit Time**	**Miss Penalty**	**Total Time**
1	1	32	14.0	10	110	253560
1	1	64	14.0	10	110	253560
1	1	32	14.0	10	110	253560
1	1	64	14.0	10	110	253560
1	2	32	14.1	10	120	269560
1	2	64	14.1	10	120	269560
1	2	32	14.1	10	120	269560
1	2	64	14.1	10	120	269560
1	1	16	16.0	10	110	275670
1	1	8	16.0	10	110	275670
1	1	16	16.0	10	110	275670
1	1	8	16.0	10	110	275670

Much like the previous table, this particular table shows the top 12 caches sorted by the smallest rate. Direct mapped caching also worked best, but the blocks were accessed in order (therefore they weren't revisited).

The algorithm is iterative, causing a noticeably lower miss rate. This grants more access to the memory than it should, thus becoming an advantage for temporal locality.