Understanding the cache features in Intel Core Processors

A Project Report Submitted
in Partial Fulfillment of Requirements
for the Degree of

Bachelor of Technology

by
T Ashok Kumar (100050083)
Saket Tiwari (100050075)
Pratik Kumar (100100018)

under the guidance of
Prof. Bernard Menezes

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076, India

Abstract

An attacker can extract sensitive information of an algorithmic implementation using other channel leaks. One such system is the implementation of the AES algorithm in which the AES lookup tables are held in cache for computational purposes. With the advent of modern multi core processors we see many advanced techniques are being used in the area of prefetching and non-inclusivity, making it harder to execute these cache based side channel attacks. The prefetching and non-inclusivity mechanisms seem to be interfering with the attack logic and therefore there is a need to understand these mechanisms. Existing papers do not delve into these techniques. We have done the same through a series of experiments. In this report we go on to experimentally verify our hypothesized prefetching mechanism. We have exhaustively experimented to determine the existence of prefetching. We have conducted these experiments for L1 data cache. To determine the type and extent of prefetching in use we propose an access time based approach. We have also experimented to verify non-inclusivity of L1 and L2 cache levels. We exploit the fact that access time increases in case of cache miss at different levels in the memory hierarchy. We have used the chrono library to measure access times. To keep results consistent we have timed multiple instances of cache misses and cache hits iteratively. This has helped us keep the timing overhead insignificant in comparison to overall access times.

Acknowledgements

We would like to express our sincere gratitude to Prof. Bernard Menezes for his constant motivation, useful suggestions and words of wisdom. He has been our primary source of guidance during our entire B Tech Project. We would also like to thank Bholanath Roy for extending his support towards the project. We would also like to extend our gratitude towards all those people who have helped me in direct or indirect ways towards the partial accomplishment of this project.

Honor Code

We certify that we have properly cited any material taken from other sources and have obtained permission for any copyrighted material included in this report. We take full responsibility for any code submitted as part of this project and the contents of this report.

T Ashok Kumar (100050083)

Saket Tiwari (100050075)

Pratik Kumar (100100018)

Certificate

It is certified that the B. Tech. project "Cache Based Side Channel Attacks" has been done by students: T Ashok Kumar (100050083), Saket Tiwari (100050075) and Pratik Kumar (100100018) under my supervision. This report has been submitted towards partial fulfillment of B. Tech. degree requirements.

Bernard Menezes
Faculty Supervisor
Professor, Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai-400076, Maharashtra, India

Contents

Abstract i

Acknowledgements ii

Honor Code iii

Certificate iv

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Problem Statement . . . 1
  - 1.1.1 Motivation . . . 1
  - 1.1.2 Goals . . . 1
  - 1.1.3 Approach . . . 1
- 1.2 Side Channel Attack . . . 2
- 1.3 Organization of the Report . . . 2

2 Random vs Sequential Access 3
- 2.1 Preliminaries . . . 3
  - 2.1.1 Random Number Generation . . . 3
  - 2.1.2 Memory Hierarchy . . . 4
- 2.2 Approach . . . 4
  - 2.2.1 Sequential Access . . . 5
  - 2.2.2 Random Access(Type 1) . . . 5
  - 2.2.3 Random Access(Type 2) . . . 6
  - 2.2.4 Reverse Access . . . 6
- 2.3 Code . . . 7
- 2.4 Results . . . 8
- 2.5 Inference . . . 13

3 Extent of Prefetching **14**
3.1 Preliminaries . . . 14
3.1.1 Noops . . . 14
3.2 Approach . . . 14
3.3 Algorithm and Code . . . 16
3.3.1 Algorithm . . . 16
3.3.2 Code . . . 16
3.4 Results/Observations . . . 17
3.5 Inference . . . 18

4 Inclusion Experiment **19**
4.1 Preliminaries . . . 19
4.1.1 LRU Policy . . . 19
4.1.2 Cache Associativity . . . 19
4.1.3 Cache Sets . . . 20
4.2 Approach . . . 20
4.3 Code . . . 23
4.4 Results . . . 23

5 Conclusion **24**

6 Future Work **25**

A Miscellaneous Experiments **i**
A.1 Experiment to verify Noops . . . i
A.1.1 Aim . . . i
A.1.2 Method/Algorithm . . . ii
A.1.3 Results/Observations . . . ii
A.1.4 Inference . . . ii
A.2 Experiment to verify Noops . . . iii
A.2.1 Aim . . . iii
A.2.2 Method/Algorithm . . . iii
A.2.3 Results/Observations . . . iii
A.3 Timing the accesses using chrono timer . . . iv
A.3.1 Time Library . . . iv
A.3.2 Example Code . . . iv
A.4 Serializing the Instructions . . . v
A.4.1 lfence and mfence . . . v
A.4.2 Example Usage . . . v

List of Figures

2.1 Comparing the access times of sequential and random access on core i3 9
2.2 Comparing the access times of sequential and random access on core i5 10
2.3 Comparing the access time of forward/backward sequential access and random access on core i3 11
2.4 Comparing the access time of forward/backward sequential access and random access on core i3 12

3.1 Flowchart of the algorithm 16
3.2 Graph of $(T_1 - T_2)$ vs Stride Length for Core i3 17
3.3 Graph of $(T_1 - T_2)$ vs Stride Length for Core i5 18

4.1 Current snapshot of the cache levels 21
4.2 Current snapshot of the cache levels 22
4.3 Current snapshot of the cache levels 22

A.1 Flowchart of the algorithm ii
A.2 Flowchart of the algorithm iii

List of Tables

2.1 Results obtained on Core i3 for different value of n 9
2.2 Results obtained on Core i5 for different value of n 10
2.3 Comparing forward sequential access and reverse sequential on core i3 11
2.4 Comparing forward sequential access and reverse sequential on core i5 12

Chapter 1

Introduction

1.1 Problem Statement

1.1.1 Motivation

With the advent of modern multi-core processors we see many advanced techniques being used in the area of cache prefetching[4]. Existing research papers in this field do not delve into the prefetching techniques being used in Intel Core Processors. Implementation of Side Channel Attacks in Intel Core Processors has not been achieved so far. The findings of this report might help in facilitating such an attack. An attack of this nature on other less advanced processors has already been implemented[1].

1.1.2 Goals

- To verify the presence of prefetching in L1 cache of Intel Core Processors (Core i3 and Core i5).
- To study the extent of prefetching in L1 cache of Intel Core Processors (Core i3 and Core i5). Here, the word extent means the number of memory blocks being prefetched into the L1 cache.
- To verify the non-inclusiveness of L1 and L2 cache levels.

1.1.3 Approach

The experiments are based on the concept that each L1 cache miss leads to an increase in access time for the memory block[8]. If we are able to show that there are no L1 cache misses happening, even without the block being accessed earlier, then we can ascertain the presence of prefetching. To

measure the access times we have used chrono[17] c++ library. It gives us results in nanoseconds, but it has a some overhead which cannot be determined. To overcome this overhead we iterate over the accesses sufficient number of times and ensure that either the cache hit or miss takes place repeatedly. All time measurements of this nature can waver a lot depending on the processes being run in the background. To mitigate this we have tried our experiments when there aren't any of these high cache consuming processes running. We also present data from multiple runs of the experiment and their means and variances as we make inferences regarding the time taken for access and consequently the cache misses or hits happening. It is also important to note that we have compiled with 0 optimization[9] in all our experiments. This has been done to ensure that the nature of the code being run is not changed in context of our experiments.

1.2 Side Channel Attack

Side Channel attacks exploit data-dependent characteristics of the implementation of the cryptographic algorithm rather than the mathematical properties of the algorithm. Researchers have worked on various side channel attacks based on cache collisions, cache timings, cache access pattern, power monitoring, packet delivery timing, packet sizes, etc[16]. These exploit the time taken to access a memory block as an indicator to the location of the memory block in the memory hierarchy.

1.3 Organization of the Report

The rest of the report is organized as follows:
Chapter 2 deals with the experiment that compares random access and sequential access which is used to determine the presence of prefetching in L1 cache.
Chapter 3 deals with the experiment that compares access time with varying strides to determine the extent of prefetching in L1 Cache.
Chapter 4 deals with the experiment on Cache Inclusion Experiment.
The report ends with the Conclusion and Future Work.
Appendix contains description of some more experiments that we conducted to aid the main experiments and the details of the timer utility that we have used.

Chapter 2

Random vs Sequential Access

To understand the prefetching in L1 Data Cache by comparing the access time of Sequential Access and Random Access.

2.1 Preliminaries

In this section we have explained the preliminaries of Group Theory. The Experiments covered in this chapter largely deal with random accesses. One of the types of random accesses employs use of group generators in creating a pseudo random access pattern.

2.1.1 Random Number Generation

In mathematics, a **group** is a set of elements together with an operation that combines any two of its elements to form a third element satisfying four conditions called the group axioms, namely closure, associativity, identity and invertibility. A **finite group** is a mathematical group with a finite number of elements[14].

Given a finite group G with modulo multiplication as the operator, an element $g \neq I$ (I is the group identity of G) is said to be a generator of $< G, * >$ if $g^L = I$ (L is the size of the group G) and $\not\exists\ k$ such that $0 < k < L$ and $g^k = I$. There can be zero or more possible generator(s) for such a group[14].

In this chapter, for a given value of n, we need to generate a sequence of numbers which is a permutation of the set $S = \{x | x \in [1, n]\}$. The common random number generation methods take up a lot of CPU time and cache. This will interfere with the execution of our experiment and adds a lot of noise to the results. Also generating a random permutation beforehand, stor-

ing it in a file and accessing the file for the random numbers takes up the cache. So these two ideas are ruled out. But if n is prime, then the set S is a finite group (modulo n) with modulo multiplication as the group operator that can be generated by a single element $g \in S$. The sequence generated by g can be considered to be a pseudo random sequence.

For example, if $n = 7$, we have $S = 1, 2, 3, 4, 5, 6$. This can be generated by 3. The generating set of 3 is $3, 3^2 = 2, 3^3 = 6, 3^4 = 4, 3^5 = 5, 3^6 = 1$. For $n = 509$ and $n = 37$, 2 is a generator.

2.1.2 Memory Hierarchy

The memory hierarchy of all multi-core processors we use is comprised of L1 data cache (32 KB), L2 cache (256 KB), L3 cache (3 MB)[5] and then comes the RAM which varies in size across different systems. As the size of each of these layers increases so does the access time[13]. **Cache Associativity**: CPU cache associativity specifies how cached data is associated with locations in main memory.
Cache Hits And Misses: In the memory hierarchy, when the processor reads or writes to a location in main memory, it first checks for a corresponding entry in the L1 cache. If the requested block of memory is present L1 cache then we have a hit else we have a miss. These misses result in a time penalty on accessing the location of memory. Similarly if there is a miss in L2 cache additional penalty is incurred upon the memory location access statement. This penalty increases as we go down the memory hierarchy. Thus, one of the ways to ascertain a hit or a miss is to measure the time taken to access the memory location [8].

Prefetching in cache: This is the name given to the phenomena where in when accessing one block of cache the CPU "guesses" the next cache block that will require fetching and fetches it before hand into L1 cache, thereby avoiding a cache miss[4].

2.2 Approach

To verify the existence of prefetching we have conducted a series of experiments. The main theme across these experiments is the nature or order in which the cache blocks are being accessed. We worked with the hypothesis that the CPU should be able to detect a sequential access of cache blocks and consequently pre-fetch the successive blocks. A random access of the cache blocks should "confuse" the CPU not allowing it to "guess" the blocks to be

prefetched and ensure that the cache hits due to prefetching are minimized or at least reduced by a certain degree. This kind of reduction in cache hits (consequently an increase in cache misses) should be reflected in the increase in overall access time.

2.2.1 Sequential Access

- We access fixed number (say n) of Cache Blocks sequentially. We do it by declaring an array. This array represents the blocks of memory which can be accessed or written into. We access different blocks of the memory represented by the array by simply accessing them in strides of the block size. By accessing in strides we mean accessing memory location $X, X+B, X+2B, \cdots$, where B is the block size. This translates to accessing indexes $j, j+(B/(sizeof(int))), j+2*(B/(sizeof(int))), \cdots$ of the aforementioned array.

- We iterate over various values of n. We access the array indexes $i, i+1, \cdots, i+n-1$. This ensures that we access n cache blocks, which is the size of sample space across different experiments. Across the varying types of accesses the sample space needs to be identical to ensure uniformity of the experiments.

- We tabulate the time taken to complete access of n blocks of cache in a sequential order (T1).

2.2.2 Random Access(Type 1)

- We access n cache blocks. We again declare an Array to represent the blocks. These blocks can be accessed in any order just by changing the index of the array element being accessed. Thus, an Array works to our advantage.

- We access the same blocks $i, i+1, \cdots, i+n-1$ in a random order. We choose a window of n blocks and then access inside this memory window in a random order. Note that accessing memory blocks again translates to accessing indexes of arrays.

- To access the blocks in a random manner we generate block numbers using a generator for the group($\{1, 2, \cdots, n-1\}$,$\star$). The generator works as follows - we find numbers $g^k \, mod \, n \;\; \forall \, k \;\; \in \{1, 2, \cdots, n-1\}$. All the numbers 0 to n-1 of the set are generated exactly once upon

choosing prime generator, this gives us a random sequence of numbers representing the blocks to be accessed.

- We tabulate the time taken to complete access of n blocks of cache in a random order of Type 1 (T1).

2.2.3 Random Access(Type 2)

- We access the same number(n) of cache blocks. This again maintains the uniformity of the size of sample space.

- In this case, we don't access the memory blocks within a window of n contiguous blocks. Instead the chosen window is of the size of L1 data cache i.e. 32 KB[13]. This is done to spread out the accesses across all of L1 data cache which in turn should reduce the number of accesses happening to blocks very nearby (adjacent mostly) blocks, thereby reducing the chances of a cache hit on a prefetched block. We access n blocks in random order across the entire L1 data cache.

- To do this, we generate a pseudo-random sequence. The sequence generation works as follows - we start with $x = 1$ and then generate the next using $(x^2 + 1)modm$ (m is the number of blocks in L1 data cache) and so on till we get a sequence of n numbers. We do n such iterations to ensure access of n blocks across all of L1 data cache. The sequence produced in this manner need not always be distinct but the probability of them being distinct is very high if m is prime. This idea has been developed from the Pollard Rho algorithm[18].

2.2.4 Reverse Access

- We again access n blocks of memory.

- In this case we access the blocks of memory in the reverse order of sequential access. We access memory blocks indexed as $i + n - 1, i + n - 2, \cdots, i$. Similar to sequential access, here we declare an array and take strides of Block size $B/sizeof(int)$, to ensure we skip one block at each step, but in the decreasing order of indexes.

- We tabulate the time taken for Reverse Access of cache blocks (T4). This is done to see the effect the direction of jumps has on the prefetching by cache.

2.3 Code

Code for Sequential Access(T_1)

```
t1 = high_resolution_clock::now();
for(i=0; i<LARGE_N; i++)
{
  seed=1;
  j=0;
  while(j<mod-2)
  {
    b = seed*sd;
    c = seed+1;
    d = seed*seed + 1;
    seed = c;
    seed = seed%mod;
    a=arr[seed*16*stride];
    clflush(&arr[seed*16*stride]);
    j++;
  }
}
t2 = high_resolution_clock::now();
time_span = duration_cast<chrono::nanoseconds>(t2 - t1);
```

Code for Random Access of Type 1(T_2)

```
t1 = high_resolution_clock::now();
for(i=0; i<LARGE_N; i++)
{
  seed=1;
  j=0;
  while(j<mod-2)
  {
    b = seed*sd;
    c = seed+1;
    d = seed*seed + 1;
    seed = b;
    seed = seed%mod;
    a=arr[seed*16*stride];
    clflush(&arr[seed*16*stride]);
    j++;
  }
}
t2 = high_resolution_clock::now();
time_span = duration_cast<chrono::nanoseconds>(t2 - t1);
```

Code for Random Access of Type 2(T_3)

```
t1 = high_resolution_clock::now();
```

```
for(i=0; i<LARGE_N; i++)
{
  seed=1;
  j=0;
  while(j<mod-2)
  {
    b = seed*sd;
    c = seed+1;
    d = seed*seed + 1;
    seed = d;
    seed = seed%mod;
    a=arr[seed*16*stride];
    clflush(&arr[seed*16*stride]);
    j++;
  }
}
t2 = high_resolution_clock::now();
time_span = duration_cast<chrono::nanoseconds>(t2 - t1);
```

Code for Reverse Sequential Access(T_4)

```
t1 = high_resolution_clock::now();
for(i=0; i<LARGE_N; i++)
{
  seed=1;
  j=0;
  while(j<mod-2)
  {
    b = seed*sd;
    c = seed-1;
    d = seed*seed + 1;
    seed = d;
    seed = seed%mod;
    a=arr[seed*16*stride];
    clflush(&arr[seed*16*stride]);
    j++;
  }
}
t2 = high_resolution_clock::now();
time_span = duration_cast<chrono::nanoseconds>(t2 - t1);
```

2.4 Results

We tabulated the results of our experiment below and plotted them.
T_1 = Time Taken for Sequential Access of n blocks
T_2 = Time Taken for Random Access of Type 1 of n blocks

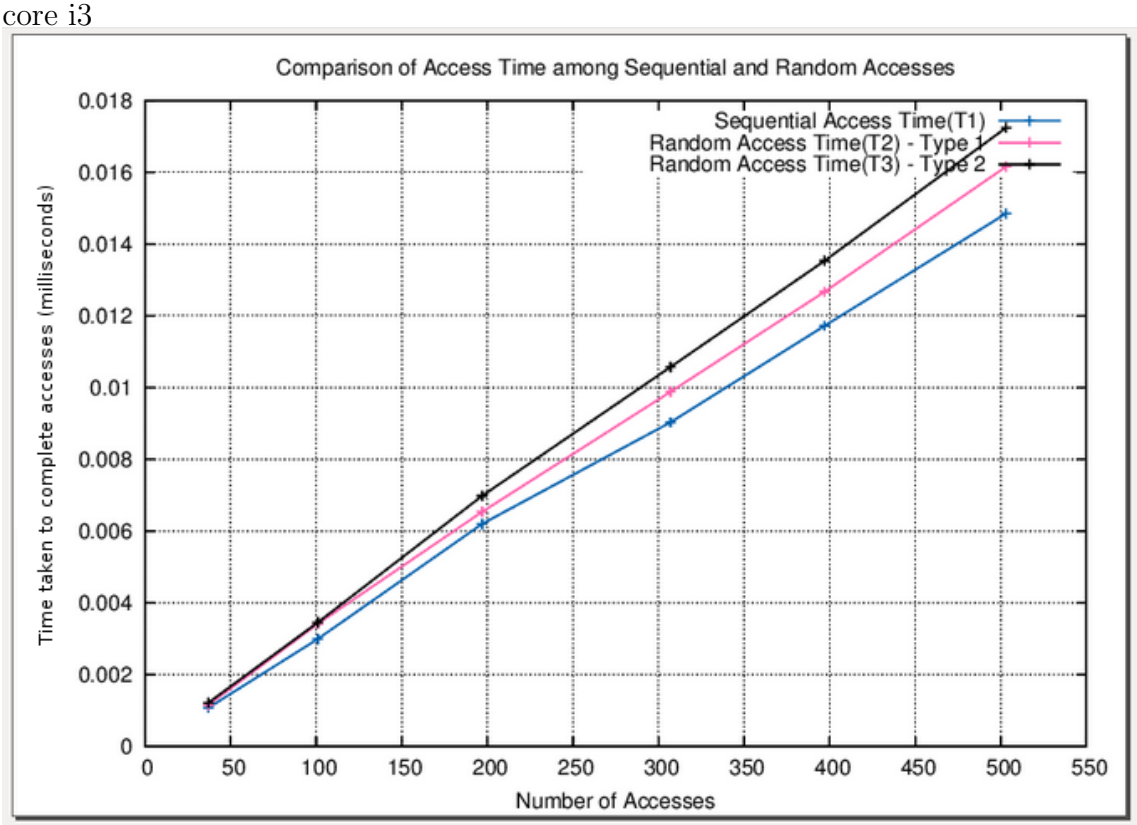
T_3 = Time Taken for Random Access of Type 2 of n blocks
T_4 = Time Taken for Sequential Access in Reverse Order of n blocks

The access times T_1, T_2, T_3 are observed for various values of n on core i3 and are tabulated in Table 2.1. The graph corresponding to this table is plotted in Graph 2.1.

Table 2.1: Results obtained on Core i3 for different value of n

n	**Mean**(T_1)	**Mean**(T_2)	**Mean**(T_3)
37	0.001069	0.001148	0.001211
101	0.002991	0.003413	0.003453
197	0.006195	0.006534	0.006978
307	0.009030	0.009880	0.010568
397	0.011717	0.012664	0.013527
503	0.014846	0.016153	0.017238

Figure 2.1: Comparing the access times of sequential and random access on core i3

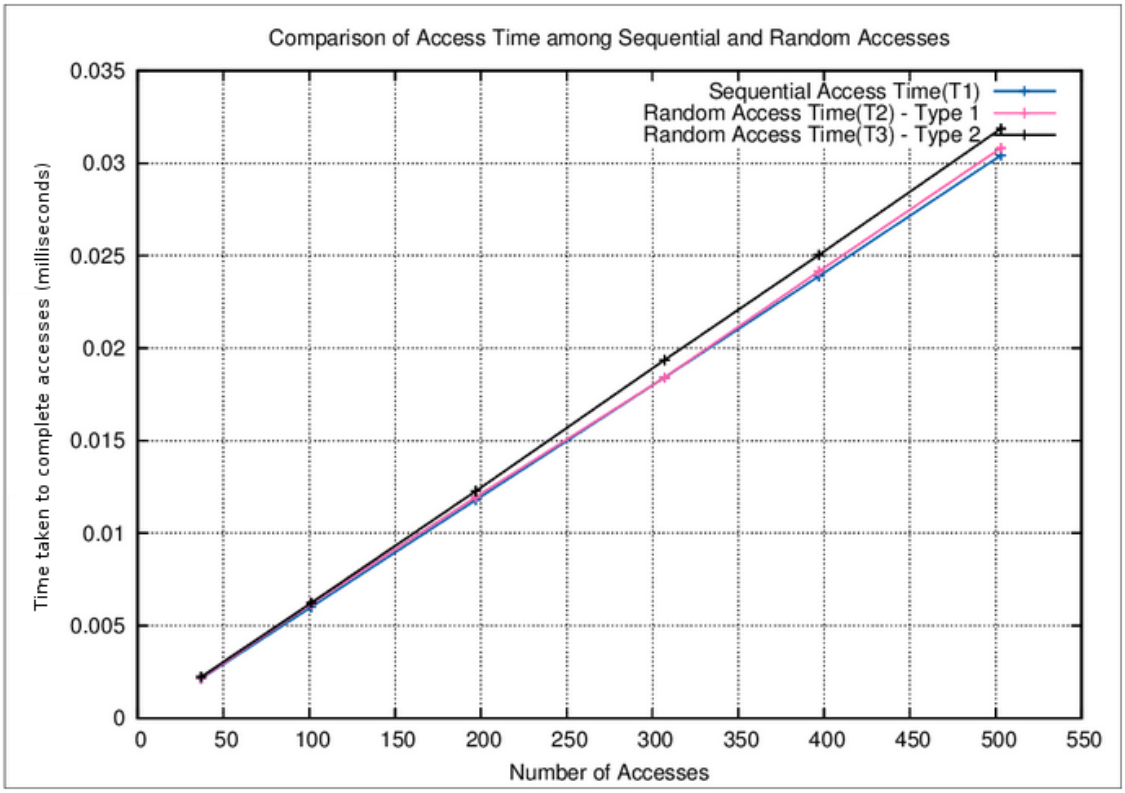


The times T_1, T_2, T_3 are observed for various values of n on core i5 and are tabulated in Table 2.2. The graph corresponding to this table is plotted in Graph 2.2.

Table 2.2: Results obtained on Core i5 for different value of n

n	**Mean**(T_1)	**Mean**(T_2)	**Mean**(T_3)
37	0.002134	0.002155	0.002229
101	0.006000	0.006177	0.006225
197	0.011791	0.011952	0.012272
307	0.018414	0.018414	0.019355
397	0.023889	0.024152	0.025041
503	0.030416	0.030814	0.031860

Figure 2.2: Comparing the access times of sequential and random access on core i5



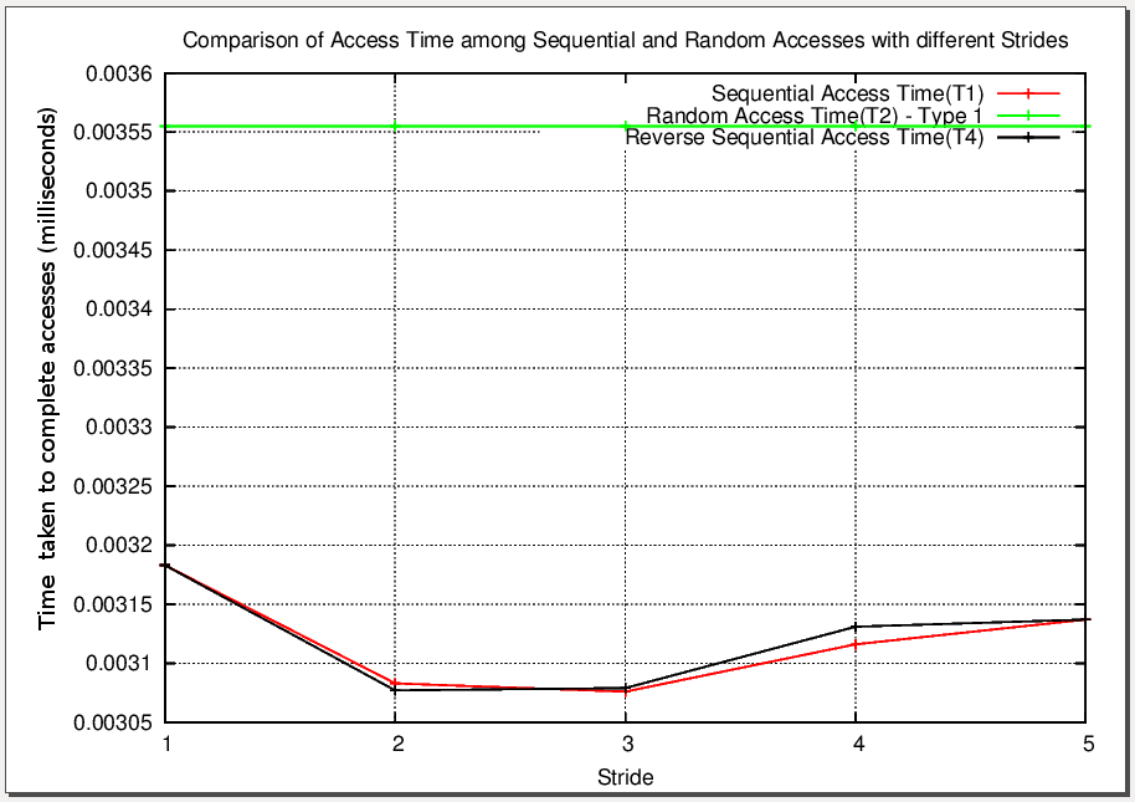
We then compared the time taken to access sequentially in forward direction and backward direction varying the stride length fixing the value of n to

be 101. The times T_1, T_2, T_4 are observed for various stride lengths on core i3 and are tabulated in Table 2.3. The graph corresponding to this table is plotted in Graph 2.3.

Table 2.3: Comparing forward sequential access and reverse sequential on core i3

Stride(in multiples of cache block size)	**Mean(T_1)**	**Mean(T_2)**	**Mean(T_4)**
1	0.003183	0.003555	0.003183
2	0.003083	0.003555	0.003077
3	0.003076	0.003555	0.003079
4	0.003116	0.003555	0.003131
5	0.003137	0.003555	0.003137

Figure 2.3: Comparing the access time of forward/backward sequential access and random access on core i3



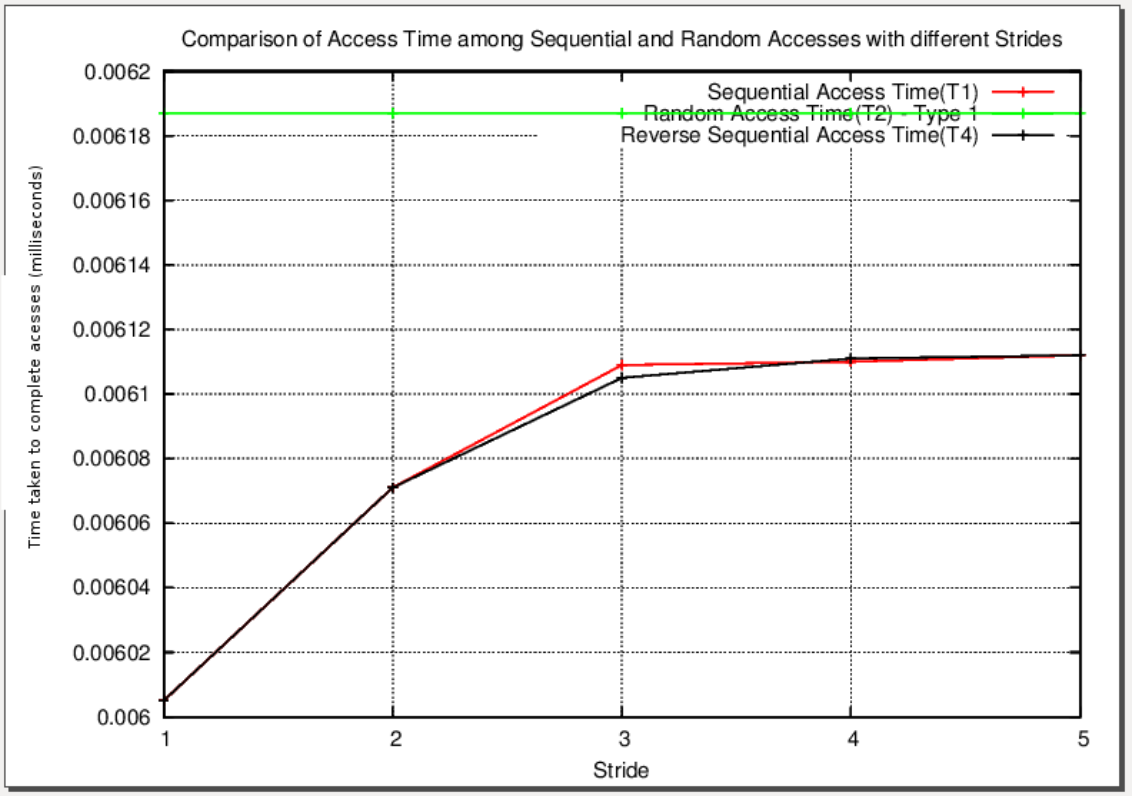
The times T_1, T_2, T_4 are observed for various stride lengths on core i5 and

are tabulated in Table 2.4. The graph corresponding to this table is plotted in Graph 2.4.

Table 2.4: Comparing forward sequential access and reverse sequential on core i5

Stride(in multiples of cache block size)	**Mean(T_1)**	**Mean(T_2)**	**Mean(T_4)**
1	0.006005	0.006187	0.006005
2	0.006071	0.006187	0.006071
3	0.006109	0.006187	0.006105
4	0.006110	0.006187	0.006111
5	0.006112	0.006187	0.006112

Figure 2.4: Comparing the access time of forward/backward sequential access and random access on core i3



2.5 Inference

- In the first experiment, we varied the value of n and compared the time taken for sequential access and random access. As we keep increasing the value of n the time difference between T_3, T_2 and T_1 goes on increasing, with $T_3 > T_2 > T_1$ always holding for all values of n.

- This result can be explained in the following way T_1 has highest amount of prefetching therefore there are lesser or almost no cache misses resulting in a low access time compared to the other two (See graph 1, 2). In case of T_2 and T_3 the nature of accesses is random and therefore there is very little chance that the processor is able to "guess" the cache block to be prefetched. We also observe that T_3 is always larger than T_2. This is because, in case of Random Access of type 1, we are confined to a given set of elements. This can lead to a fewer cache misses due to prefetching. But in the case of Random Access of type 1, the accessed elements are not confined resulting in a very less chance of prefetching being effective.

- In the second experiment. we have varied the stride length to compare the access times in forward-sequential, reverse-sequential and random access.

- Here we again observe that for a given n (101) as we increase the stride length both T_1 and T_4 behave the similar way and random access always results in more cache misses and consequently a larger value of T_3 (See graph 3, 4). Thus, we can say that prefetching happens irrespective of the stride length and the accesses are being done in the reverse-sequential order.

Chapter 3

Extent of Prefetching

To verify the extent of prefetching in L1 cache. By the extent of prefetching we mean the number of blocks being pre-fetched into L1 cache upon accessing a memory block in L1 cache.

3.1 Preliminaries

A basic know how of the memory hierarchy and cache hits and misses is required for this section. All these preliminaries have already been discussed in detail in the previous chapter. In addition to these we have used the notation *Noops* through out our experiment, which we have gone on to explain in this section.

3.1.1 Noops

Throughout our experiment we used noops to represent empty function calls. These are not noops in their computer architechture sense but rather a call to an empty function which allows for the prefetcher to prefetch cache blocks into L1 cache. Here Noops are only used to overcome the effects of pipelining and serve as an empty statement which leads to the prefetcher having enough time to prefetch.

3.2 Approach

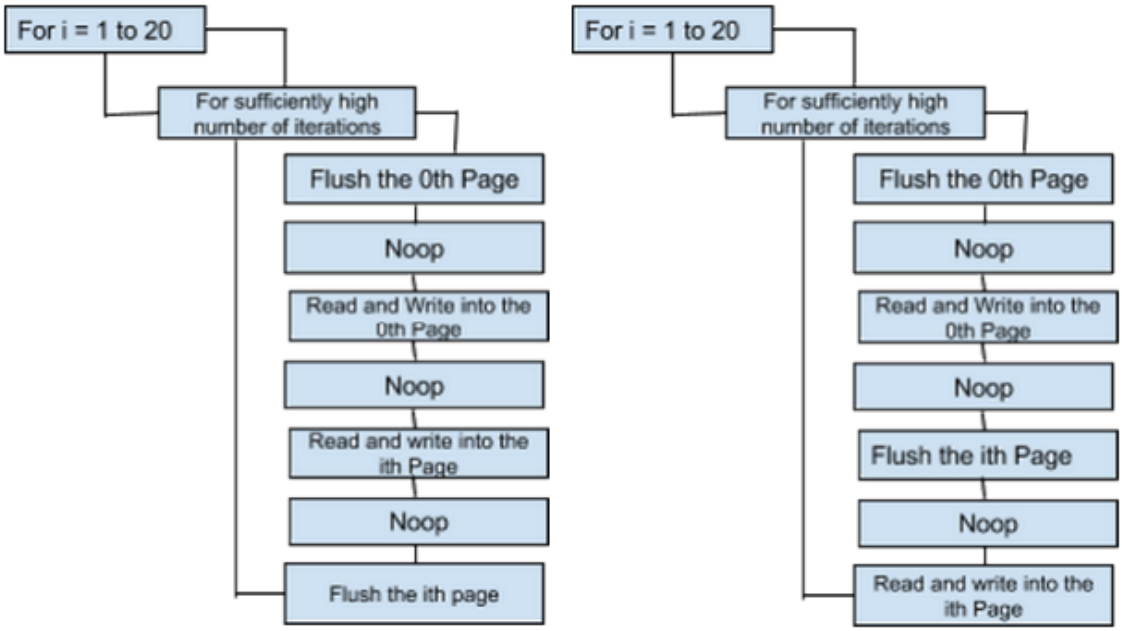
We run two loops for a given value of i (the stride length). The stride length in this case represents the distance between memory locations being accessed. In the first loop (Loop 1) we ensure that the i^{th} memory block is not present in all the cache layers by making a call to clflush(i) [7] and

then we access the i^{th} page. Then we record the time taken for the execution of this loop. In the second loop (Loop 2) we flush the i^{th} page after doing the read and write operation on it. In Loop 2 if the i^{th} page had been prefetched into L1 cache (As hypothesized in Chapter 2) the time taken to run through Loop 1 would be more than that of Loop 2 due to cache hit at block numbered i in the case of latter and a cache miss at block numbered i in case of the former. This is so because in Loop 1 we always flush the i^{th} page before we access it, leading to a cache miss thereby resulting in a higher run time for Loop 1. In Loop 2 the access is done after reading and writing the i^{th} page and the flush is done only after that, to keep the number and nature of instructions across the loops identical. In Loop 2 if the i^{th} page had been prefetched into L1 cache it would lead to a cache hit resulting in a smaller run time for Loop 2 as compared to Loop 1. The flush statement also serves the purpose of ensuring that the i^{th} page is not in L1 cache over subsequent iterations for a given value of i. In this experiment the noops have been added to ensure a gap in between memory operations - flush and read/write. This gap helps reduce the effects of pipelining, the utility of noops has been discussed in detail in the experiments described in the appendix.

3.3 Algorithm and Code

3.3.1 Algorithm

Figure 3.1: Flowchart of the algorithm



3.3.2 Code

```
for (int i = 1; i < 20; i++)
{
  // Loop 1 starts here
  for(j = 0; j < 10000000; j++)
  {
    clflush(&arr[0]);
    noop();
    a += arr[0];
    noop();
    clflush(&arr[i*16]);
    a -= arr[i*16];
  }
  // End of Loop 1

  // Loop 2 start Here
  for(j = 0; j < 10000000; j++)
  {
```

```
18      clflush(&arr[0]);
19      noop();
20      a += arr[0];
21      noop();
22      a -= arr[i*16];
23      noop();
24      clflush(&arr[i*16]);
25    }
26    // End of Loop 2
27  }
```

3.4 Results/Observations

T_1 - time for Loop 1
T_2 - time for Loop 2 Time difference = $(T_2 - T_1)$.

Figure 3.2: Graph of $(T_1 - T_2)$ vs Stride Length for Core i3

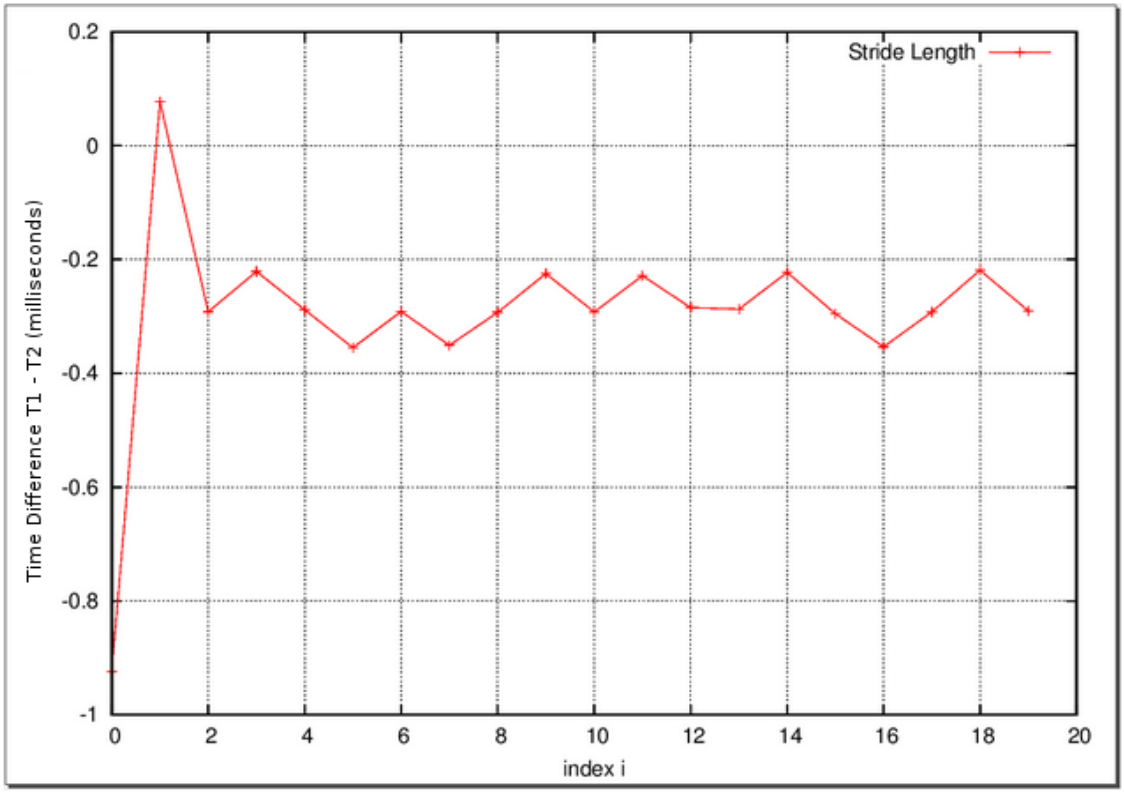
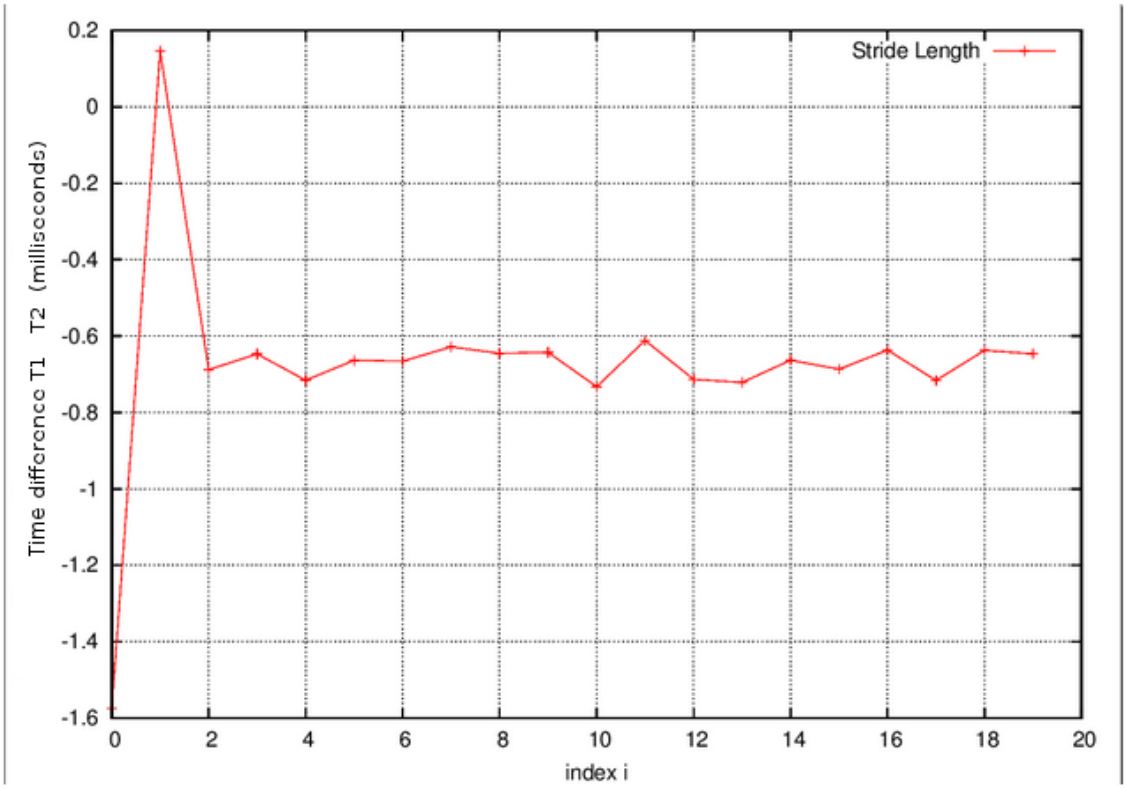


Figure 3.3: Graph of $(T_1 - T_2)$ vs Stride Length for Core i5



3.5 Inference

- The graph shows that the value of $T_2 - T_1$ is positive for $i = 1$ only.
- This suggests that the time taken for Loop 1 is lesser than that of Loop 2 only in the case where we access the immediately succeeding page.
- From the structure of our experiment we can infer that only this page (i.e. the immediately succeeding page) is being prefetched into L1 cache.
- We observe that the values of $T2 - T1$ are negative, this maybe due to the flush statement in Loop 1. It flushes a block which is present in the cache (all layers) in Loop 1 where as it flushes an a block that is in cache in Loop 2.

Chapter 4

Inclusion Experiment

To prove that L1 cache and L2 cache are non-inclusive

4.1 Preliminaries

4.1.1 LRU Policy

A good approximation to the optimal page replacement algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called LRU (Least Recently Used) paging.

Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even via hardware (assuming that such hardware could be built)[19].

4.1.2 Cache Associativity

The replacement policy decides where in the cache a copy of a particular entry of main memory will go. If the replacement policy is free to choose any entry in the cache to hold the copy, the cache is called fully associative. At

the other extreme, if each entry in main memory can go in just one place in the cache, the cache is direct mapped. Many caches implement a compromise in which each entry in main memory can go to any one of N places in the cache, and are described as N-way set associative.

4.1.3 Cache Sets

The L1 cache maybe physically addressed or virtually addressed. The address translation process translates a virtual address to physical address keeping the last twelve bits same (offset). Two cache blocks mapping to the same set of L1 cache must have a difference of integer multiple of 4KB (Since 4KB = 32KB/8, where 32 KB is the size of L1 cache and 8 is the associativity of L1 cache), which can be represented by 12 bits. Therefore, the last twelve bits of the physical/virtual address can be used to identify the blocks which map to the same set of L1 cache.
The L2 cache level is physically addressed. The two cache blocks mapping to the same set of L2 cache must have their last 15 bits same. This is because 256KB/8 = 32 KB (where 256 KB is the size of L2 cache and 8 is the associativty of L2 cache), which can be represented by 15 bits. As mentioned earlier the virtual and physical addresses have only the last 12 bits same. This requires us to translate the virtual address to physical address in order to find memory blocks mapped to the same set in L2 cache.

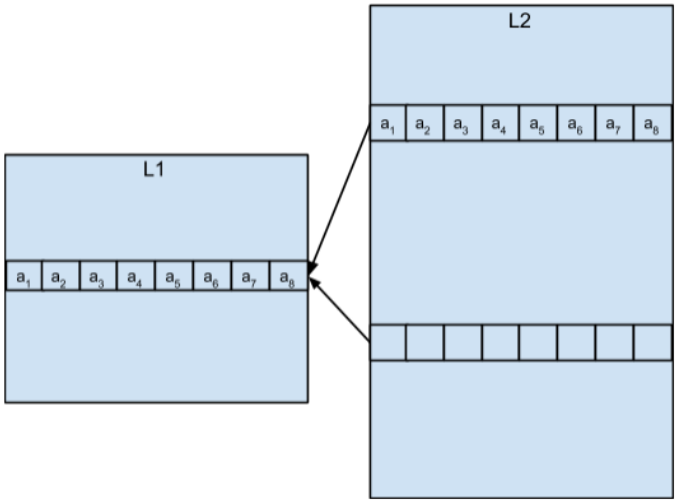
4.2 Approach

The general theme of the experiment is to show that there exists a cache block in L1 Cache and at the same time not in L2 Cache. We declare a sufficiently large array to access cache blocks. Since the cache blocks mapping to the same set of L1 cache are located at a distance of integer multiple of 4 KB (or 1024 integers), we iterate through all the cache blocks with a stride of 4 KB (therefore all such blocks mapped to the same set of L1 cache level), starting from beginning of the array. While doing so, we convert their virtual addresses to physical addresses and consider the blocks which have the same last 15 bits of physical address. All of these have the last twelve bits the same because they map to the same L1 cache set. We consider a total of 17 blocks of all these, such that 8 of them map to one set of L2 cache and the other 9 map to another set of L2 cache. Let us call the selected 8 blocks $b_1, b_2, \cdots b_8$ and the selected nine as $a_1, a_2 \cdots a_9$. This gives us $a_1, a_2, \cdots a_9$ and $b_1, b_2 \cdots b_8$ belonging to the same set of L1 cache and $a_1, a_2 \cdots a_9$ belonging to the one set of L2 cache and $b_1, b_2 \cdots b_8$ belonging to another set of L2 cache level.

We perform the following operations sequentially:

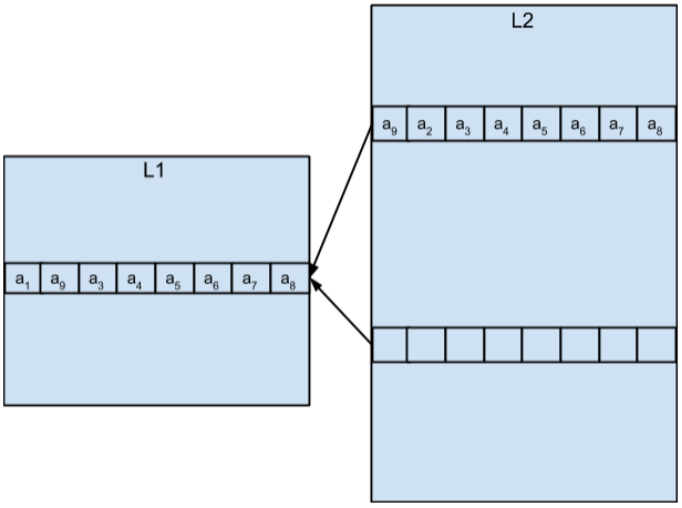
- We flush everything from all the cache levels that belongs to our array, using clflush()[7].
- We then follow the access pattern (all accesses are reads): $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$. This brings up $a_1, a_2, \cdots, a_8$ into all the cache levels. The current snapshot of L1 and L2 as as follows:

Figure 4.1: Current snapshot of the cache levels



- If we now access a_9, the LRU policy of L1 and L2 evicts a_1. Before accessing a_9 we access a_1. Since a_1 is already present in L1 cache, it does not result in L1 cache miss. But we can observe that a_1 is no more the Least Recently Used cache block. Instead a_2 becomes the Least Recently Used cache block.
- Now if we access a_9, the LRU policy of L2 evicts a_1 but LRU policy of L1 evicts a_2. The current snapshot of L1 and L2 should be as follows:

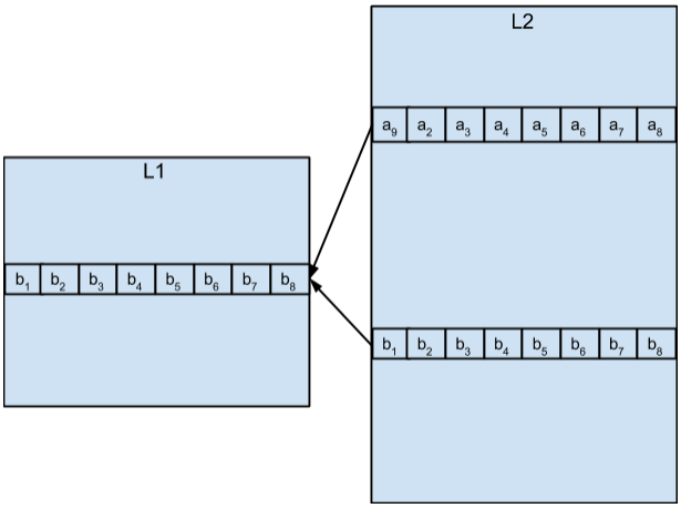
Figure 4.2: Current snapshot of the cache levels



We can observe that a_1 is present in L1 but not in L2 by measuring the access time. We measure the access time for a_1 (T1).

- We then access $b_1, b_2, \cdots, b_8$ to bring all b_i into L1 cache. The current snapshot of L1 and L2 as as follows:

Figure 4.3: Current snapshot of the cache levels



We now access a_1 which is not present in L1 and L2 cache levels. We measure the access time for a_1 now (T2).

If we can show that T1 is almost same as L1-access time and T2 is atleast as large as L3-access time, then we can prove that L1 and L2 are non-inclusive.

We could improve the precision of the timer by performing this operation on several sets of L1 cache. We then can time the operations cumulatively to minimize the error incured due to timer overhead.

4.3 Code

```
1    //access the elements a1, a2, .. a8
2    for(k=0; k<8; k++){
3      a += arr[fin1[k] ];
4    }
5    //access a1
6    a += arr[fin1[0] ];
7    //access a9
8    a += arr[fin1[8] ];
9    r1 = rdtsc_start();
10     //access a1
11     a += arr[fin1[0] ];
12   r2 = rdtsc_end();
13   T1 = (r2 - r1);
14
15   //access b1, b2, ... b8
16   for(k=0; k<8; k++){
17     a += arr[fin2[k] ];
18   }
19
20   r1 = rdtsc_start();
21     //access a1
22     a += arr[fin1[0] ];
23   r2 = rdtsc_end();
24   T2 += (r2 - r1);
```

4.4 Results

Results are not satisfactory yet. This is because of the following reasons:

- The replacement policy is not strict LRU. The hardware uses a learning algorithm for replacement.
- We do not have a sophisticated timer to find the exact time taken by an instruction. The timer::start() and timer::end() functions creates timer overhead, resulting in some inaccuracy.

Chapter 5

Conclusion

Based on the results of various experiments performed, we conclude the following.

- The time taken to access the memory in a sequential manner with a given stride is same for forward sequential access and reverse sequential access.
- The time taken to access the memory in sequential manner is less than the time taken to do the same in a pseudo-random manner is less since there are many cache misses in the latter case. This concludes that prefetching takes place.
- The Hardware prefetches only one page every time. The time taken to access pages which are more than a page away from the initial page is always high in case of sequential access with stride one, implying a cache miss at pages which are further away. A cache miss in turn implies the absence of the page in L1 cache and consequently the absence of prefetching in this case.This helps us conclude that only one page is being prefetched into L1 cache.
- The replacement policy used in the cache is not pure LRU.
- Also, if we assume that cache replacement policy is pure LRU, we can prove that the L1-cache and L2-cache are non-inclusive.
- Determining the time taken to execute a single instruction is difficult because of the overhead due to the function calls and out-of-order execution of statements.

Chapter 6

Future Work

There still are aspects or features of Cache in multi core processors that need to be determined, to have a complete understanding of the system. One of these features on which further investigation is required is the inclusivity or non-inclusivity of the caches at different layers. An experiment demonstrating the inclusive or non-inclusive nature of the layers of caches can be developed. Such a feature could affect a cache based side channel attack, and an attacker requires a complete understanding of it in order to engineer and execute an attack.
Another interesting development with time is a better Page Replacement Policies. Earlier, a pseudo-LRU policy was a lot common with regards to page replacement in different layers of cache. Currently, a more complex and detailed policy which is being used which has not been determined completely. This is of utmost importance in conducting the above experiment of cache inclusivity and therefore this experiment will serve as a base for the above experiments.
Another obscure feature is the policy used in prefetching. We know from our experiments that only one page is being fetched and that the hardware has a smart way of prefetching when it prefetches with varying strides.
There is a need for a better timing approach in these time sensitive experiments. The existing timer has a significant overhead when it comes to timing experiments having single or less memory access instructions just like in our case. A better timer would help to solve the above problem and will help in determining the above mentioned features
Once these features have been fully understood a cache based side channel attack on AES can be engineered and executed. Another possibility is that one or some of these features could be exploited to engineer such an attack.

Appendix A

Miscellaneous Experiments

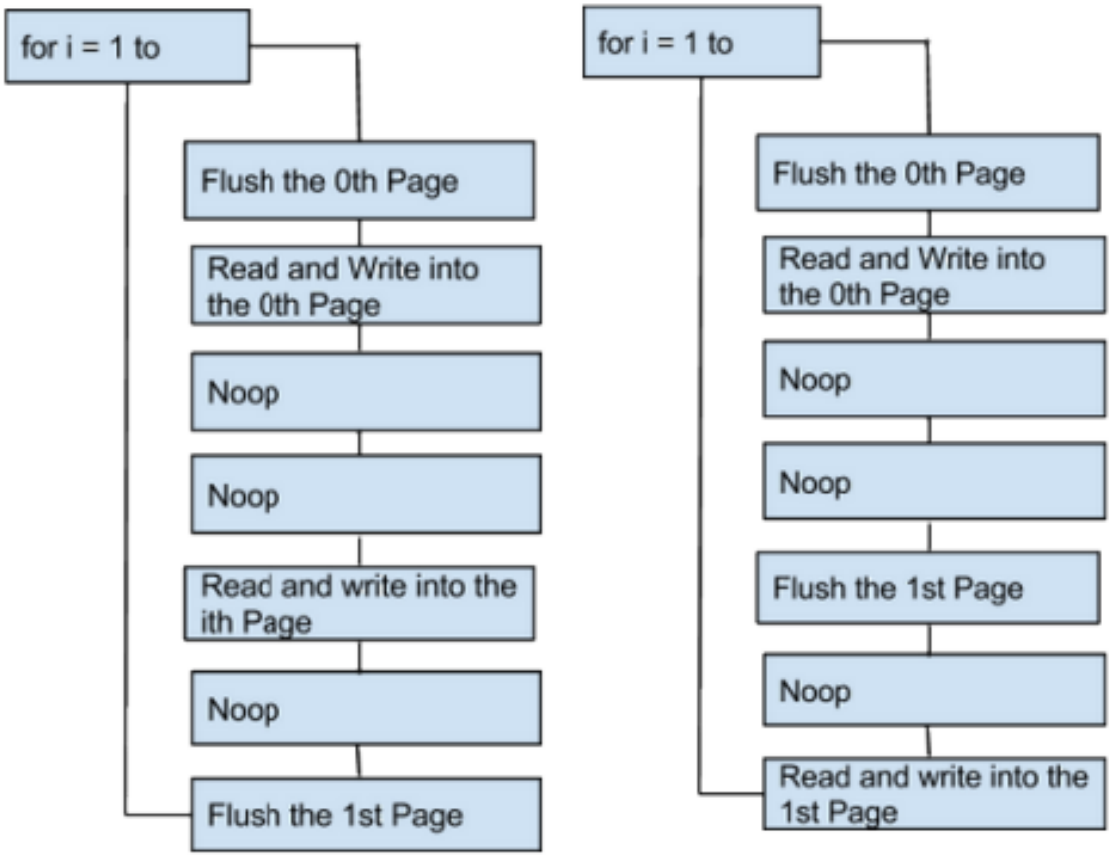
A.1 Experiment to verify Noops

A.1.1 Aim

In this experiment we change the position of noops to verify the hypothesis that noops give some time in between two memory operations flush and read/write. The noop reduces the effect of pipelining in between consecutive memory operations. If we let them be consecutive we must observe that the time taken for loop 2 is more than that for Loop1 giving a negative result. This is so because it would be inconsistent with the results in Chapter 3.

A.1.2 Method/Algorithm

Figure A.1: Flowchart of the algorithm



A.1.3 Results/Observations

T_1 - time for Loop 1 T_2 - time for Loop 2

CPU	**Mean of $T_1 - T_2$**	**Variance of $T_1 - T_2$**
Core i3	-0.143038	0.001221
Core i5	-0.382379	0.009228

A.1.4 Inference

This result reinforces our hypothesis that noops are required to give some time interval in between two memory operations.

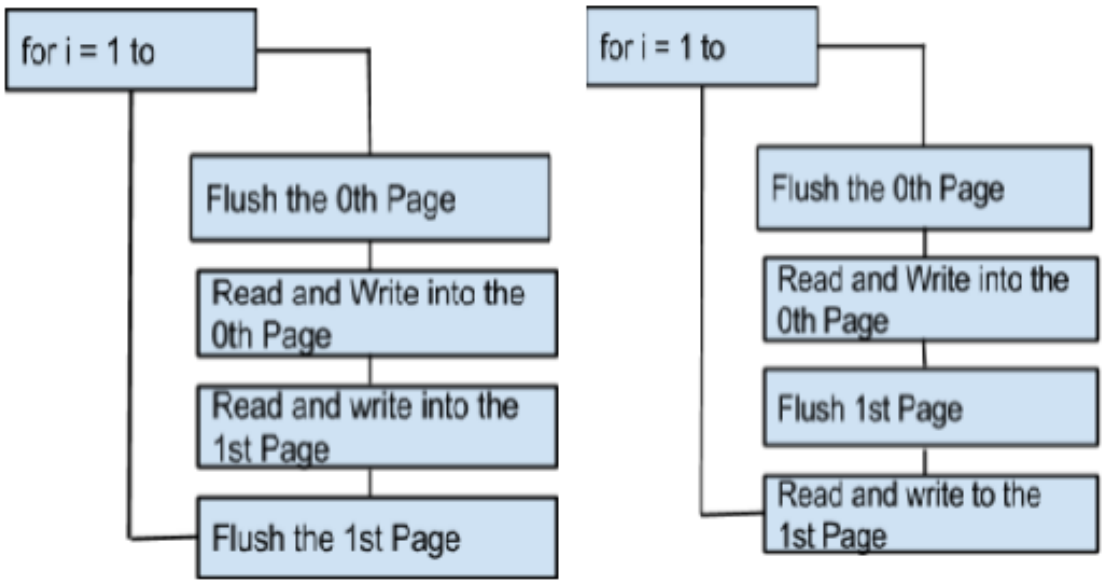
A.2 Experiment to verify Noops

A.2.1 Aim

In this experiment we remove the noops entirely. Again here we test our hypothesis that two consecutive memory operations should have noops in between them in order to overcome the effects of pipelining in case of consecutive memory operations. Upon the removal of noops from the loops we must observe negative result, contradictory to Chapter 3.

A.2.2 Method/Algorithm

Figure A.2: Flowchart of the algorithm



A.2.3 Results/Observations

T_1 - time taken for Loop 1 T_2 - time taken for Loop 2

CPU	**Mean of $T_1 - T_2$**	**Variance of $T_1 - T_2$**
Core i3	-0.741841	0.000927
Core i5	-0.248119	0.056614

Inference

This again reinforces our hypothesis. We get a higher T2 than a T1 implying that read and write statements take more time if there are no noops in between.

A.3 Timing the accesses using chrono timer

A.3.1 Time Library

chrono is the name of a header, but also of a sub-namespace: All the elements in this header (except for the common_type specializations) are not defined directly under the std namespace (like most of the standard library) but under the std::chrono namespace.

The elements in this header deal with time. This is done mainly by means of three concepts:

Durations
They measure time spans, like: one minute, two hours, or ten milliseconds.
In this library, they are represented with objects of the duration class template, that couples a count representation and a period precision (e.g., ten milliseconds has ten as count representation and milliseconds as period precision).

Time points
A reference to a specific point in time, like one's birthday, today's dawn, or when the next train passes.
In this library, objects of the time_point class template express this by using a duration relative to an epoch (which is a fixed point in time common to all time_point objects using the same clock).

Clocks
A framework that relates a time point to real physical time.
The library provides at least three clocks that provide means to express the current time as a time_point: system_clock, steady_clock and high_resolution_clock.

A.3.2 Example Code

```
1 #include <iostream>
2 #include <chrono>
```

```
using namespace std;
int main()
{
    cout << chrono::high_resolution_clock::period::den << endl;
    auto start_time = chrono::high_resolution_clock::now();
    int temp;
    for (int i = 0; i< 242000000; i++)
      temp += temp;
    auto end_time = chrono::high_resolution_clock::now();
    cout << chrono::duration_cast<chrono::seconds>(end_time -
   start_time).count() << ":";
    cout << chrono::duration_cast<chrono::microseconds>(end_time
    - start_time).count() << ":";
    cout << chrono::duration_cast<chrono::nanoseconds>(end_time
   - start_time).count() << ":";
    return 0;
}
```

A.4 Serializing the Instructions

While timing using the chrono library we tried to execute two timing instructions consecutively. This resulted in out of order execution with the second timing event giving a lower time. This led to the use of instrutions to serialize the execution of code.

A.4.1 lfence and mfence

The Intel architecture may execute instructions in parallel or out of order. Without serialisation, instructions surrounding the measured code segment may be executed within that segment. Intel recommends using the serialising instruction cpuid for that purpose. However, in virtualised environments the hypervisor emulates the cpuid instruction. This emulation takes significant time (over 1,000 cycles) and this time is not constant, reducing the granularity and the stability of the attack. The purpose of the mfence and lfence instructions is to serialise the instruction stream. The lfence instruction performs partial serialisation. It ensures that load instructions preceding it have completed before it is executed and that no instruction following it executes before the lfence instruction.The mfence instruction orders all memory access, fence instructions and the clflush instruction. It is not, however, ordered with respect to other instructions and is, therefore, not sufficient to ensure ordering[20].

A.4.2 Example Usage

```
asm volatile ("mfence\t\n" "lfence\t\n" "CPUID\n\t" "RDTSC\n\t
  " "lfence\t\n" "mov %%edx, %0\n\t" "mov %%eax, %1\n\t": "=r"
  (cycles_high), "=r" (cycles_low):: "%rax", "%rbx", "%rcx", "%
  rdx");
uint64_t t = ( ((uint64_t)cycles_high << 32) | cycles_low );
return t;
```

The difference of two such timings, similar to the variable t above, gives the number of clock cycles elapsed between the two timings. Dividing these clock cycles with the processor speed gives the time taken between the two timer calls. ¡changes required¿

Bibliography

[1] Bernard Menezes Pooja Mazumdar, Sampreet Sharma and Jyoti Gajrani. Challenges in implementing cache-based side channel attacks on modern processors, 2012-13.

[2] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on aes to practice. *IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[3] R.H. Saavedra and A.J. Smith. Measuring cache and tlb performance and their effect on benchmark runtimes. *IEEE Transactions on Computers, 44 (10)*, pages 490–505, October 1995.

[4] Kim H. Lee J. and Vuduc R. When prefetching works, when it doesnt, and why. *ACM Trans. Archit. Code Optim. 9, 1, Article 2 (March 2012), 29 pages*, 2012.

[5] Joshua Ruggiero. Measuring cache and memory latency and cpu to memory bandwidth. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-cache-latency-bandwidth-paper.pdf`, 2008.

[6] Dr David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. `http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`, 2008.

[7] Intel 64 and ia-32 architectures software developers manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2a-manual.pdf`, February, 2014.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier India, 4th edition edition, 2010.

[9] Ravi Sethi, Alfred V. Aho, D. Jeffrey Ulman, and Monica S. Lam. *Compilers Principles, Techniques, and Tools*. Pearson Education Singapore Pte Ltd, 2nd edition edition, 2008.

[10] D. M. Dhamdhere. *Operating Systems : A Concept-Based Approach*. Tata Mc-graw Hill Publishing Co.ltd., 3rd edition edition, 2012.

[11] Lmbench - tools for performance analysis. `http://lmbench.sourceforge.net/`.

[12] Untangling memory access measurements - memory latency. `https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W51a7ffcf4dfd_4b40_9d82_446ebc23c550/page/Untangling%20memory%20access%20measurements%20-%20memory%20latency`.

[13] Trent Rolf. Cache organization and memory management of the intel nehalem computer architecture. *University of Utah Computer Engineering*, December 2009.

[14] Keith Conrad. Generating sets. `http://www.math.uconn.edu/~kconrad/blurbs/grouptheory/genset.pdf`.

[15] Avinatan Hassidim. Cache replacement policies for multicore processors. 2010.

[16] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, 2006. `http://eprint.iacr.org/`.

[17] Chrono: Timing utility. `http://www.cplusplus.com/reference/chrono/`.

[18] Thomas H. "Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford" Stein. *Section 31.9: Integer factorization, Introduction to Algorithms*. Cambridge, MA: MIT Press, 2nd edition edition, 2001.

[19] Trilok Acharya and Meggie Ladlow. Cache replacement algorithms in hardware. May 2008.

[20] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. Cryptology ePrint Archive, Report 2013/448, 2013. `http://eprint.iacr.org/`.