CS 610 Semester 2020–2021-I: Assignment 1

9th September 2020

Due Your assignment is due by Sep 20 2020, 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not plagiarize or turn in solutions from other sources. You will be PENALIZED if caught.

Submission

- Submission will be through mooKIT.
- Submit a PDF file with name "<roll-no>.pdf". We encourage you to use the LaTeX typesetting system for generating the PDF file.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Problem 1 **[20 marks]**

Consider the following loop.

```
    double s = 0.0, A[size];
    int i, it, stride;
    for (it = 0; it < 10 * stride; it++) {
        for (i = 0; i < size; i += stride) {
            s += A[i];
        }
    }
```

Assume a 4-way set-associative cache with capacity of 256 KB, line size of 32 B, and word size of 8 B (for `double`). Furthermore, assume the cache is empty before execution, and uses an LRU replacement policy.

Given `size`=32K, determine the total number of cache misses on `A` for the following access strides: 1, 4, 16, 32, 2K, 8K, and 32K.

Problem 2 **[40 marks]**

Consider a cache of size 32K words and line of size 8 words, and arrays 512×512. Perform cache miss analysis for the *ikj* and the *jik* forms of matrix multiplication (shown below) considering direct-mapped and fully-associative caches. The arrays are stored in row-major order. To simplify analysis, ignore misses from cross-interference between elements of different arrays (i.e., perform the analysis for each array, ignoring accesses to the other arrays).

Listing 1: ikj form

```
for (i = 0; i < N; i++)
    for (k = 0; k < N; k++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```

Listing 2: jik form

```
for (j = 0; j < N; j++)
```

```
    for (i = 0; i < N; i++)
        for (k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Your solution should have a table to summarize the total cache miss analysis for each loop nest variant and cache configuration, so there will be four tables in all. Briefly justify your computations.

Problem 3 [30 marks]

Consider the following code.

```
double y[4096], X[4096][4096], A[4096][4096];
for (k = 0; k < 4096; k++)
    for (j = 0; j < 4096; j++)
        for (i = 0; i < 4096; i++)
            y[i] = y[i] + A[i][j] * X[k][j];
```

Assume a direct-mapped cache of capacity 16 MB, with 64 B cache line and a word of 8 B. Assume that there is negligible interference between the arrays `A`, `X`, and `y` (i.e., each array has its own 16 MB cache for this question), and arrays are laid out in row-major form.

Estimate the total number of cache misses for `A` and `X`.

Problem 4 [100 points]

Implement a parser program in Java to automate counting cache misses. The input Java test cases allow `for` loop nests and up to 3D array references. You need to parse the `.java` file, collect the necessary information, and report the cache misses for each array in each test.

See the attachment which should help you with your setup.

You will use a popular top-down (LL) parser called ANTLR. The restricted grammar to parse the test cases is provided in a file called `LoopNest.g4`. You should not need to change the grammar. You can use the following instructions to generate the lexer, parser, and the syntax analysis template. Extend the necessary callbacks in `Analysis.java` to implement your analysis.

Run the program. Read the accompanying `LoopNest.g4` and `run.sh` files to get familiar with the ANTLR environment.

You can use the included shell script `run.sh` to automate using ANTLR.

```
bash run.sh TestCase1.java
```

Visualize the parse tree for a given test case to get a sense of what callbacks you should implement.

```
grun LoopNest tests –gui < TestCase1.java
```

Implementation. When you compile the grammar `LoopNest.g4`, it will autogenerate the lexer and parser stub files. One such autogenerated file will be `LoopNestBaseListener.java`, which includes empty callbacks for all rule entry and exits. For example, during parsing, whenever the parser will see a method declaration, it will call `enterMethod`, and after walking through all the child nodes, it will call `exitMethod`. You need to extend the callback methods to record relevant information like cache dimensions and array accesses, and compute the number of cache misses.

Note that `LoopNestBaseListener.java` is autogenerated and will be overwritten every time the grammar is compiled. Do NOT directly edit `LoopNestBaseListener.java`. Instead, you should extend `LoopNestBaseListener.java` and write your code there. In the attached files, `Analysis.java` is one such listener.

The longest chain of rules is for the rule `expression`. You can simplify how you maintain information across the rules related to different expression types. Note that the test cases will use a restricted syntax, and our simple cache model ignores the effect of local variables and arithmetic computations.

Note that you may have to substitute the values of constants during parsing. For example, if a declaration is `int[][] A = new int[N][N]`, you will have to keep track and infer the proper value of the array dimensions.

For every test case, you should create a `HashMap<String, Long>`. The keys of this map will be the names of the arrays present in a test case. The values will be the number of cache misses.

You will create such maps for all test cases and collect them in a `List<HashMap<String, Long>>`. After going through all the test cases, you will serialize this object into a file `Results.obj`. The name of the file MUST be `Results.obj`. We have included a serialization wrapper in `Analysis.java` (search for `FIXME:`).

Assumptions

- There are two special variables: `cachePower` will contain the size of the cache and `blockPower` indicates the size of a cache block. The cache and block sizes are specified in powers of two, i.e., a size of k indicates that the total cache size is 2^k Bytes.
- There will be only one variable `cacheType` of type `String` in a test case. Its values can be any of `FullyAssociative`, `SetAssociative`, and `DirectMapped`.
- You can assume that the input test methods will be syntactically valid Java snippets. You NEED NOT do any error checking in your implementation.
- In all the `for` loops, the lower bound will always start from 0.
- There will be at most 3 dimensions for any array.
- Array dimensions will always be a power of 2.

Evaluation The output format should be strictly followed since we will use automated scripts to evaluate the submissions. The scripts will call your program as follows.

```
java Driver Testcases.t
```

where `Testcases.t` is a file provided as a command line argument and will contain one or more test cases.

ANTLR Resources Feel free to browse through additional ANTLR material on the Internet.

- `https://github.com/antlr/antlr4/blob/master/doc/getting-started.md`
- Tutorial 1: `https://www.baeldung.com/java-antlr`
- Tutorial 2: `https://tomassetti.me/antlr-mega-tutorial/`