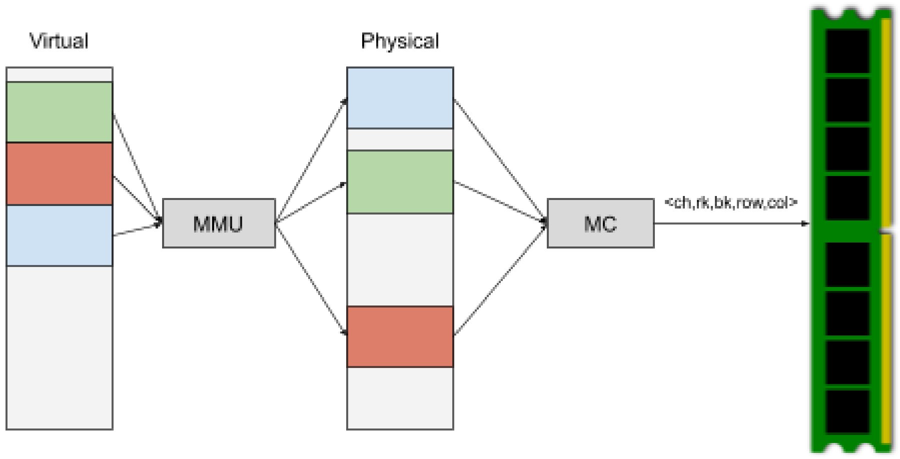
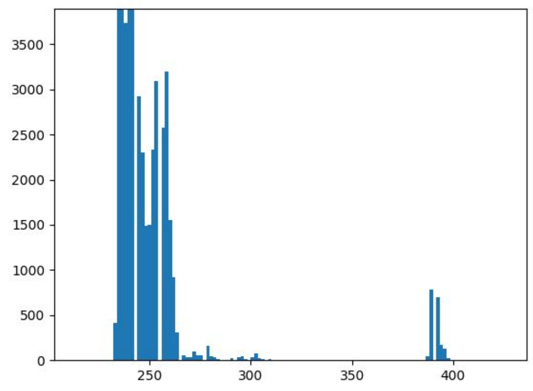
DRAMA attacks

DRAMA[1] is a known side-channel attack used to leak information about the geometry of memory addresses in DRAM. Memory addresses follow two levels of translations: virtual→physical, and physical→DRAM address (Figure below).



The goal of this assignment is to partially reverse engineer the mapping between physical addresses and DRAM addresses. As we saw in the Memory Subsystem lecture, data inside DRAM is stored inside an 2d-array of cells known as a **bank**. A bank is composed of multiple rows. When you want to read data from these rows, you need to bring the content inside a structure known as **Row Buffer** that acts as a “cache”. Accessing two separate rows in the same bank causes a slow down in the memory accesses due to a conflict in the row buffer. As you can see in the plot below you can easily detect these conflicts when measuring time from software.



This lab is divided in two parts:

1. In the first part you will learn to detect these bank conflicts
2. In the second part you will partially recover the functions used for mapping these addresses

[IMPORTANT!] We provide you with some code stubs and a Makefile we ask you to use for the assignment. Do use them, since we use them for grading. If the code does not compile with the given Makefile you will lose points :)

Bank Conflict (6 points)

File: `conflict.c`
Build with: `make conflict`

The goal of this part is to manage to plot a histogram like the one shown above.
You can simply mmap a large chunk of memory (e.g., 1GB) and generate a whole bunch of random `probe` addresses within this allocation (use the provided code stub). Then, for each of these random addresses you will measure the access time when accessed together with a `base` address (always the same). Plotting the frequency of the access times should give you a histogram like the one above.
You will need this plot to identify a cut-off time to identify the addresses landing on the same bank (e.g., 350 cycles in the plot above).

Expected `stdout`: ONLY the time measurements of each of your address tuples. This will be piped into our grading script. The code should complete in less than 2 mins.

Deliverable:

- The source code of your program.
- A histogram like the one above.

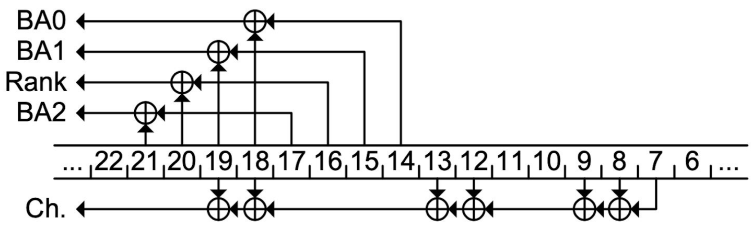
Hints:

- Reducing noise
    - Perform multiple accesses over the same tuple `(base, probe)` in order to filter out the noise.
    - Median is a more resistant value than mean
    - You can call `sched_yield()` before timing to avoid your measurement from being interrupted by context switches

- Use the CPU timestamp counter for high precision timing (i.e., `rdtscp` instruction).
- After the first memory access the data will be stored in the cache. Use the `clflush` instruction to evict this data from the cache.
- The number of addresses generating a bank conflict (i.e., right cluster on the histogram) depends on the memory configuration. For instance on a 1-channel, 2-rank, 16-bank configuration the number of conflicts should be around 1/(1 x 2 x 16) = 1/32 of the total tuples. You can verify the configuration of the system by reading the file `/etc/dram-config` on the different machines of the hwsec cluster.
- Different systems may have different cut-off frequencies due to different DRAM timings.
- The binary should run fairly quickly. For instance, you can measure around 50K tuples in <10s.

Function Recovery (4 pt)

File: `functions.c`
Build with: `make functions CF=-DCF=cutoff_val`
In the previous task you managed to recover a cut-off time to detect addresses generating bank conflicts. In this second part we will try to recover the functions mapping your physical addresses to the same bank. These functions are simply XORs of different bits of the physical address (Figure from the DRAMA paper [1]).



Intel Haswell (desktop system) – DDR3

In the example above you can see that bits 14⊕18 for instance decide one of the bank address bits. Since you do not have access to the physical address from userspace we remove one level of indirection (virtual→physical) by providing you with 1GB HugePages. In a 1GB HugePage the last 30 bits of the virtual and physical address are the same allowing you to identify any function using bits >=30 (we provide you with the allocated memory already in `functions.c`).

You can now generate a pool of conflicting addresses (i.e., tuples with access time > cut-off) and try to bruteforce all the possible XOR functions of *n* bits over the lowest 30 bits of your addresses.
The end goal is to find functions where `base ⊕ fn == probe ⊕ fn` for every `probe` address—or almost since you may have some false positives.

In our code stub we provide you with the function `next_bit_perm(size_t v)` generating the next possible value with *n* bits set. For instance if you want to bruteforce all the functions with a single bit set (*n* = 1) you will start testing for `0b1` and then call `next_bit_perm(0b1) = 0b10`. You will continue this process up to bit 30 since it's the last bit you control. And you will do this for 1≤*n*≤6. For instance, if the system is configured with 2 channels, the function detecting the channel XORs together 6 address bits (see Figure above).

Expected `stdout`: (ONLY) All the candidate functions in hex format (e.g., `0x4080`). This will be piped into our grading scripts. The code should complete in less than 2 mins.

Deliverable:

- The source code of your program.

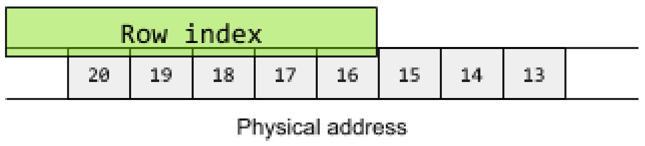
Hints:

- You can avoid any function xoring the lowest 6 bits since they are used to address cachelines (more on this in the next lectures).
- The number of total functions required depends on the configuration of the system. For instance in the same 1-channel, 2-rank, 16-bank configuration you need log2(1x2x16) = log2(32) = 5 functions. However, your code should identify more than 5 candidates. This is due to the fact that some of the functions you identify are linear combinations of the others.
- You can compute the minimal set of functions with `py/fn-reduce.py` script.
- For every system of the hwsec cluster you can read the file `/etc/dram-config` to verify your results.
- While in the figure above you can distinguish between ranks, channels and banks address functions, from software you actually cannot detect the difference since the only side channel are the bank conflicts. For this reason you can consider the 1x2x16 configuration above as a 32-banks configuration.

Bonus: Row Indexing Recovery (1pt)

File: `row-index.c`
Build with: `make rows CF=-DCF=cutoff_val`

Now you know the addressing functions of addresses mapping to the same bank you can use them to find addresses mapping to the same row (inside the same bank). Memory accesses mapping to the same row (known as **row hits**) are faster than row conflicts which can be once again detected by timing. The bits used for row indexing are usually higher order bits (≥16) and are simply linearly mapped to the physical address (see figure below).



As a result you can try to flip single address bits (≤30) and see if the access is faster or generate a random pool of addresses, as in the previous task, and try to bruteforce the functions. You can verify the function of the machine in `/etc/dram-config`.

Expected `stdout`: (ONLY) the bitmask computed by your code limited to bit 30 in hex format (e.g., `0x3fff8000` if bits higher than 15 are used for row indexing).

Deliverable:

- The source code of your program.

References

[1] Pessl, P, et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." *arXiv* (2015): arXiv-1511.