Computer Architecture HW2

Fabian Wüthrich

October 18, 2020

1 Critical Paper Reviews [1000 points]

see here

2 RowHammer [200 points]

2.1 RowHammer Properties

a) True

b) False

c) True (i.e. protector cells)

d) True (i.e. aggressor or protector cells)

e) True

2.2 RowHammer Mitigations

a) If the refresh interval is reduced from 64ms to 8ms, each row is refreshed 8 more times. Thus, bank utilization increases to $8U$ and energy consumption increases to $8E$

b) With a doubling of the rows, the mitigation is still possible, but the bank is occupied by refresh operations 80% of the time ($U = 0.05 \times 8 \times 2 = 0.8$).

With another doubling of the rows, the mitigation cannot be implemented because we cannot refresh every row in 8ms ($U = 0.05 \times 8 \times 4 = 1.6$).

c) No, we need to know which rows are adjacent.

d) With a 8ms refresh interval an attacker can issue a limited number of activations, constrained by t_{RC}. If T is less than the maximum number of activations issued in 8ms, we can guarantee the same level of security. We have $t_{RC} = t_{RAS} + t_{RP} = 35ns + 13.5ns = 48,5ns$ and therefore

$$T = \frac{8ms}{48.5ns} = 164948.4536 \approx 164948$$

e) A single counter requires $log_2(164948) = 17.3317 \approx 18$bits and each row needs a counter. Thus,

$$18\,bits/row \times 2^{15}\,rows/bank \times 8\,banks \times 2\,ranks = 9Mbit$$

When the number of rows per bank and the number of banks per chip are doubled we get

$$18\,bits/row \times 2^{16}\,rows/bank \times 16\,banks \times 2\,ranks = 36Mbit$$

f) The memory controller performs unnecessary activations which is bad for performance and energy consumption.

g) Let X be a RV that describes the number of errors in a year. Each 64ms interval can be seen as a independent experiment of getting an error. Therefore, X has a binomial distribution with the parameters $p = 1.9 \cdot 10^{-22}$ and $n = \frac{365 \cdot 24 \cdot 3600s}{64ms} = 492'750'000$. Then, the probability can be calculated as follows:

$$\begin{aligned} P(X \geq 1) &= 1 - P(X = 0) \\ &= 1 - (1-p)^n \\ &= 9.3622 \cdot 10^{-14} \end{aligned}$$

3 Processing in Memory: Ambit [200 points]

3.1 In-DRAM Bitmap Indices I

a) All users occupy $\frac{u}{8}$ bytes so we need $\frac{u}{8 \cdot 8k}$ subarrays. Including the weeks, $\frac{u \cdot w}{8 \cdot 8k}$ rows are occupied.

b) Using Ambit, we copy each row into the *Operand* row and perform a bulk `and`. This requires $\frac{u \cdot w}{8 \cdot 8k} \times (t_{rc} + t_{and})$ seconds overall. Then, we need to transfer $\frac{u}{8}$ bytes to the CPU and count the bits. This takes $\frac{u}{8X}$ seconds. Therefore, the throughput is

$$\frac{u}{\frac{u \cdot w}{8 \cdot 8k} \times (t_{rc} + t_{and}) + \frac{u}{8X}} \, users/second$$

c) The CPU has to transfer all users and weeks from memory which takes $\frac{uv}{8X}$ seconds. Therefore, the throughput is

$$\frac{u}{\frac{uv}{8X}} = \frac{8X}{w} \, users/second$$

d) We want that the execution time on the CPU is lower than in memory. Therefore,

$$\frac{uv}{8X} < \frac{u \cdot w}{8 \cdot 8k} \times (t_{rc} + t_{and}) + \frac{u}{8X}$$

Solving the inequality for w gives

$$w < \frac{1}{1 - \frac{X}{8k} \times (t_{rc} + t_{and})}$$

3.2 In-DRAM Bitmap Indices II

a) According to the cache specification we use `addr[5:0]` for the offset, `addr[22:6]` for the set index and `addr[31:23]` as tag. In each loop iteration the addresses `0x05000000+4*i`, `0x06000000+4*i` and `0x07000000+4*i` are loaded into cache. These addresses map to the same set but have a different tag so they evict each other out of the cache. Thus, the code has only cache misses and requires $3 \times 2^{15} \times 100$ cycles.

b) i) Ambit can perform bitwise operations on the size of a row so the loop iterates $\frac{database\ size}{row\ size} = \frac{32 \cdot 1024 \cdot 4\,bytes}{8 \cdot 1024\,bytes} = 16$ times. The query can be executed as in the listing below:

```
for(int i = 0; i < 16; i++){
    bbop_xor 0x08000000,            0x05000000 + i*8192,    0x0A000000
    bbop_xor 0x09000000,            0x06000000 + i*8192,    0x0B000000
    bbop_or  0x07000000 + i*8192,   0x08000000 ,            0x09000000
}
```

ii) Ambit requires 25 ACTIVATE and 11 PRECHARGE commands for `bbop_xor` and 11 ACTIVATE and 5 PRECHARGE commands for `bbop_or`. The query executed with Ambit takes $16 \times (2 \times (25 \times 50 + 11 \times 20) + (11 \times 50 + 5 \times 20))$ cycles so the speedup is $\frac{9830400}{57440} \approx 171$x

4 In-DRAM Bit Serial Computation [200 points]

a) The STREAM benchmark transfers two arrays with 102'400 4-bit elements and performs no computation. In total, $\frac{2\times102'400\times4}{8} = 102'400$ bytes are moved to the CPU. Therefore,

$$M = \frac{num\_bytes}{t_{cpu}} = \frac{102'400}{0.00001} = 10.24\,GB/s$$

The first experiment runs 65'536 operations (additions) and transfers $\frac{3\times65'536\times4}{8}$ bytes between CPU and memory. With $M = 10.24$ GB/s we get

$$K = \frac{t_{cpu} - num\_bytes/M}{num\_operations} = \frac{0.0001 - \frac{3\times65'536\times4}{8}/10.24\cdot10^9}{65'536} = 1.38\,ns/operation$$

b) The Ambit base implementation is shown in the following listing:

```
// S = 0x00C00000
// Cout/Cin = 0x00900000
for(int i = 0; i < num_bits_per_element; ++i) {
    bbop_xor 0x00700000,            0x00A00000+i*0x2000,    0x00B00000+i*0x2000
    bbop_xor 0x00C00000+i*0x2000,   0x00700000,             0x00900000
    boop_and 0x00700000,            0x00700000,             0x00900000
    boop_and 0x00800000,            0x00A00000+i*0x2000,    0x00B00000+i*0x2000
    boop_or  0x00900000,            0x00700000,             0x00800000
}
```

c) We get the maximum number of operations when we use each bit of a row as an element. With a row size of 8KB we can perform $8 \times 1028 \times 8 = 65'536$ additions i.e. max. number of operations.

Ambit requires 25 ACTIVATE and 11 PRECHARGE commands for `bbop_xor` and 11 ACTIVATE and 5 PRECHARGE commands for `bbop_or` and `bbop_and`. The code executed with Ambit takes $n \times (2 \times (25 \times 20ns + 11 \times 10ns) + 3 \times (11 \times 20ns + 5 \times 10ns) = n \times 2030\,ns$. The maximum throughput is then $\frac{65'536}{2030\cdot n\cdot10^{-9}}$ operations per second.

d) As calculated in subtask (c) the Ambit-based implementation takes $2030n$ns to execute. The execution time of the CPU-based implementation is $1.38 \times 10^{-9} \times 65'536 + \frac{3\times65'536\times n}{8\times10.24\times10^9} \approx 2.4 \times 10^{-6}$ns. Thus, there is no element size that makes the CPU-based implementation faster than the Ambit implementation.

5 Caching vs. Processing-in-Memory [200 points]

a) The inner loop has two iterations and two memory accesses. The outer loop iterates 16 times and has one memory access. Therefore, the execution time is $50 \times (16 + 16 \times 2 \times 2) = 4000$ cycles.

b) According to the cache specification two elements fit in each cache block. The code has the following access pattern per outer loop iteration:

1. Iteration
Access: A[0], B[0], A[1], B[1], A[0]
H/M patter: M M M M M
Latency: $5 \times 50 = 250$ cycles
Cache: Set 0: A[0:1]

2. Iteration
Access: A[1], B[0], A[2], B[1], A[1]
H/M patter: H M M H M
Latency: $2 \times 5 + 3 \times 50 = 165$ cycles
Cache: Set 0: A[0:1] Set 1: A[2:3]

3. Iteration
Access: A[2], B[0], A[3], B[1], A[2]
H/M patter: H M H H H
Latency: $4 \times 5 + 1 \times 50 = 70$ cycles
Cache: Set 0: B[0:1] Set 1: A[2:3]

4. Iteration
Access: A[3], B[0], A[4], B[1], A[3]
H/M patter: H H M H H
Latency: $4 \times 5 + 1 \times 50 = 70$ cycles
Cache: Set 0: B[0:1] Set 1: A[2:3] Set 2: A[4:5]

5. Iteration
Access: A[4], B[0], A[5], B[1], A[4]
H/M patter: H H H H H
Latency: $5 \times 5 = 25$ cycles
Cache: Set 0: B[0:1] Set 1: A[2:3] Set 2: A[4:5]

Now the single-miss and no-miss patterns are interleaved until the end.

Latency: $250 + 165 + 70 + 7 \times 70 + 6 \times 50 = 1275$ cycles

c) This is the same as the processor without a core but with a latency of 10 instead of 50 cycles. The new latency is 800 cycles.

d) Increasing the cache size would not eliminate the conflicts between A and B in the first set, so she is not correct.

e) One possibility could be to use a set-associative cache. 2-ways would eliminate the conflict in set 0 so only compulsory misses occur.