Assignment 4 TDT4200

raymont

October 2016

Part 1 Theory

Problem 1, Memory and Caching

a - Types of cache misses

There are three types of cache misses, also referenced as "The Three C's";

Compulsory This is the first kind of cache misses that occurs the first time the program tries to access a block of memory. The "compulsory" cache miss is somewhat unavoidable, but can be reduced by loading the data needed in advance.

Capacity This type of miss happens when the the working data set is larges than cache capacity and a block of memory have to be discarded form cache to in order to make room for more urgent data. The miss is when this discarded memory block is reaccessed later in the program execution. The best way to avoid this kind of cache misses is using temporal and spatial locality such that the least used data that is far away from the current used data is discarded to lower the probability of missing the data. Another option is to add a small buffer to temporarily store the discarded data, and search this buffer before entering main memory when the data is needed again.

Conflict Since the caches is substantially smaller than main memory, a set associative or direct mapping block placement is used. This may cause a conflict miss when several blocks are mapped to the same cache line. One strategy to avoid such misses is applying higher cache associativity or a pseudo-associative cache where the cache has a virtual second half, with flipping the most significant bit of the index field to search in the "psudo set". Other from more advanced associative mapping, this kind of cache misses is unavoidable given a particular associative scheme.

b - Access patterns

I In this code snippet the access pattern exhibits a spatial locality, since for each data access there is a high correlation with the accessing of data that is stored close to the accessed data.

II Here the data is reaccessed in order repeatedly. It is therefore clear that the code have both a spatial and a temporal access pattern.

III In the final code snippet, the data points are reused frequently, but they are stored relatively far off each other. This causes the code to have only a temporal access pattern.

Problem 2, Branching

a - Branch prediction

When the processor enters parts of a program that introduce branches though a condition statement, it originally had to wait until the condition is checked before executing the code following the branch. This is a waist of time and thus branch prediction is used. The way branch prediction works is that it tries to guess which branch the condition checking is most likely to take and executes it immediately. If the branch predictor later detect that the prediction was wrong, it dumps the work done after the prediction point, and restart on the right branch. If the condition is met in succession, the branch prediction keeps a record of the outcome, and improve its guessing. This way a lot of cycles can be saved if the branch predictor guess the correct branch most of the time and thus increase the performance of the code. This is why the condition that is most likely to occur should be the first of the available branches.

b - Prediction in practice

We here look at a loop with two possible branches for each iteration. If the condition, taking time p, succeeds a fast function is performed on the special elements with time f. If the condition fails, a slower function, with run time s, is performed. A miss prediction delay takes time b, and the fraction of elements satisfying the first prediction is $r < 0.5$ We want to find the critical value of r where it is cheaper to only use the slow function given a condition if it always predicts the last result and starts off with the first branch;

I) All special elements at start of a: We will here get a single miss prediction:

$$rn(f+p)+b+(1-r)ns < ns \tag{1}$$
$$rn(f+p-s) < ns-b \tag{2}$$
$$r < \frac{ns-b}{n(f+p-s)} \tag{3}$$
$$r < \frac{s}{f+p-s} - \frac{b}{n(f-s)} \tag{4}$$

This means that when the fraction of r becomes equal or greater than the fraction of $\frac{ns-b}{n(f+p-n)}$.

II) Special elements are evenly distributed in a: In this case we will get a miss prediction for both entering and leaving all of the special elements.

$$rn(f+2(p+b))+(1-r)ns < ns \tag{5}$$
$$r(f+2(p+b)-s) < 0 \tag{6}$$
$$f+2(p+b) < s \tag{7}$$

As wee see, here the benefit threshold of using a branch is independent of the fraction between the two types of elements. This is as expected since we only will have a speedup if the total time of testing, delay and computing the fast function is faster than simply performing the slow function.

III) Special elements are randomly distributed in a: In this case we want to look at the expected value of the run time. In the two previous examples, we looked into two extrema, and can thus take an average of the two to get the expected result of a random distributed array.

$$[rn(f+p)+b+(1-r)ns]+[rn(f+2(p+b)+(1-r)ns] < 2ns \tag{8}$$
$$rn(2f+3p+2b-2s) < -b \tag{9}$$
$$r < \frac{b}{2s-2f-3p-2b)} \tag{10}$$

As long as the fraction r is smaller than the final fraction computed above, we will expect in average a speedup by using a branch to apply the fast function for special elements.

Problem 3, Vectorization

We will here consider the following for loop:

```
1  for(int i = 0; i < 1024; i++){
2      a[i] = b[i] + c[i];
3  }
```

a) SIMD instructions

A SIMD instruction is a single instruction on multiple data. This can be described as performing a instruction on a vector where each element is computed in parallel opratin units. This means that on eg. a x86 processor can apply library functions that use SIMD instructions to perform the same operation on several data points in parallel.

b) Speedup by applying vectorization

Here we look at how much faster the loop above would execute if it was vectorized using vector instructions. It is here assumed that there are computations on doubles which is 64 bits.

I) 128 bit vector instruction Here we see that the vector instructions can hold two doubles($\frac{128}{64} = 2$), which means that we get a speedup by a factor of two.

II) 512 bit vector instruction In this case we see that the vector instruction can hold $\frac{512}{64} = 8$ doubles such that we get a speedup by a factor of eight.

Problem 4, Optimization

Here we look at a code computing the $sin(x)$ using the Taylor series expansion. It also uses an external function computing the factorial. Following I will suggest four different ways to speed up the above code.

I) - Avoid branching: The code is heavily prone to branching and miss predictions as it switches between two branches for every iteration. It is regular enough such that the branch predictor can pick up the pattern, but before this happens a lot of cycles is wasted. The simplest way to avoid this is by performing two iterations simultaneously and thus removing the branching completely;

```
1  for(i = 0; i < 100; i+=2){
2      r+=pow(x,i*2+1) / factorial(i*2+1);
3      r-=pow(x,(i+1)*2+1) / factorial((i+1)*2+1);
4  }
```

II) - Improve branching: In the function *factorial* the least used branch is placed on top of the selection of cases. By switching the branch order, we can thus improve the function by letting the actually most used branch be stated first.

```
1  double factorial(int n){
2      if(n>2)
3          return n*factorial(n-1);
4      else
5          return 1.0;
6  }
```

III) - Avoid recursive calls A recursive call can often make the code more compact and readable, but it is not necessarily the most efficient choice. The reason for this is that repeatedly calling a function from itself causes the allocation of a new stack frame. This is not always the case and can wary between compilers and the usage of flags, that may eliminate this overhead. The cost also varies between programming languages.

```
1  double factorial(int n){
2      if(n<2)
3          return 1.0;
4      else{
5          double result =1;
6          for(int i=1;i<n;i++)
7              result = result*i;
8          return result;
9      }
10
11 }
```

IV) - Reusing computations II: Instead of computing the power from screatch for each iteration, it can be stored and reused. This can also be done with the computation to omit the call to the factorial function.

```
1  double power = x;
2  for(i=0;i<100;i++){
3      power = power*x*x;
4      ...
5          r+=power / factorial(i*2+1);
6      ...
7          r-=power / factorial(i*2+1);
8      ...
9  }
```

Problem 5, OpenMP schedules

In this problem we look at parallelizing the following code with OpenMP with different schedules based on the expression for *arg*. The execution time of compute is prportioal to the value of its argument:

```
1  for(int i = 0; i < 1024; i++){
2      int arg = ...
3      compute(arg);
4  }
```

a) Static

The static scheduling is best when the expected computational cost is equal for all of the iterations. The simplest expression exploiting this is $arg = 1$.

b) Dynamic

Dynamic scheduling is best suited when there is a high uncertainty of how costly each iteration may be and it can wary from one iteration to another. A simple example can be $arg = i\%42$. After a thread are done computing it's assigned chunk of iterations, it requests anther chunk, and thus we avoid eg. that one thread have a significant more costly iteration block.

c) Guided

Guided scheduling works much the same as dynamic scheduling with blocks of iterations assigned to each thread. IN addition the size of the blocks is reduced for each request. This is great for loops where one expect the computational cost to steadily vary with along with the iterations. An example of this may be $arg = i$. Here the cost increases for each iteration, and thus it is beneficial to reduce the amount of iterations assigned. To avoid one thread to receive all the heaviest iterations a low minimum chunk size should be selected.