Dan Ruley
CS 3810
Study Assignment #9
December, 2019

Proof-of-Study:

For this problem I coded my cache simulations in Java, corroborating and checking some of my logic by hand to make sure things work out well. I wrote three separate methods and a few helper functions to simulate Direct Mapped, Fully Associative, and Set Associative caches. I coded the basic ideas and then debugged a bit until I got the same results as Professor Jensen for the set of addresses he used for the practice problems. Before I get to that, here is a breakdown of how some example caches look for the three types:

Fully Associative with 5 entries and 16 byte data blocks:

Total cache size: 5*(128(data) + 12(tag) + 3(LRU) + 1(valid)) = 720 bits ⇒ 120 bits left over.
This is the cache architecture:

Valid(1bit)	Tag(12 bits)	Data(16 bytes)	LRU(3 bits)

Addresses break down like this:

Tag bits	Offset bits
b15, b14, b13, b12, b11, b10, b9, b8, b7, b6, b5, b4	b3, b2, b1, b0

Memory access time:
1 cycle for a hit
10 + 1 + 16 = 27 cycles for a miss

Direct Mapped with 8 rows and 8 byte blocks:

Total cache size: 8*(64(data) + 10(tag) + 1(valid)) = 600 bits ⇒ 240 bits left over.

This is the cache architecture:

Valid(1bit)	Tag(10 bits)	Data(8 bytes)

Addresses break down like this:

Tag bits	Row bits	Offset bits
b15, b14, b13, b12, b11, b10, b9, b8, b7, b6	b5, b4, b3	b2, b1, b0

Memory access time:

1 cycle for a hit

10 + 1 + 8 = 19 cycles for a miss

<u>Set Associative with 4 rows, 8 byte blocks, and 2 ways</u>:
4*(2*(64(data) + 11(tag) + 1(valid) + 1(LRU)) = 616 bits ⇒ 224 bits left over.

This is the cache architecture:

/	Valid(1bit)	Tag(11 bits)	Data(8 bytes)	LRU(1 bit)
ROW 0				
\				
/	Valid(1bit)	Tag(11 bits)	Data(8 bytes)	LRU(1 bit)
ROW 1				
\				
/	Valid(1bit)	Tag(11 bits)	Data(8 bytes)	LRU(1 bit)
ROW 2				
\				
/	Valid(1bit)	Tag(11 bits)	Data(8 bytes)	LRU(1 bit)
ROW 3				
\				

Addresses break down like this:

Tag bits	Row bits	Offset bits
b15, b14, b13, b12, b11, b10, b9, b8, b7, b6, b5	b4, b3	b2, b1, b0

And here is the state of the cache after the second pass through the addresses:

/	Valid(1bit)	Tag(11 bits)	Data(8 bytes)	LRU(1 bit)
ROW 0	1	3	d|d|d|d|d|d|d|d	0
\	1	4	x|x|x|x|x|x|x|x	1
/	Valid(1bit)	Tag(11 bits)	Data(8 bytes)	LRU(1 bit)
ROW 1	1	3	y|y|y|y|y|y|y|y	0
\	1	4	z|z|z|z|z|z|z|z	1
/	Valid(1bit)	Tag(11 bits)	Data(8 bytes)	LRU(1 bit)
ROW 2	1	3	a|a|a|a|a|a|a|a	0
\	1	4	b|b|b|b|b|b|b|b	1
/	Valid(1bit)	Tag(11 bits)	Data(8 bytes)	LRU(1 bit)

ROW 3	1	2	c|c|c|c|c|c|c|c	0
\	1	3	g|g|g|g|g|g|g|g	1

Memory access time:
1 cycle for a hit
10 + 1 + 8 = 19 cycles for a miss

For the specific results of how each address is read and cached, please see the output of my Java program. The program also outputs the average CPI value for each cache type I simulated, but since there are quite a few of these, I decided to collate the results into three tables that correspond to each cache type.

Table 1: Average CPI for Direct Mapped Caches

Rows	Block SIze	Hits	Misses	Avg CPI
16	4	16	16	8
8	8	20	12	7.75
4	16	23	9	8.313
2	32	25	7	10.19
1	64	27	5	12.57

Table 2: Average CPI for Fully Associative Caches

Entries	Block SIze	Hits	Misses	Avg CPI
15	4	10	22	10.63
10	8	18	14	8.88
5	16	23	9	8.313
3	32	27	5	7.563
1	64	27	5	12.563

Table 3: Average CPI for Set Associative Caches

Rows	Block SIze	Num Ways	Hits	Misses	Avg CPI
16	4	1	16	16	8
8	4	2	15	17	8.44
8	8	1	20	12	7.75
4	4	4	16	16	8
4	8	2	19	13	8.313
4	16	1	23	9	8.313

2	4	8	13	19	9.313
2	8	5	20	12	7.75
2	16	2	22	10	9.13
2	32	1	25	7	10.19
1	4	16	13	19	9.313
1	8	10	20	12	7.75
1	16	6	23	9	8.313
1	32	3	27	5	7.563
1	64	1	27	5	12.563

It is clear that the best performing cache types are a Fully Associative Cache with 3 entries and 32 byte blocks and a Set Associative with 1 row, 3 way association, and 32 byte blocks. They both result in an average CPI of 7.563. It is unsurprising that these two caches yield the same results; a three way set associative cache with one row is identical in performance to a three-entry fully associative cache.

My conclusion from these data is that the spatial locality is primary factor for performance, at least given this specific set of addresses. It is interesting that some other cache types come close to 7.5 CPI but with far more misses. For example, the 8 row, 8 byte block Direct Mapped cache has an average CPI of 7.75 but actually has 12 misses instead of 5. The 32 byte, 3 way Set Associative/Fully Associative seems to hit a “sweet spot” where there are few misses and the large block size takes advantage of the spatial locality. However, increasing the block size any further than this results in huge CPI penalties for misses. In fact, with 32 byte blocks, the penalty is already quite severe but since the number of misses is so low it wins out over other cache architectures.

Please see below for the source code of my Java program, and please excuse any poor software engineering on my part - it had been several months since I wrote any Java before I began this assignment!

```
import java.util.ArrayList;
/*
 * Assumptions: 1 cycle for cache hits
 *
 * 1 + 10 + [1 cycle per byte] for cache misses
 * Total cache size must not exceed 840 bits
 */
```

```
public class StudyAssignment9 {

	public static void main(String[] args) {
		int[] addresses = new int[] { 4, 8, 12, 16, 20, 32, 36, 40, 44, 20, 32, 36, 40, 44, 64,
68, 4, 8, 12, 92, 96,
				100, 104, 108, 112, 100, 112, 116, 120, 128, 140, 144 };

		directMapped(addresses, 16, 4);
		directMapped(addresses, 8, 8);
		directMapped(addresses, 4, 16);
		directMapped(addresses, 2, 32);
		directMapped(addresses, 1, 64);

		fullyAssociative(addresses, 15,4);
		fullyAssociative(addresses, 10,8);
		fullyAssociative(addresses, 5,16);
		fullyAssociative(addresses, 3,32);
		fullyAssociative(addresses, 1,64);

		//16 row SA
		setAssociative(addresses, 16,4,1);
		//8 row SA's
		setAssociative(addresses, 8,4,2);
		setAssociative(addresses, 8,8,1);
		//4 row SA's
		setAssociative(addresses, 4,4,4);
		setAssociative(addresses, 4,8,2);
		setAssociative(addresses, 4,16,1);
		//2 row SA's
		setAssociative(addresses, 2,4,8);
		setAssociative(addresses, 2,8,5);
		setAssociative(addresses, 2,16,2);
		setAssociative(addresses, 2,32,1);
		//1 row SA's
		setAssociative(addresses, 1,4,16);
		setAssociative(addresses, 1,8,10);
		setAssociative(addresses, 1,16,5);
		setAssociative(addresses, 1,32,3);
		setAssociative(addresses, 1,64,1);
```

```
}

/**
 * Simulates a Direct Mapped cache
 *
 * @param addrs        - array of addresses to read/cache
 * @param rows         - # rows in the cache
 * @param block_size   - # of bytes / block
 */
public static void directMapped(int[] addrs, int rows, int block_size) {

	double hits_total = 0;
	double misses_total = 0;
	int offset_bits = logbase2(block_size);
	int row_bits = logbase2(rows);

	int[] dm_cache = new int[rows];

	//First pass to populate the cache
	for (int i = 0; i < addrs.length; i++) {
		int tag = addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits + row_bits));
		int row = (addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits))) % rows;

		dm_cache[row] = tag;
	}

	//Second pass for the "real" analysis
	for (int i = 0; i < addrs.length; i++) {
		int tag = addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits + row_bits));
		int row = (addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits))) % rows;

		String result;
		if (dm_cache[row] == tag) {
			hits_total++;
			result = "hit from row " + row;
		} else {
			misses_total++;
			result = "miss - cached to row: " + row;
		}
```

```java
				System.out.println("address: " + addrs[i] + ", tag: " + tag + ", result: " +
result);

				dm_cache[row] = tag;
			}

			System.out.println("Direct Mapped Cache - rows: " + rows + "    blocksize: " +
block_size + " bytes" +
					"\n" + "Total misses: " + misses_total + ", Total hits: "
					+ hits_total + "\nCPI for this set of addresses: "
					+ ((hits_total + misses_total * (11 + block_size)) / addrs.length) +
"\n");
		}

		/**
		 * Simulates a Fully Associative cache.
		 *
		 * @param addrs        - array of addresses to read/cache
		 * @param entries      - # entries in the cache
		 * @param block_size   - # of bytes / block
		 */
		public static void fullyAssociative(int[] addrs, int entries, int block_size) {

			double hits_total = 0;
			double misses_total = 0;
			int offset_bits = logbase2(block_size);

			int[] fa_cache = new int[entries];

			for (int i = 0; i < fa_cache.length; i++) {
				fa_cache[i] = -1;
			}

			int j;

			//First pass to get things warmed up
			for (int i = 0; i < addrs.length; i++) {
				boolean hit = false;
```

```
			int tag = addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits));
			for (j = 0; j < fa_cache.length; j++) {

				if (fa_cache[j] == tag) {
					swap(fa_cache, tag, j);
					break;
				}
				if (!hit) {
					swap(fa_cache, tag, -1);
				}
			}
		}

		//Do it for real now!
		for (int i = 0; i < addrs.length; i++) {

			int tag = addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits));

			boolean hit = false;
			String result = "";
			for (j = 0; j < fa_cache.length; j++) {
				if (fa_cache[j] == tag) {
					swap(fa_cache, tag, j);
					result = "hit from entry " + j;
					hits_total++;
					hit = true;
					break;
				}
			}
			if (!hit) {
				swap(fa_cache, tag, -1);
				misses_total++;
				result = "miss - cached to entry: " + j;
			}

			System.out.println("address: " + addrs[i] + ", tag: " + tag + " , result: " +
result);
		}
```

```
		System.out.println("Fully Associative Cache - entries: " + entries + "	blocksize: 
" + block_size + " bytes" +
					"\n" + "Total misses: " + misses_total + ", Total hits: "
					+ hits_total + "\nCPI for this set of addresses: "
					+ ((hits_total + misses_total * (11 + block_size)) / addrs.length) + 
"\n");
	}

	/**
	 * Simulates a Set Associative cache.
	 *
	 * @param addrs	- array of addresses to read/cache
	 * @param rows	- # rows in the cache
	 * @param block_size	- # of bytes / block
	 * @param setsize	- how many entries there are in a set that corresponds to
	 *		one row (e.g. for two-way, setsize = 2)
	 */
	public static void setAssociative(int[] addrs, int rows, int block_size, int set_size) {

		double hits_total = 0;
		double misses_total = 0;

		int offset_bits = logbase2(block_size);
		int row_bits = logbase2(rows);

//		int tagbits = (16 - offset_bits - row_bits);
//		int LRUbits = logbase2(set_size);
//		int SIZE = rows * (set_size * (block_size * 8 + tagbits + LRUbits + 1));
//		System.out.println(SIZE);
//		if (SIZE > 840) {
//			System.out.println("TOO BIG!!!!" + " rows: " + rows + " blocksize: " + 
block_size + " ways: " + set_size);
//			return;
//		}

		ArrayList<int[]> sa_cache = new ArrayList<>();

		//build up the array with a set for each row
		for (int i = 0; i < rows; i++) {
```

```
        sa_cache.add(new int[set_size]);
    }

    // first pass to populate the cache before analysis
    for (int i = 0; i < addrs.length; i++) {
        int tag = addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits + row_bits));
        int row = (addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits))) % rows;

        int[] rowcache = sa_cache.get(row);

        boolean hit = false;
        for (int j = 0; j < rowcache.length; j++) {

            if (rowcache[j] == tag) {
                swap(rowcache, tag, j);
                hit = true;
                break;
            }
        }
        if (!hit) {
            swap(rowcache, tag, -1);
        }
    }

    //second pass through the address array to perform the analysis.
    for (int i = 0; i < addrs.length; i++) {
        int tag = addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits + row_bits));
        int row = (addrs[i] / (int) Math.ceil(Math.pow(2, offset_bits))) % rows;

        int[] rowcache = sa_cache.get(row);

        String result = "";
        boolean hit = false;
        for (int j = 0; j < rowcache.length; j++) {
            if (rowcache[j] == tag) {
                swap(rowcache, tag, j);
                hit = true;
                hits_total++;
                result = "hit from row: " + row;
```

```
                        break;
                    }
                }
                    if (!hit) {
                        swap(rowcache, tag, -1);
                        result = "miss - cached to row: " + row;
                        misses_total++;
                    }

                    System.out.println("Address: " + addrs[i] + ", tag: " + tag + " ,
result: " + result);
                }

            System.out.println("Set Associative Cache - rows: " + rows + "    blocksize: " +
block_size + " bytes    ways: " + set_size + "\n"
            + "Total misses: " + misses_total + ", Total hits: "
                    + hits_total + "\nCPI for this set of addresses: "
                    + ((hits_total + misses_total * (11 + block_size)) / addrs.length) +
"\n");

        }


    /*
     * Places the tag at the end of the array and shifts everything else down to
     * represent it as being used less recently.
     */
    private static void swap(int[] associative_cache, int tag, int index) {

        // If tag was already the most recently used and it's in the set, do nothing
        if (index == associative_cache.length - 1)
            return;

        // If tag was not in the set, shift everything left and replace the oldest entry
        // with tag
        else if (index == -1) {
            for (int i = 0; i < associative_cache.length - 1; i++) {
                int tmp = associative_cache[i];
                associative_cache[i] = associative_cache[i + 1];
```

```
                associative_cache[i + 1] = tmp;
            }
            associative_cache[associative_cache.length - 1] = tag;
            return;
        }

        // Tag is in the set but it's not the most recent - adjust accordingly.
        else {
            for (int i = index; i < associative_cache.length - 1; i++) {
                int temp = associative_cache[i + 1];
                associative_cache[i + 1] = tag;
                associative_cache[i] = temp;
            }
        }
    }

    /*
     * Return the log base 2 of input as an integer (seriously, the Java math
     * library doesn't have this?!)
     */
    private static int logbase2(int x) {
        return (int) (Math.log(x) / Math.log(2) + 1e-10);
    }

}
```