Measurement of Memory Hierarchy Parameters

Professor **Xiaoming Li**

Fall 2014

Pouya Fotouhi

Sid Raskar

Introduction

In this project, we supposed to measure the parameters of memory hierarchy and latency of operations. Our approach is based on the X-ray framework presented by Yotov et al. To explain our progress, we've divided our report into two sections: Implementation Phase and Results. The source code for this project can provided for an interested reader.

Implementation Phase:

We decided to use C for our project. At the beginning, after reading the papers, we've noticed the importance of two functions: 'Latency' and 'Is_Compact' because of the facts that:

1. Latency function is the base for all other functions. In other words, results of Latency are used in every function so it needs to be accurate and consistent.

2. One the key points of this method is 'Is_Compact' function because it is used for detecting the situation which out memory access pattern goes form one level of cache hierarchy to another.

Initially, we were planning to go through the implemented version of X-Ray and try to understand how the authors actually implemented these two, but as we've been told in during our presentation, the implemented code uses high-level techniques and complicated structure in code such that we couldn't gain any information from code. After all, we've started to implement from the scratch.

Latency

In order to measure the latency, we have to access consecutive elements in memory with a stride of one. This process is done in two parts: Initialization and measuring the Latency itself. There are several issues that we have to take into account:

1. We have to measure the time for exactly one memory operation.

2. To measure latency accurately, we have to repeat the memory access several times and then calculate the average time per access. It has to be done because the time taken by a single memory access is less than the accuracy of timer.

3. Unrolling the loop to avoid loop overhead.

Initialization: Initialization is coded as function called *initialize (S ,N)* which takes *S* as the stride and *N* as the Number of elements to access. In this function we initialize the array such that every element we access according to memory access pattern, contain the address of next element in memory access pattern. We do this to guarantee one pure memory access further in loop. One problem we've faced during implementation of this function was the problem with allocating memory. We've defined our array as *void** and we know that each element of array should hold the address of next elements according to access pattern. To keep our code portable and executable on every architecture, we allocate memory based on *sizeof(intptr_t).*

Latency Measurement: In function called *latency(X)*, we have to measure latency for exactly one pure memory access. For that propose, we access elements using *p=*(void **)p*. As we discussed before, we should unroll the loop to avoid loop overhead. According to experiments we've made, ten times unrolling is sufficient. To accurately measure the latency, we should repeat the memory access several times (in fact *X* times) and then calculate the average time per access. In the beginning, when we've implemented the first version of code, we've noticed that results are inaccurate and inconsistent. To overcome this issue, we tried going through assembly code and figuring out what instructions are executing. We've observed that without using the compiler optimizations, we cannot achieve single pure memory access. Therefore, we decide to compile our code using compile time optimizations. After that, we've found that in this case compiler detects the whole memory accesses as dead code and does not execute them. To solve the latest problem, we added a *printf* statement at the end of the *latency(X)* function which prints the value of last pointer *p*.

Capacity/Associativity

We will be making use of functions initialize() and latency() explained in the previous section. However, we made some changes to the functions so that it can work for changing value of Stride S. The basic idea behind calculating capacity and associativity is to call latency function multiple times and note down values of number of elements for which the sequence changes from compact to non-compact and once we know that we adjust the values of Stride S.

We start with sequence of length 2 and stride of length 8. We did these adjustments as the code for lesser values than this doesn't work as pointers used in the latency function instead of loop unrolling throw segmentation fault. After the initialization, we double the value of N until sequence is not compact. We note that value of N and then start doubling the value of S and compute smallest N for which sequence is not compact. We use binary search to find the value of N. Here is the pseudocode given in the paper which explains the same.

```
S ← 1;
N ← 1;
while (is compact (⟨m0, S,N⟩))
       N ← 2 × N;
repeat
       S ← 2 × S;
       Nold ← N;
       N ← minN' ∈ [1,Nold] : ¬is compact (⟨m0, S,N'⟩);
until (N = Nold);
A ← N - 1;
C ← s/2 × A;
```

Integer Add

To measure the latency of Integer Add operation, we have to execute an Integer Add and measure the time that it takes. However, we have to consider:

1. A minimum number of executions should be made to guarantee that total time is larger than accuracy of timing function in orders of magnitude.

2. Unrolling the loop to avoid loop overhead.

3. Avoid compiler to replace n Add operation with one optimized-equivalent Add operation.

In function *add_latency(X),* we execute a loop *X* times and we also unrolled the loop 16 times to cover the loop overhead and then compute the average latency per Add operation. To overcome the problem with compiler optimizations, we use a *switch(v)* where *v* is a *volatile int* and then in each case we execute one Add operation. As we expected, because *v* is a *volatile int* compiler cannot predict the behavior of code and therefore, it cannot replace Add operations with a single equivalent Add operation. Last issue that we've faced in implementation stage was the problem with dead-code elimination. To overcome this problem, we are reading the elements of our Add operation from two *volatile int* variables and at the end of function we also write the results back into a *volatile int* to avoid compiler detecting Add operations as dead-code.

Results

Latency

According to results presented by Yotov et al. the latency for the common architectures at that time was between 2-3 CPU cycles. After running our code on a few machines, we've noticed two issues:

1. Our results were usually around 3-4 CPU cycles.
2. Our results were varying around 10%.

First of all, we tried to double check our implementations and methods to make sure that everything is working as expected and we couldn't find any problem in our implementations. After that, we searched for technical specifications of architectures which we tested our code on (they are all 1-2 generations after the generation of architectures at the time that original paper was published) and we noticed that L1 cache latency for recent architectures is expected to be between three to five CPU cycles. Besides that, as we know, nowadays processors are using various methods of frequency stepping (DVFS, …) to meet their energy efficiency goals and their frequencies are dynamically adjusting. This adjustments in frequencies, can well explain 10% variation in our results.

Integer Add

Results for this part are pretty accurate and they are equal to one CPU cycles, as expected.

Computer	Processor	Latency	Integer Add
Macbook Pro Retina Laptop	Intel Core i7(Haswell) 2GHz	1.673800e-03	4.228125e-04
Atlantic Capsl	Intel Xeon 3.00GHz	3.357500e-03	4.188125e-04
Mills HPC	AMD Opteron Processor 6234	2.659900e-03	3.638750e-04

Conclusion

We were able measure the hardware parameter and compare our results with the actual values. We also evaluated our results by running them on various platforms to check for consistency of results.