Encryption Base Set Indexing to Protect from Cache Side Channel Attacks

ABSTRACT

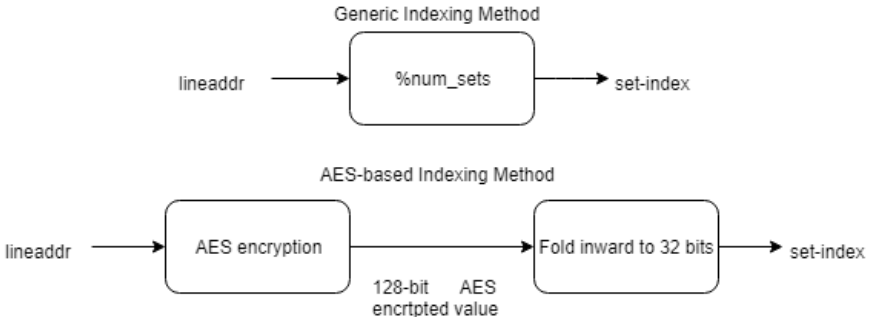
Much of the innovation in the computing industry has long been focused on bringing more performance to the table with subsequent processor designs. The complexity in the design and operation of these new mechanisms that aim to improve performance could induce security vulnerabilities that require significant resources to correct, especially when discovered later in the product life cycle. Recently discovered vulnerabilities in branch prediction logic and the memory system, like Spectre and Meltdown, reinforce the necessity that performance should not come at the cost of security. This report focuses on improving the security of cache systems and presents an approach to a model that could potentially reduce the risk of side-channel attacks.

INTRODUCTION

Cache side-channel attacks utilize the state of the cache to steal information from a target process. When an attacking process and a victim process execute together, the cache memory accesses by the attacking process in this state allow it to discern information about the nature and contents of data being accessed by the victim process in a shared set with sensitive data. The method of attack used by Spectre[1] and Meltdown[2], two cache side-channel vulnerabilities that were recently discovered, is by forcing the victim to pick the incorrect branch direction by having the attacker mis-train the branch predictor shared between these two processes.

Methods of obtaining information in cache have been detailed and demonstrated in previous literature, such as Flush+Reload, Evict+Time and Prime+Probe [5] [3]. These methods are used once the victim brings the sensitive data into the cache. While more vulnerabilities could exist [4], it is imperative to design a memory system that can minimize the risk of these attacks. There is ample scope for proposing a mechanism for solving these problems as very few solutions exist that can guarantee security.

This paper uses an AES-128 encryption model to dynamically generate the set index of a cache line based on an encryption key. The objective is to remove the notion of assurance that certain cache lines will only index certain sets in the cache. AES accomplishes this by flipping at least half the bits in an address. This allows us to use AES encryption on the set index as a method of redirection and increase the difficulty for an attacking process to steal information.



BACKGROUND

Modern processors use speculative execution to guess the correct path of execution when the hardware decodes a branch instruction. The pipeline executes the instructions that follow pre-maturely, on the

notion that time taken in waiting for the result and precious stall cycles would be saved if the execution was on the right path. Rather than wait many cycles for the branch result, the current register state is saved, and the system will execute the branch and the following code segment speculatively.

The processor would need to perform two different functions based on the accuracy of branch prediction. When the branch reaches the execute stage of the pipeline, the correct direction that should have been taken is now known. If the prediction was correct, precious cycles have been saved and the execution proceeds. If the prediction was incorrect, the entire state of the pipeline and registers will have to be flushed and reverted to the checkpoint state. The outcome of this prediction therefore is a significant contributor to performance. Attack vectors such as Spectre and Meltdown use vulnerabilities in this area and use cache side channel attacks to obtain information from the cache after exploiting the branch predictor vulnerabilities.

While developing a solution for such attacks was the motivation of this project, we try to develop a method to alleviate all side-channel attack vectors. Our paper focuses on methods to prevent side-channel attacks, by removing the guarantee that there is a direct relationship between the cache line address and calculation of the set-index for a cache. The key insight is that, by introducing a sense of redirection into the indexing, a process running on the core will not be able to attribute memory addresses to certain specific regions in the cache.

Multiple side-channel attack vectors exist, that utilize specific vulnerabilities in the memory system. All of these attack vectors make the assumption that the set-index is directly dependent on the physical/virtual address of the cache line. Some major attacks are described below.

1) Prime+Probe

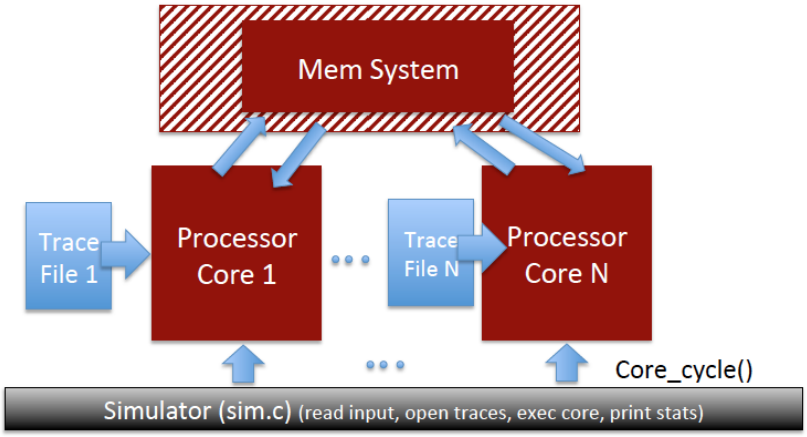
A Cross-core, Cross-VM attack touted to allow information gathering at a rate of 1.2Mb/s. The attack consists of three stages. The PRIME stage is when the attacker fills the entire cache with its own data set, evicting all other cache lines. The IDLE stage is when the victim process executes, inserting its own cache lines and evicting some of the attacker's in the process. In the PROBE stage, the attacker streams through the data set that was brought in to the cache to determine which line that were brought in during the PROBE stage were evicted. This information is then used to guess the address access location of the victim process.

2) Flush+Reload

A Cross-core, Cross-VM attack that utilizes the clflush instruction in the x86 ISA. This attack is a variant of PRIME+PROBE that relies on shared pages being used between the attacker and the victim. However, instead of filling the cache with it's lines, the attacker will flush out the shared line between the processes and detect if the victim brings it back into cache.

SIMULATION INFRASTRUCTURE

For simulating our implementation, we use a trace driven timing simulator *[Provided by Prof. Moinuddin Qureshi for an assignment in ECE 6100]* that models two levels of cache and a DRAM based main memory. We simulate the AES based set-indexing scheme that provides benefits against side-channel attacks. We however do not model a side-channel attack in this study. The simulator can be extended to a multicore implementation as needed. However, to keep simulation time short, we retain a single core model.



Our contribution to the code base involves designing the AES based encryption model, implementing a viable encryption scheme, and improving the performance of the simulator when AES has been implemented. We also coded segments to extract cache state information for our analysis. We believe that both the code contribution and the analysis of this study are sufficient contributions for this project.

We model the performance of the AES based encryption to analyze the effects on IPC and memory traffic. This allows us to determine the feasibility of the AES based indexing in modern processing engines. We do this by attributing a delay to the encryption-based model along the critical path for cache reads.

AES ENCRYPTION – IMPLEMENTATION

The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext. The AES algorithm can use cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. For our application, only the 128-bit version was implemented in C++

AES is based on a design principle known as a substitution–permutation network, a combination of both substitution and permutation, and is fast in both software and hardware. AES operates on a 4 × 4 column-major order matrix of bytes, termed the state. The key size used for an AES cipher specifies the number of repetitions of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext. For this implementation, a 128bit key is used and requires 10 cycles of repetition.

The AES Algorithm is broken down into several segments.

KeyExpansions—round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more. This is done after every key change.

InitialRound

- AddRoundKey—each byte of the state is combined with a block of the round key using bitwise xor.

Rounds

- SubBytes—a non-linear substitution step where each byte is replaced with another according to a lookup table.
- ShiftRows—a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
- MixColumns—a mixing operation which operates on the columns of the state, combining the four bytes in each column.
- AddRoundKey

Final Round (no MixColumns)

- SubBytes
- ShiftRows
- AddRoundKey.

SubBytes:

In the SubBytes step, each byte in the state matrix is replaced with a SubByte using an 8-bit substitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. While performing the decryption, the InvSubBytes step (the inverse of SubBytes) is used, which requires first taking the inverse of the affine transformation and then finding the multiplicative inverse. For this implementation these were implemented by lookup tables.

ShiftRows:

The ShiftRows step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. For blocks of sizes 128 bits and 192 bits, the shifting pattern is the same. Row n is shifted left circular by n-1 bytes. In this way, each column of the output state of the ShiftRows step is composed of bytes from each column of the input state. The importance of this step is to avoid the columns being encrypted independently, in which case AES degenerates into four independent block ciphers.

MixColumns Step:

In the MixColumns step, the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with ShiftRows, MixColumns provides diffusion in the cipher.

AddRoundKey step:

In the AddRoundKey step, the subkey is combined with the state. For each round, a subkey is derived from the main key using Rijndael's key schedule; each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])                  // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                              // See Sec. 5.1.1
      ShiftRows(state)                             // See Sec. 5.1.2
      MixColumns(state)                            // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

PSEUDOCODE FOR ENCRYPTION

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

PSEUDO CODE FOR KEY EXPANSION

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) // See Sec. 5.1.4

   for round = Nr-1 step -1 downto 1
      InvShiftRows(state)                    // See Sec. 5.3.1
      InvSubBytes(state)                     // See Sec. 5.3.2
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      InvMixColumns(state)                   // See Sec. 5.3.3
   end for

   InvShiftRows(state)
   InvSubBytes(state)
   AddRoundKey(state, w[0, Nb-1])

   out = state
end
```

PSEUDOCODE FOR DECRPYTION

ANALYSIS OF OUR IMPLEMENTATION

We used the following benchmarks from the SPEC 2006 benchmarks:

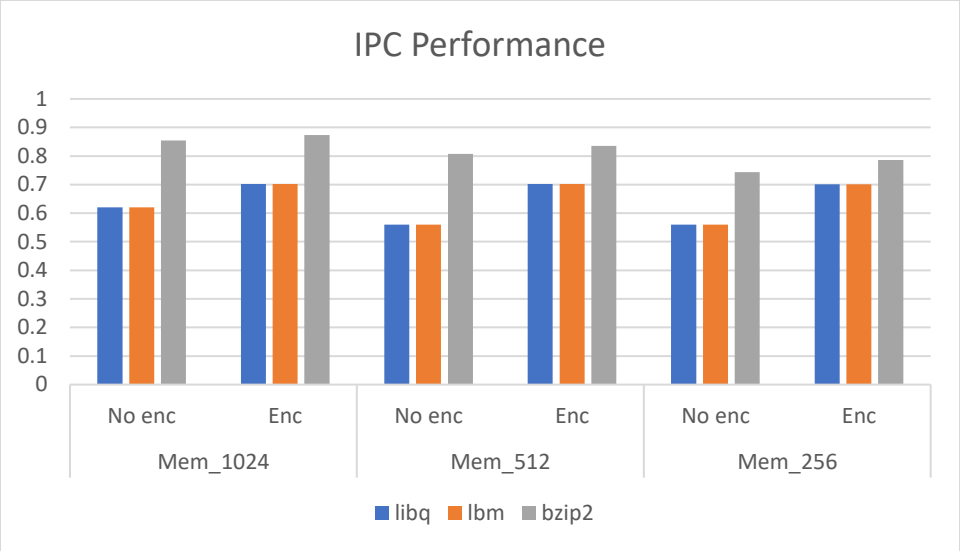
- Bzip2
- Libq
- Lbm

1. Single core

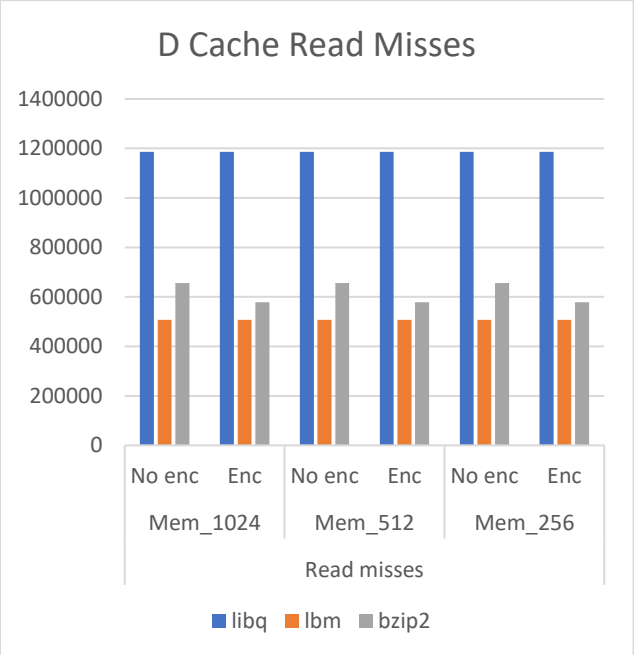
When we talk about performance, the first key factor is the IPC that the system can deliver for various workloads. That is exactly what we start with in our analysis of the multi-hierarchy memory system that we designed with AES encryption for this work.

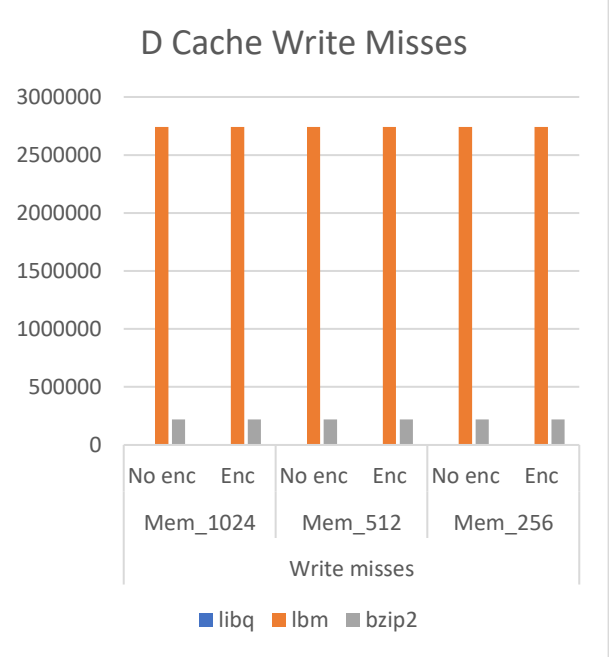
We notice two things from the IPC (Instructions per cycle) behavior of these applications. The first observation is that bzip2 performs better than the other two in all configurations, regardless of activation of the encryption algorithm. This is because the bzip2 application is a compression workload whereas Libq and lbm are streaming workloads which cannot usually use any locality provided by the cache.

The interesting thing to notice with respect to our implementation is that the IPC improves for **ALL** workloads when using the encryption. This could be due to encryption spreading the cache lines across the associative cache better than the one without encryption. This goes on to show that AES Encryption in caches **not only provides security as mentioned in the previous sections of the report, it also improves performance.** Let us now take a deeper dive into why the IPC improvement.

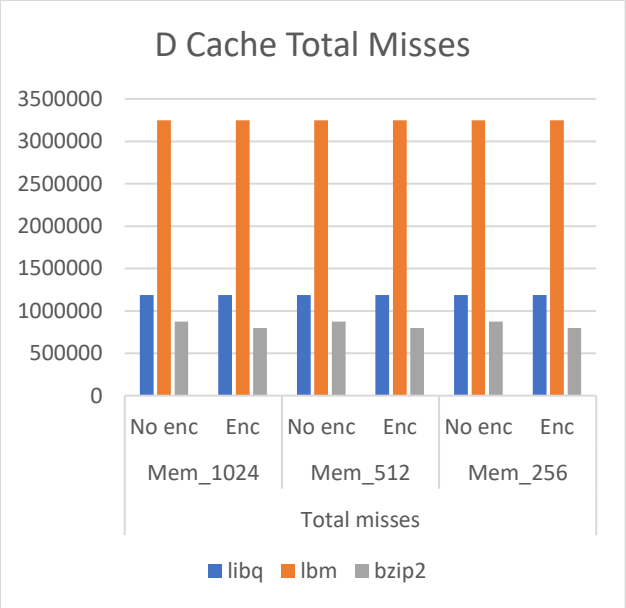


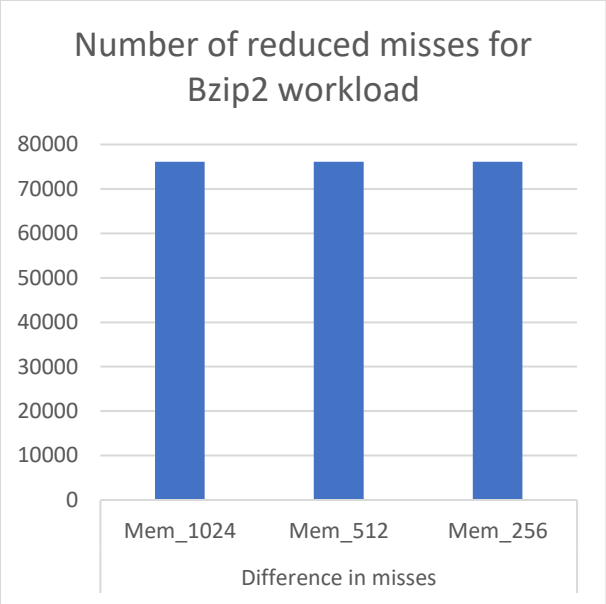
As we can see from the figure, the AES encryption performs better for every workload. The most direct impact on IPC comes from caches; each miss incurring the penalty of going to a farther away memory block in the hierarchy. Therefore, looking at cache misses could provide significant insight into the performance achieved.



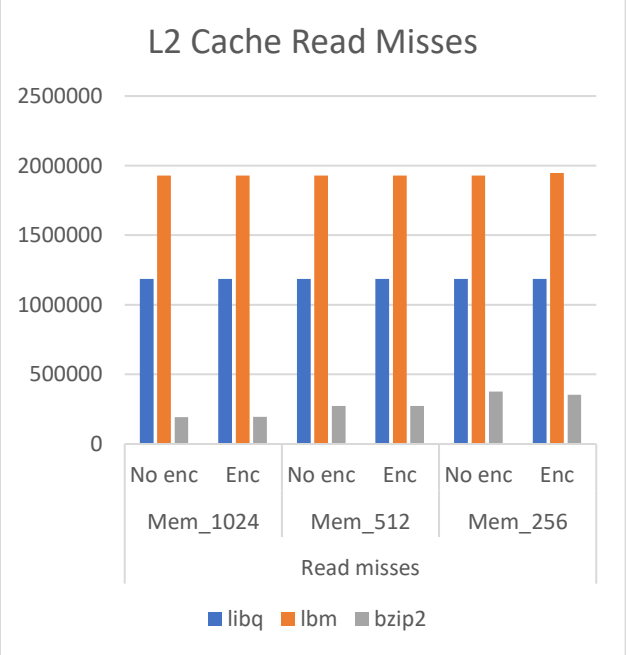


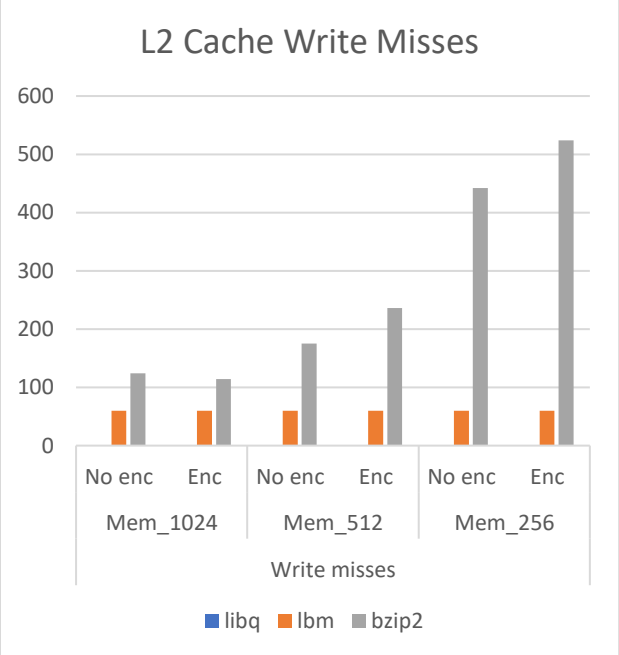
Let us look at the read misses bzip2 workload where the improvement is more prominent. The reduction in the read misses in the D cache prevents longer latency memory accesses and hence stall cycles in the pipeline leading to a much better IPC. This is as in line with our expectations that a reduction in cache misses would have resulted in the improvement in IPC. We urge the reader to look at the figure showing the number of reduced misses including reads and writes for bzip2; which is more than 70,000 misses avoided explaining the significant improvement in IPC.





Let us also explore L2 cache statistics.



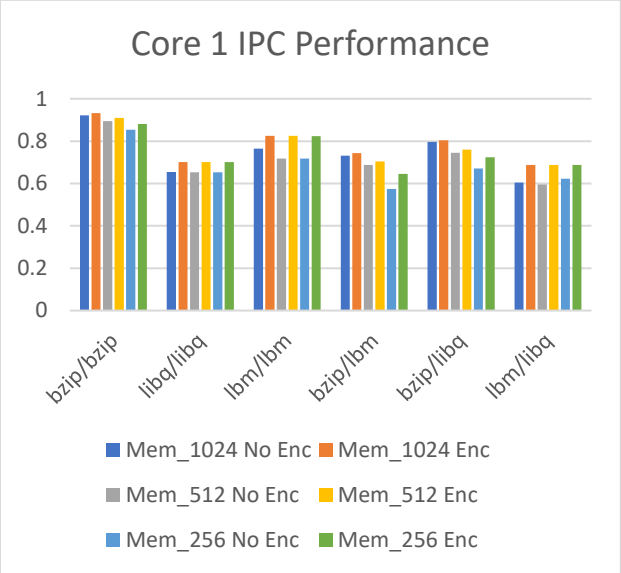


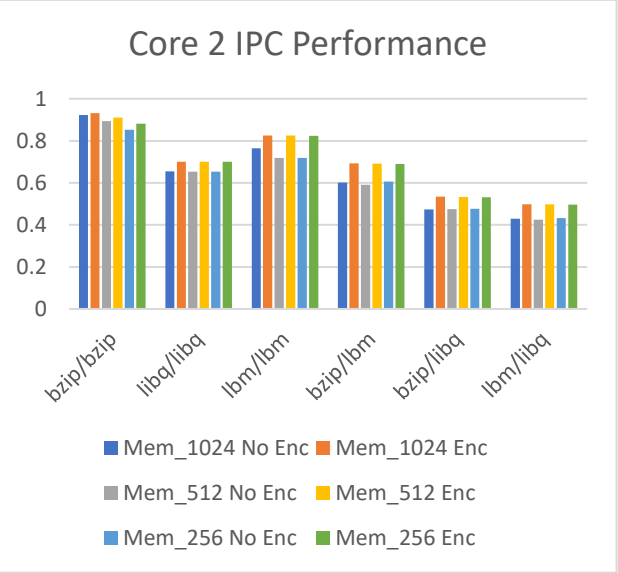
Though we observe some increase in write misses with encryption, they are in the orders of 10's and fail to affect the performance enough to degrade the benefits of the more critical L1 cache. We see some improvement in read misses in some cases but again, the reduction is little to have any affect on the improvement of performance. This is expected as the improvement in L1 cache leaves little or no scope of improvement for the L2 cache.

Let us now look at how the same system affects a dual-core performance.

2. Dual Core performance

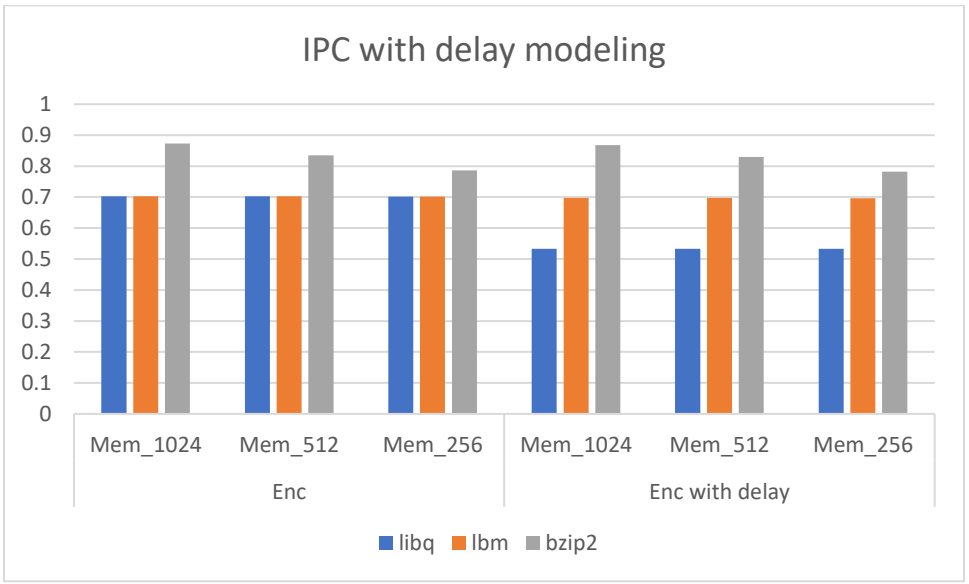
Let us start by looking at the IPCs of the two cores. We ran all possible combinations of the workloads we are used for the analysis. Note that bzip/lbm signifies that bzip was run on Core 1 while lbm benchmark was run on the 2nd core. A key point to note is also that bzip/lbm and lbm/bzip are essentially the same and results in the numbers reported for Core 1 and Core 2 just reversed.





We see significant improvement in the IPC when activating encryption for all combinations on both the cores. The reasons for these are similar as described in the uni-core section. We conducted similar experiments and found the same trend and those results have been omitted here for brevity.

3. Modeling delay of performance encryption



We modeled the delay of encryption in the L2 caches only as the L1 caches are designed to be fast and within single cycle delay access. L2 accesses which are comparatively longer latency can have the encryption delay modeled without destroying the design philosophy of caches.

We noticed some degradation in the IPC while performing delay modeling as we approach more realistic simulation of performing encryption. Performing encryption comes in the critical path and the latency of the encryption was modeled as 2 cycles on every new encryption in the L2 cache. However, the implementation of the Look up table (LUT) can help reduce this delay and that delay was modeled as a single cycle delay mitigating some of the effects of performing encryption while accessing the cache.

Overall, in conclusion we saw that encryption can not only help provide security but also improve performance of the memory hierarchy and improve the IPC of various workloads. We hope the simulation tool used for this project will be beneficial for further research and can proliferate many more ideas in the field with the ease of tweaking parameters and modeling innovative solutions coming from the simulator.

CONTRIBUTIONS

Adithya Nallan Chakravarthi
GTID: 903290650

- Understanding the importance of AES encryption in caches and its placement in the memory hierarchy.
- Integrating the AES encryption module built as part of the project with the augmented simulator being used.

Chaitra Hegde
GTID: 903291630

- Data collection using various workloads and parameters and carefully tuning the simulator to match our needs.
- Implementing the performance counters and integrating them with the simulator.
- Support implementation of cache accesses and install functions.

Parth Mannan
GTID: 903291571

- Analysis of performance and cache parameters while tuning the simulator to match our needs and performing delay modeling in the encryption module to approach more realistic results.
- Implementation of accesses to the cache and the installing of cache lines into the data sans encryption module.
- Support implementation of look up table functionality for encryptions.

Mathew Prabakar
GTID: 903326601

- Implemented 128-bit AES encryption, AES decryption and all required functions.
- Created a simple interface to use AES encryption within the main simulator.
- Implemented the look up table structure to store encrypted cache line addresses to speed up the simulator.