Cache Simulation

Stefano Lupo - 14334933 - SS Computer Engineering - 16/12/17

The repository contains the code for simulating the operations of an instruction cache and data cache. A trace of over 2 million physical addresses collected from an Intel i486 microprocessor while gcc was compiling a program is used as the list of addresses requested from the cache by the CPU.

Setup

The setup contaions two caches - one for instructions and one for data. The instruction cache is a 16KB direct mapped cache *(L=16, K=1, N=1024)* and the data cache is a an 8-way 32KB cache *(L=16, K=8, N=256)*. Both caches have L=16 bytes meaning four separate 32bit words can be stored in each cache line.

The instruction cache is direct mapped *(K=1)* meaning it has only one tag in each set. Thus the `tag` extracted from each physical address is compared with exactly one tag in the cache.

The data cache is a set associative cache containing *N=256* sets with *K=8* directories in each. The set number is first extracted from the physical address and used to select the relevant set. Each of the k tags in that set are then compared in parallel against the tag extracted from the physical address. If a match is found, the data can then be returned.

The traces contain a `burstCount` field which represents the number of adjacent (same set and directory) addresses the CPU would like to retrieve from the cache. As discussed, there are four 32 bit words in each cache line *(L=16)* and all four of these words may be retrieved simultaneously from the cache in one operation. When reading data from memory address *A*, the 3 memory addresses adjacent to **A** (modulo 16 bytes alligned on a 16 byte boundary) may also be read simultaneously into the other three words in the same cache line. This takes advantage of *locality of reference* - addresses nearby to an address that has just been read are likely to be required in the future. Thus caching these ahead of time provides a significant performance improvement.

When the cache inevitably fills up, a protocol must be put in place in order to replace entries in the cache. The protocol used was **Least Recently Used** meaning the tags that had not been accessed in the longest amount of time were evicted first.

Implementation

The implementation consists of two main classes - an `Analyser` class and a `Cache` class. The analyser class is responsible for examining the meta data associated with each trace and routing the request to the appropriate cache (eg if the request was an instruction fetch, it goes to the instruction cache etc). There is also a third `Runner` class which is just responsible for reading in / displaying (in dec / binary / hex) some of the data to help understand the make up of each of the traces.

Analyser Class

This class is instantiated by providing it with one or two Cache objects to be used as the caches for the simulation and an array of traces. Once instantiated an `analyseAssignment()` method may be called which does the following:

- Iterates over each trace (skipping every second 32bit word which is not required)
- Extracting the `cycleType` in order to determine if this trace should be routed through the caches
  - This is done by *ANDING* the 32bit value with the `CYCLE_MASK` and (unsigned) shifting it right by `CYCLE_SHIFT` places.
  - This extracts the three most significant bits which encode the cycle type.

  ```
  int cycleType = (word & 0xE0000000) >>> 29;
  ```

- Extracting the `burstCount` in order to inform the cache of how many adjacent values the CPU would like to retrieve.
  - This is done by *ANDING* the 32bit word with the `BURST_MASK` and shifting it right by `BURST_SHIFT`
  - This extracts bits 27 and 28 which encode the `burstCount` .

  ```
  int burstCount = (word & 0x18000000) >>> 27;
  ```

- Extracting the physical `address` in order to pass it to the `cache` objects.
  - This is done by first *ANDING* the 32bit word with the `ADDRESS_MASK` and then shifting it left twice in order to fill the missing two address bits (which are ommited from the trace file)
  - This extracts bits 24 to 2 from the trace and pads it to the required 25bit physical address.

  ```
  int address = word & 0x007FFFFF;
  address <<= 2;
  ```

- Ensures these values are valid and passes the `address` and `burstCount` to the appropriate cache depending on `cycleType` .

Cache Class

This class is responsible for implementing the actual behaviour of a cache. It can be instantiated with values for `l` , `k` and `n` , and an optional `addressSize` (which defaults to 25). This does the following:

- Computes `setBits` and `offsetBits` using log2 - the number of bits required for encoding the set (indexes N) and offset (indexes L).
- Computes the masks and shifts for the set, offset and tag, for the given `l` , `k` , `k` and `addressSize` configuration by the following:

```
int offsetMask = (1 << offsetBits) - 1;
int offsetShift = 0;
int setMask = (1 << setBits) - 1;
int setShift = offsetBits;
int tagMask = (1 << (addressSize - offsetBits - setBits)) -1;
int tagShift = offsetBits + setBits;
```

The Cache class contains a Hash Map (to represent a *set*) which maps from a `setNumber` --> `HashMap` . This second Hash Map then maps from `tag` --> `TagData` object. This `TagData` object contains a `timestamp` (which is just an integer) in order to implement LRU and could contain further objects for representing the *actual data* in the cache (not needed for our purposes).

This allows us to lookup our cache first by set number, returning a hash map which represents a `set` . This `set` hash map can then be looked up using the `tag` integer contained in the physical address, allowing us to simulate extracting data from the cache.

The Cache class also has the `feedAddress(int physicalAddress, int burstCount)` method which does the following:

- Extracts the `setNumber` , `offset` and `tagNumber` from the phsyical address using the above masks and shifts.
- Extracts the required `set` (by `setNumber`) from the `sets` Hash Map and extracts the `tagData` from that set (by `tagNumber` and checks:
  - if `tagData` is not null, this is a cache hit: `hits += (burstCount + 1)`
  - else it was a miss: `hits += burstCount` , `misses++`
    - if the k directories are not all filled - insert the new tag in the empty directory
    - else remove LRU tag (the tag with the lowest `timestamp` integer) in that set and replace it with the new tag.

Simulation Results

The simulation produced the following results.

```
Data Reads: 41.870308%
Data Writes: 14.540052%
Instruction Reads: 40.848686%
Skips: 2.7409554%
Cycle Types (encoded): [8836, 38901, 100, 9645, 856659, 0, 878084, 304927]
Analysed 2039670 traces, skipped 57482

Instruction Cache
Total accesses: 3426555
Misses: 128052
Hits: 3298503
Hit Rate: 96.262955%

Data Cache
Total accesses: 1188855
Misses: 31778
Hits: 1157077
Hit Rate: 97.327%

File Reading Time: 70ms
Cache Simulation Time: 214ms
Total Elapsed Time: 284ms
```

Code

Runner.java

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.IntBuffer;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Arrays;

public class Runner {

    private final static String FILE_NAME = "gcc1.trace";
    private final static String OUTPUT_FILE_NAME = "subtrace.txt";


    public static void main(String... aArgs) throws IOException, IllegalArgumentException{

        // Start a timer
        final long startTime = System.currentTimeMillis();

        // Read the file
        byte[] bytes = readSmallBinaryFile(FILE_NAME);


        // Create an int array from bytes
        int[] traces = getAsIntArray(bytes, false);
        System.out.println("Number of 32bit ints: " + traces.length + ", number of traces = " + traces.length / 2 + "\n");

        // See how long it takes to read the file
        final long timeAfterRead = System.currentTimeMillis();

        // Get subset of integers for examination
        // int[] littleEndianSnippet = Arrays.copyOfRange(littleEndian, 250000 , 250100);

        // Print out the contents
        // displayIntArray(littleEndian);

        // Write a subset to test file for inspection
        // writeSmallBinaryFile(bytes, OUTPUT_FILE_NAME, 100);


        // Run the assignment simulation
        runAssignment(traces);

        // Run the tutorial
//        runTutorial();

        // Stop the timer
        final long finishTime = System.currentTimeMillis();

        System.out.println("\nFile Reading Time: " + (timeAfterRead - startTime) + "ms");
        System.out.println("Cache Simulation Time: " + (finishTime - timeAfterRead) + "ms");
        System.out.println("Total Elapsed Time: " + (finishTime - startTime) + "ms");

    }


    /*
     **********************************************************
     * Helper Methods
     **********************************************************/

    /**
     * Sets up and runs the simulation for the tutorials (like the online animation)
     */
    public static void runTutorial() {
        int[] tutorialData = {
                0x0000, 0x0004, 0x000c, 0x2200, 0x00d0, 0x00e0, 0x1130, 0x0028, 0x113c, 0x2204,
                0x0010, 0x0020, 0x0004, 0x0040, 0x2208, 0x0008, 0x00a0, 0x0004, 0x1104, 0x0028, 0x000c,
                0x0084, 0x000c, 0x3390, 0x00b0, 0x1100, 0x0028, 0x0064, 0x0070, 0x00d0, 0x0008, 0x3394
        };

        // Pick some parameters here and pick same on website
        Cache iCache = new Cache(32, 8, 4, 8);
        Analyser analyser = new Analyser(iCache, tutorialData);
        analyser.analyseTutorial();
    }
```

```java
/**
 * Sets up and runs the simulation for the assignment
 * @param traces they int array from the tracefile
 */
private static void runAssignment(int[] traces) {
    // Create the caches
    Cache iCache = new Cache(16, 1, 1024);
    Cache dCache = new Cache(16, 8, 256);

    // Analyse the cache
    Analyser analyser = new Analyser(iCache, dCache, traces);
    analyser.analyseAssignment();
}

/**
 * Builds array of 32 bit words from the bytes read from the trace file
 * @param bytes the byte array to get the ints from
 * @param bigEndian true if data is in big endian
 * @return int array
 */
private static int[] getAsIntArray(byte[] bytes, boolean bigEndian) {
    IntBuffer intBuffer = ByteBuffer.wrap(bytes)
            .order(bigEndian ? ByteOrder.BIG_ENDIAN : ByteOrder.LITTLE_ENDIAN)
            .asIntBuffer();
    int[] array = new int[intBuffer.remaining()];
    intBuffer.get(array);
    return array;
}


/**
 * Displays the byte arrays in a specific radix formatted nicely
 * @param bytes array to print
 * @param radix base to print in
 * @param numToShow number of elements from the array to print
 */
private static void displayBytes(byte[] bytes, int radix, int numToShow) {
    System.out.println("Displaying in base " + radix);
    for(int i=0;i<(numToShow*8); i++){

        // Split the byte strings nicely
        if(i % 8 == 0) {
            System.out.println();
        } else if(i % 4 == 0) {
            System.out.print("  -  ");
        }

        // Create the String
        String str = Integer.toString(bytes[i], radix);

        // Pad binary strings with X's
        if(radix == 2){
            str = String.format("%8s", str).replace(' ', 'X');
        }

        System.out.print(str + " ");
    }
    System.out.println("\n");
}


/**
 * Prints the integer array in multiple bases
 * @param arr array to print
 */
private static void displayIntArray(int[] arr) {
    System.out.println("Hex, Dec, Binary");
    for(int i : arr) {
        System.out.println(Integer.toString(i, 16));
        System.out.println(Integer.toString(i));
        System.out.println(Integer.toString(i, 2) + "\n");
    }
}

/**
 * Reads the bytes in from the tracefile
 * @param filename name of tracefile
 * @return byte array containing all bytes in that file (little endian)
 * @throws IOException if anything scary happens when reading the file
 */
private static byte[] readSmallBinaryFile(String filename) throws IOException {
    Path path = Paths.get(filename);

    return Files.readAllBytes(path);
}
```

```java
    /**
     * Writes a subset of the bytes to a file for easier viewing
     * @param bytes the full bytes array
     * @param filename name of the output file
     * @throws IOException if something scary happens when writing the file
     */
    private static void writeSmallBinaryFile(byte[] bytes, String filename, int number) throws IOException {
        Path path = Paths.get(filename);
        byte[] snippet = Arrays.copyOfRange(bytes, 0 , number);
        Files.write(path, snippet); //creates, overwrites
    }
}
```

Analyser.java

```java
import java.util.Arrays;

class Analyser {

    private final static boolean DEBUG = false;
    private final static int DEBUG_MAX_TRACES_TO_PROCESS = 100;

    private final static int ADDRESS_MASK = 0x007FFFFF;
    private final static int CYCLE_MASK = 0xE0000000;
    private final static int BURST_MASK = 0x18000000;

    private final static int CYCLE_SHIFT = 29;
    private final static int BURST_SHIFT = 27;

    private Cache iCache, dCache;
    private int[] traces;

    /**
     * Creates an analyser for simulations with two caches
     * Assumes traces is [word00, word01, word10, word11 ...etc]
     * Assumes words are in big endian format
     * Assumes wordsX0 lowest (address) byte have not yet been shifted
     * @param iCache the instruction cache to be used
     * @param dCache the data cache to be used
     * @param traces int array of the traces
     */
    Analyser(Cache iCache, Cache dCache, int[] traces) {
        this.iCache = iCache;
        this.dCache = dCache;
        this.traces = traces;
        System.out.println("Traces Length: " + this.traces.length);
    }

    /**
     * Creates an analyser for simulations with one cache
     * @param cache the cache to be used for the simulation
     * @param traces the traces to be used for the simulation
     */
    Analyser(Cache cache, int[] traces) {
        this.iCache = cache;
        this.traces = traces;
    }

    /**
     * Runs the basic (single cache) tutorial simulations
     */
    void analyseTutorial() {
        for(int trace : traces) {
            iCache.feedAddress(trace, 0);
        }

        System.out.println("\nInstruction Cache");
        iCache.printResults();
    }


    /**
     * Runs the more complex assignment (multi cache) simulations
     * @throws IllegalArgumentException if something scary happens
     */
    void analyseAssignment() throws IllegalArgumentException {

        // Keep track of the different cache access to ensure everything seems appropriate
        int skips = 0;
        int dataReads = 0;
        int dataWrites = 0;
        int instructionReads = 0;

        // Keeps track of the frequencies of the different cache accesses
        int[] types = new int[8];
```

```java
    // Process some or all of the traces
    int tracesToAnalyse = DEBUG ?  2 * DEBUG_MAX_TRACES_TO_PROCESS : traces.length;
    System.out.println("Analysing traces: " + tracesToAnalyse / 2);

    // We only actually process half that number of traces since there is two words per trace
    float actualNumTraces = tracesToAnalyse / 2;

    // Iterate over all of the traces
    for(int i=0;i<tracesToAnalyse; i+=2) {

        int word = traces[i];

        // Print the trace
        // String binary = Integer.toString(word, 2);
        // System.out.println(binary.length() + ": " + binary);

        // Extract the relvant bits (three angles is unsigned bit shift)
        int cycleType = (word & CYCLE_MASK) >>> CYCLE_SHIFT;
        int burstCount = (word & BURST_MASK) >>> BURST_SHIFT;

        // Get bits 24 - 2 of the address
        int address = word & ADDRESS_MASK;

        // Shift bits to add the missing two LS zeros
        address <<= 2;


        // Ensure trace was valid
        if(burstCount > 3 || burstCount < 0 || cycleType > 7 || cycleType < 0) {
            String error = "Invalid Trace: cycle type: " + cycleType + ", burstCount: " + burstCount;
            throw new IllegalArgumentException(error);
        }

        // Increment number of these cycles that have occurred
        types[cycleType]++;


        // Feed the address to the appropriate cache
        // 4: instruction read, 6: data read, 7: data write
        if(cycleType == 4) {
            instructionReads++;
            iCache.feedAddress(address, burstCount);
        } else if(cycleType == 6) {
            dataReads ++;
            dCache.feedAddress(address, burstCount);
        } else if(cycleType == 7) {
            dCache.feedAddress(address, burstCount);
            dataWrites ++;
        } else {
            skips++;
        }
    }

    // Print the distributions of cycle types and other info
    System.out.println("\n\nData Reads: " + dataReads * 100 / actualNumTraces + "%");
    System.out.println("Data Writes: " + dataWrites * 100 / actualNumTraces+ "%");
    System.out.println("Instruction Reads: " + instructionReads * 100  / actualNumTraces + "%");
    System.out.println("Skips: " + skips * 100 / actualNumTraces + "%");
    System.out.println("Cycle Types (encoded): " + Arrays.toString(types));
    System.out.println("Analysed " + (instructionReads + dataReads + dataWrites) + " traces, skipped " + skips);

    System.out.println("\nInstruction Cache");
    iCache.printResults();

    System.out.println("\nData Cache");
    dCache.printResults();
}

/**
 * Adds missing bits to LSB of the address
 * This is **NOT** used as I _think_ its the same as the shifting done instead
 */
private int addMissingBitsToStart(int address) {
    // System.out.println("Initial Address: " + Integer.toBinaryString(address));

    int addressUpper = address & 0xFFFFFF00;
    // System.out.println("Address Upper: " + Integer.toBinaryString(addressUpper));

    int addressLower = address & 0x000000FF;
    // System.out.println("Address Lower: " + Integer.toBinaryString(addressLower));

    addressLower <<= 2;
    // System.out.println("Address Lower Shifted: " + Integer.toBinaryString(addressLower));

    address = addressUpper + addressLower;
    // System.out.println("Final Address: " + Integer.toBinaryString(address));
```

```
        return address;
    }
}
```

Cache.java

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;

class Cache {

    private static final int DEFAULT_ADDRESS_SIZE = 25;
    private int addressSize = DEFAULT_ADDRESS_SIZE;


    private int l;
    private int k;
    private int n;

    private int setMask;
    private int offsetMask;
    private int tagMask;

    private int setShift;
    private int offsetShift;
    private int tagShift;

    private int hits = 0;
    private int misses = 0;
    private int timestamp = 0;
    private HashMap<Integer, HashMap<Integer, TagData>> sets;

    /**
     * Create a Cache with a specified addressSize
     * @param l number of bytes in each cache line
     * @param k number of cache lines per set (directories)
     * @param n number of sets
     * @param addressSize size of the physical address (required for calculating # bits for offset)
     */
    Cache(int l, int k, int n, int addressSize) {
        this(l, k, n);
        this.addressSize = addressSize;
    }

    /**
     * Creates a cache with default (assignment) address size of 25 bits
     * @param l number of bytes in each cache line
     * @param k number of cache lines per set (directories)
     * @param n number of sets
     */
    Cache(int l, int k,  int n){
        this.l = l;
        this.k = k;
        this.n = n;
        System.out.println("Created " + l + ", " + k + ", " + n + " cache (" + l*k*n + "kb)");

        // Get number of bits for l
        Double math = (Math.log(l) / Math.log(2));
        int offsetBits = math.intValue();

        // Get number of bits for n
        math = Math.log(n) / Math.log(2);
        int setBits = math.intValue();
        System.out.println(offsetBits + ", " + setBits);

        // Offset mask is least significant l bits (after shifting)
        offsetMask = (1 << offsetBits) - 1;
        offsetShift = 0;
        System.out.println("Offset Mask: " + Integer.toBinaryString(offsetMask));

        // Set mask is least significant n bits (after shifting)
        setMask = (1 << setBits) - 1;
        setShift = offsetBits;
        System.out.println("Set Mask: " + Integer.toBinaryString(setMask));

        // Tag mask is least significant remainder of bits (after shifting)
        tagMask = (1 << (addressSize - offsetBits - setBits)) -1;
        tagShift = offsetBits + setBits;
        System.out.println("Tag Mask: " + Integer.toBinaryString(tagMask) + "\n");

        // Instantiate the hash maps for our cache
        sets = new HashMap<>(n, 1);
        for(int i=0; i<n; i++) {
            sets.put(i, new HashMap<>(k, 1));
```

```java
        }
    }

    /**
     * Requests data from the cache
     * @param physicalAddress "addressSize" bit physical address
     * @param burstCount the number of adjacent memory address requested
     * @throws IllegalArgumentException if the physical address is invalid
     */
    void feedAddress(int physicalAddress, int burstCount) throws IllegalArgumentException{

        // Extract the info
        int setNumber = (physicalAddress >> setShift) & setMask;
        int offset = (physicalAddress >> offsetShift) & offsetMask;
        int tagNumber = (physicalAddress >> tagShift) & tagMask;

        // Ensure values are valid
        if(setNumber < 0 || setNumber > setMask || offset < 0 || offset > offsetMask || tagNumber < 0 || tagNumber > tagMask
            String error = "Invalid Cache Values: setNumber: " + setNumber+ ", tagNumber: " + tagNumber + ", offset: " + off
            throw new IllegalArgumentException(error);
        }

        // Display the values
        // System.out.println("\n" + Integer.toHexString(physicalAddress));
        // System.out.println("Set: " + setNumber + ", Offset: " + offset + ", Tag: " + tagNumber + ", Burstcount: " +burstC

        // Get the k tags in this set
        HashMap<Integer, TagData> set = sets.get(setNumber);

        // Check for the tag we are interested in
        TagData tagData = set.get(tagNumber);

        // Check for tag match
        if(tagData != null) {
            // System.out.println(Integer.toHexString(physicalAddress) + ": Hit found");
            hits += burstCount + 1;

            // Update this tags last access time
            tagData.lastAccess = ++timestamp;
        } else {
            // System.out.println(Integer.toHexString(physicalAddress) + ": Miss");
            misses++;

            // Think the memory address that were adjacent count as hits since they will technically be read from the cache?
            hits += burstCount;

            // Check if k directories are full
            if(set.values().size() < k) {
                // System.out.println("Compulsory miss - k directories not full, inserting..");
                set.put(tagNumber, new TagData(++timestamp));
            } else {
                int lruTag = getLRU(set);
                // System.out.println(Integer.toHexString(physicalAddress) + ": directory full, removing lru: " + lruTag);
                set.remove(lruTag);
                set.put(tagNumber, new TagData(++timestamp));
            }
        }
    }


    /**
     * Performs the least recently used algorithm
     * @param set the set from the cache that needs to have an eviction
     * @return the tag to be evicted
     */
    private int getLRU(HashMap<Integer, TagData> set) {

        // Find the tag with the smallest timestamp as its last access
        int minAccess = Integer.MAX_VALUE;
        int lruTag = -1;
        for(Map.Entry<Integer, TagData> mapEntry : set.entrySet()) {
            int lastAccess = mapEntry.getValue().lastAccess;
            if(lastAccess < minAccess) {
                minAccess = lastAccess;
                lruTag = mapEntry.getKey();
            }
        }

        return lruTag;
    }

    /**
     * Prints the results of the simulation
     */
    void printResults() {
        System.out.println("Total accesses: " + (misses + hits));
        System.out.println("Misses: " + misses);
        System.out.println("Hits: " + hits);
```

```
        System.out.println("Hit Rate: " + (float)hits * 100 / (hits + misses) + "%");
    }

    /**
     * Data structure which could be expanded to actually hold some data
     */
    class TagData {

        int lastAccess;

        TagData() {
            this.lastAccess = 0;
        }

        TagData(int lastAccess) {
            this.lastAccess = lastAccess;
        }
    }
}
```