ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs

Fei Gao
Department of Electrical Engineering
Princeton University

Georgios Tziantzioulis
Department of Electrical Engineering
Princeton University

David Wentzlaff
Department of Electrical Engineering
Princeton University

MICRO, 2019

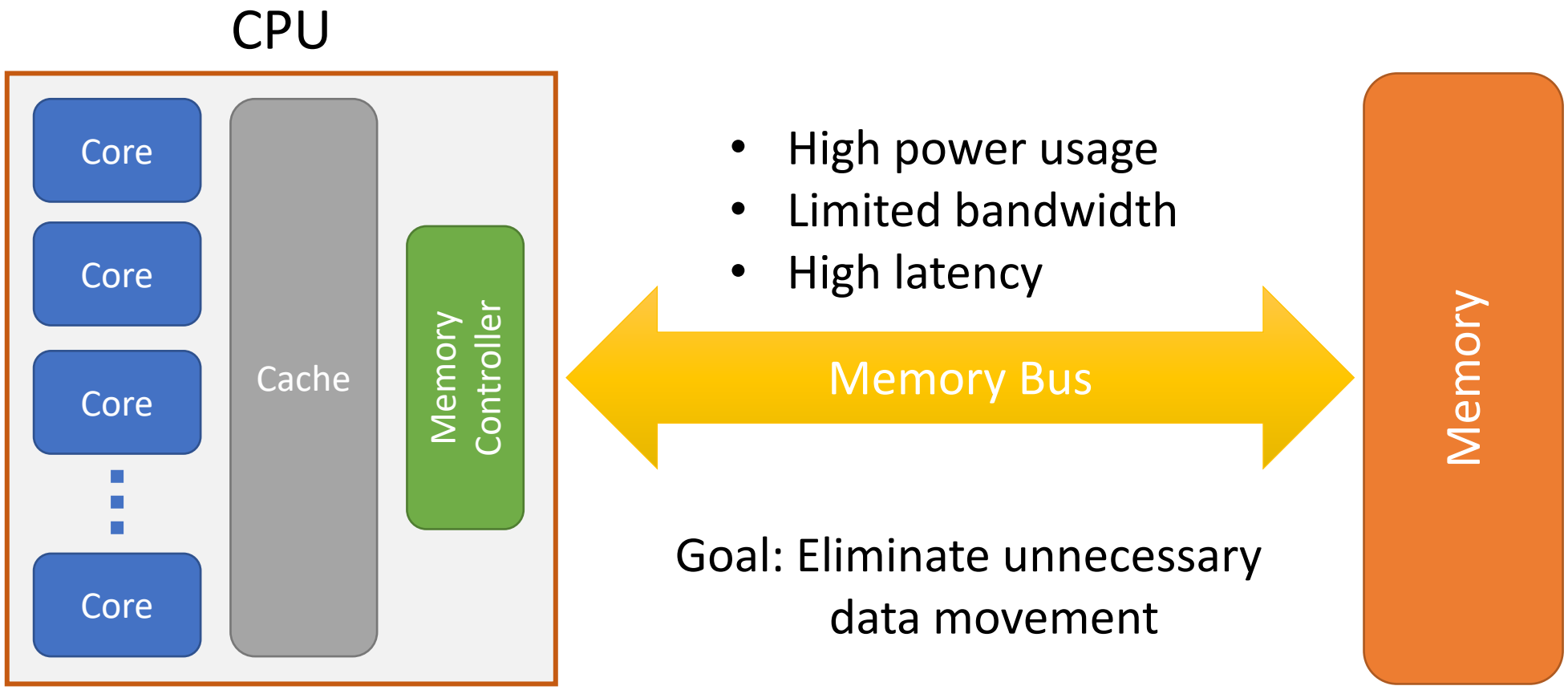
Lorenzo Rai
1.4.2021
Seminar in Computer Architecture ETHZ – Spring 2021

Executive Summary

- **Problem**: Memory latency and bandwidth are a significant power and performance bottleneck in modern computing systems
- **Goal**: Perform in-memory computing using current off-the-shelf commodity DRAM and demonstrate a framework for arbitrary computations.
- **Key Approach**: Violate DRAM timing constraints to achieve non standard state
- **Evaluation**: Test feasibility and reliability of using 32 different modules from 7 different vendors. Performance and efficiency of AND/OR and 8-bit ADD are tested on modules that can reliably perform the operations.
- **Results**:
  - Nearly all chips can perform row copy on at least some rows
  - 3 modules were able to perform AND/OR operations
  - If an operation is possible then the reliability of the operation is high (between 92.5% to 99.98% of rows that can perform AND or OR have 100% success rate)
  - Peak throughput of 19 GOPS for 8-bit AND/OR and 2.46 GOPS for 8-bit ADD

Background, Problem & Goal

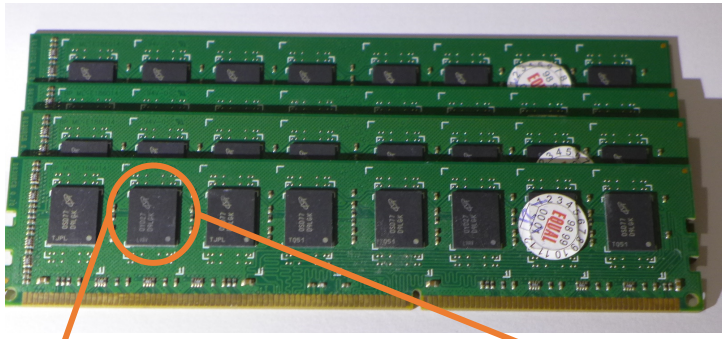
Memory Wall

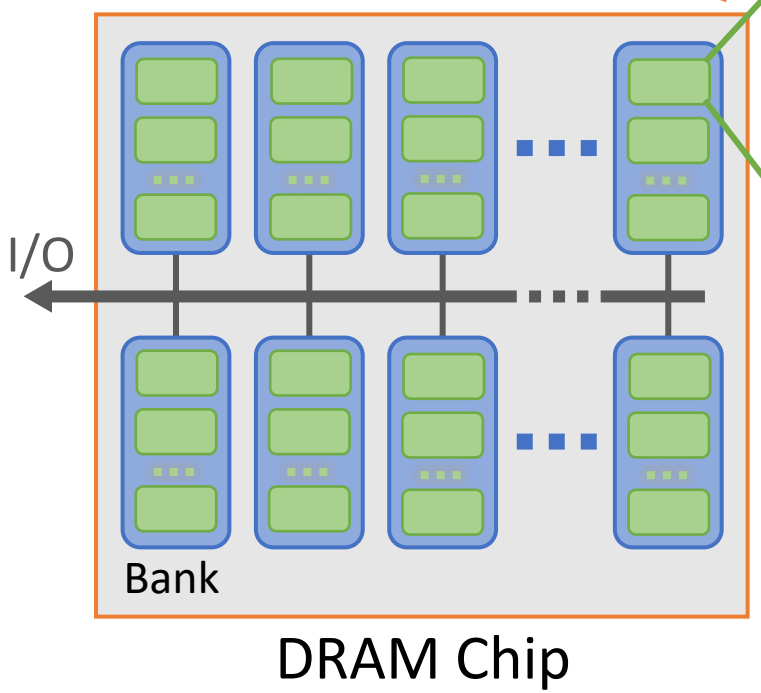

CPU
Core
Core
Core
Core
Cache
Memory Controller
High power usage
Limited bandwidth
High latency
Memory Bus
Memory
Goal: Eliminate unnecessary data movement

Idea: Move Computation to Memory

- Data doesn’t need to be moved from main memory to CPU for computation
  - Reduced power usage of the memory bus
  - Bandwidth is available for other data
  - CPU is free to work on other stuff

How DRAM works

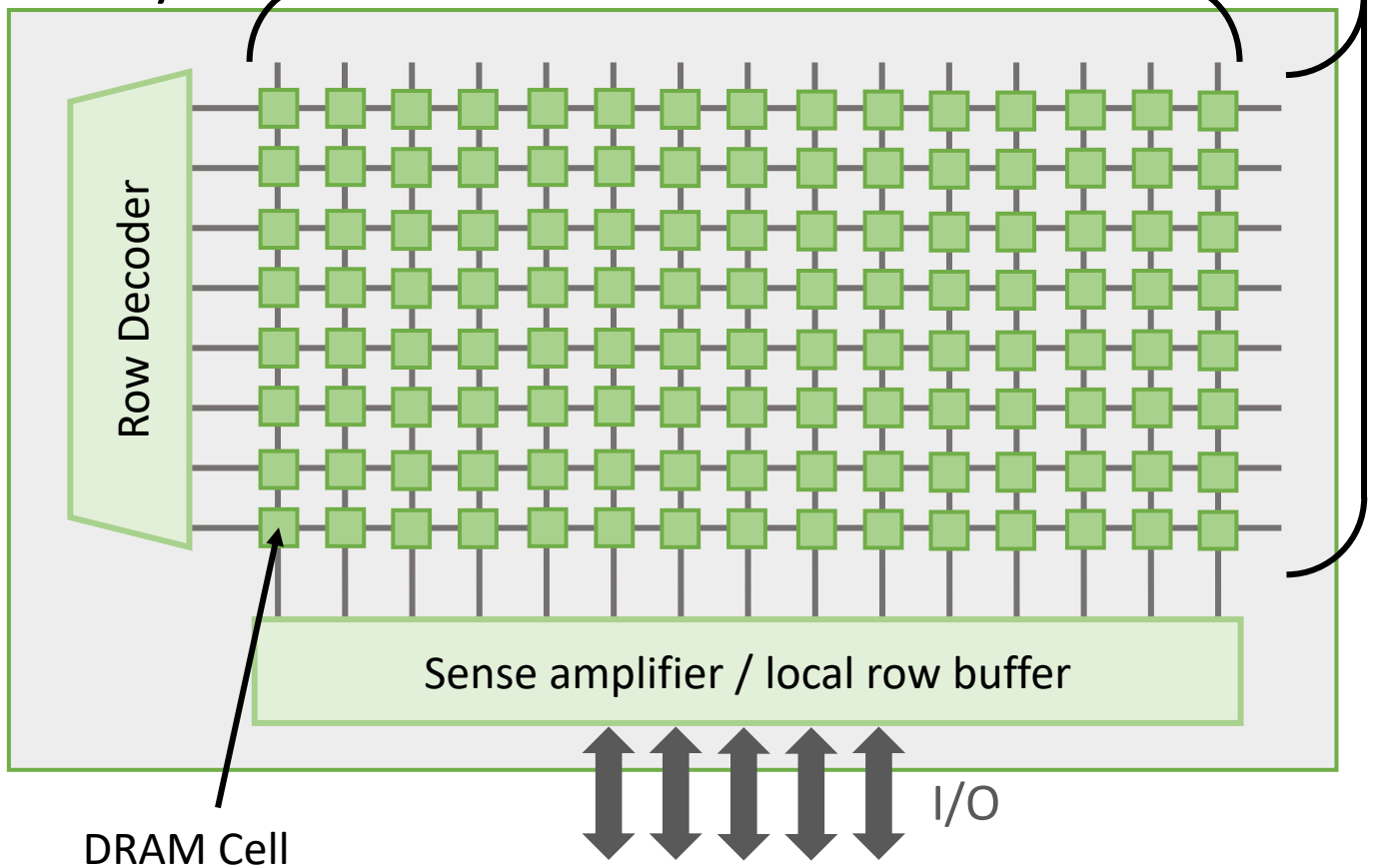




Columns / bitlines
Typically 8kB

Rows / wordlines
Typically 512

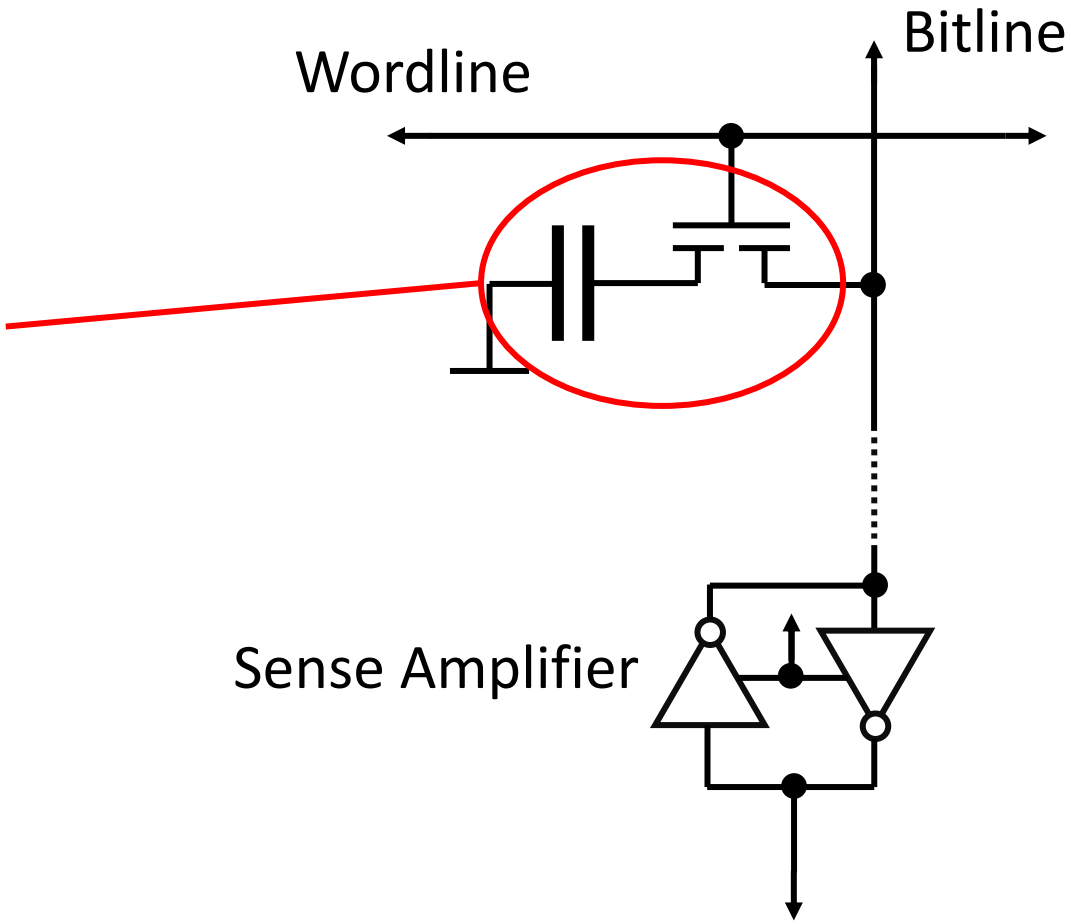
Subarray



DRAM Cell
1 bit

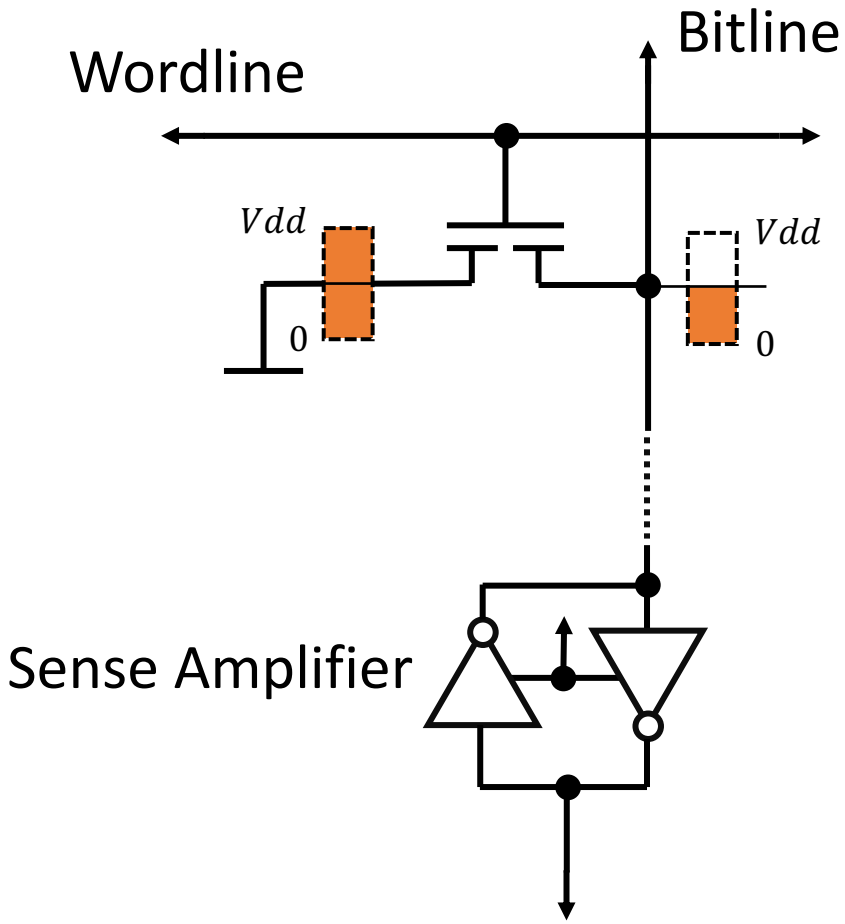
DRAM Cell Operation

Single DRAM cell consisting of a capacitor and transistor

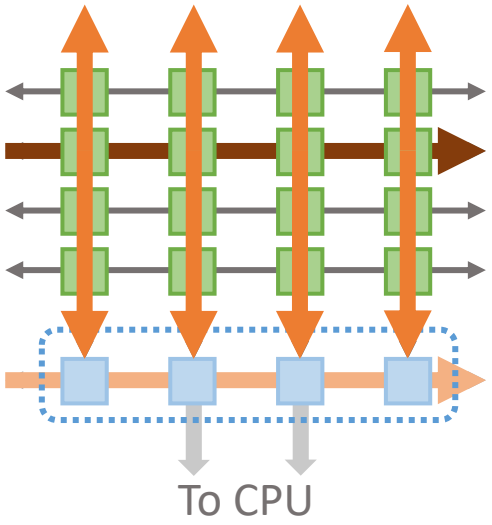


DRAM Cell Operation

- Bitline is precharged with Vdd/2
- After activation, charge from the cell pulls the bitline voltage towards the cell state
- Sense amplifier amplifies the deviation pulling the bitline either towards 0 or Vdd



DRAM Commands

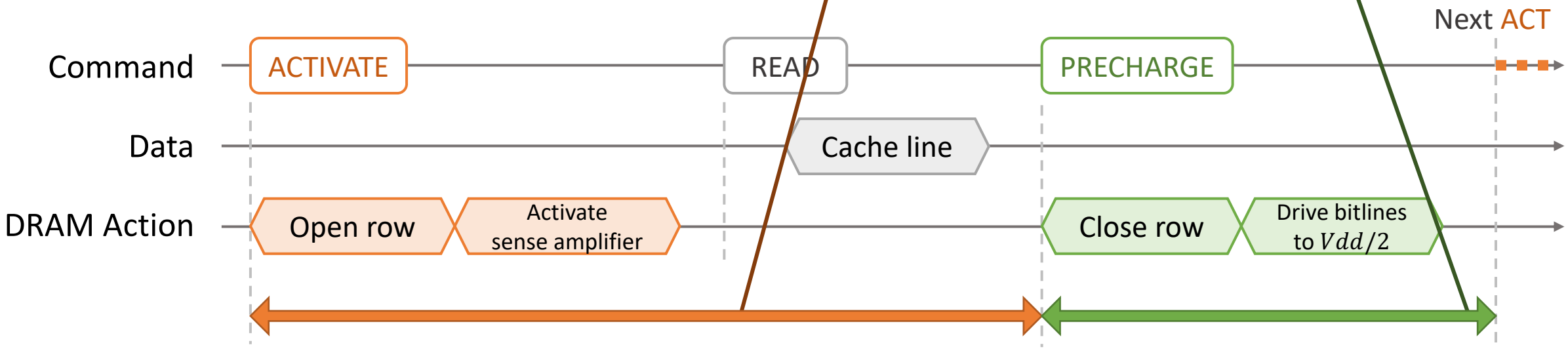


Row access strobe: $tRAS$

Minimum delay between ACTIVATE and the next PRECHARGE command

Row precharge: tRP

Minimum delay between PRECHARGE and the next ACTIVATE command



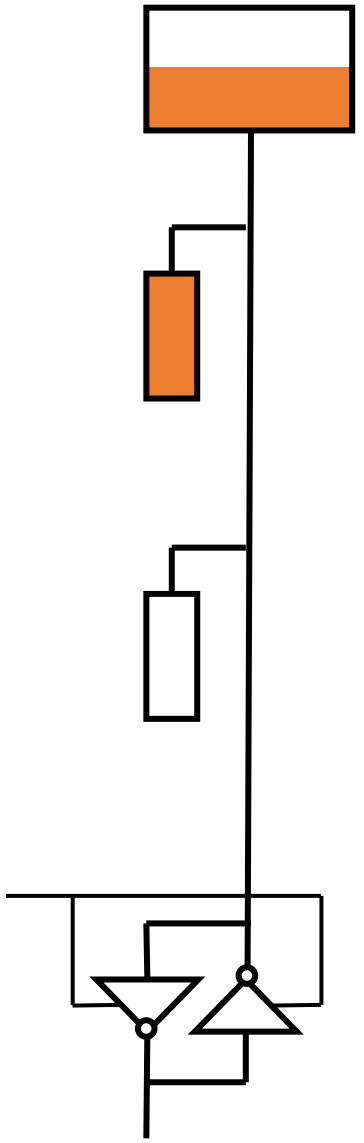
Modified slide from "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization" by Onur Mutlu

In-Memory Copy

RowClone: Fast and Energy-Efficient
In-DRAM Bulk Data Copy and Initialization

MICRO 2013

1. Open source row to copy data into row buffer
2. Open destination row to move row buffer data into destination

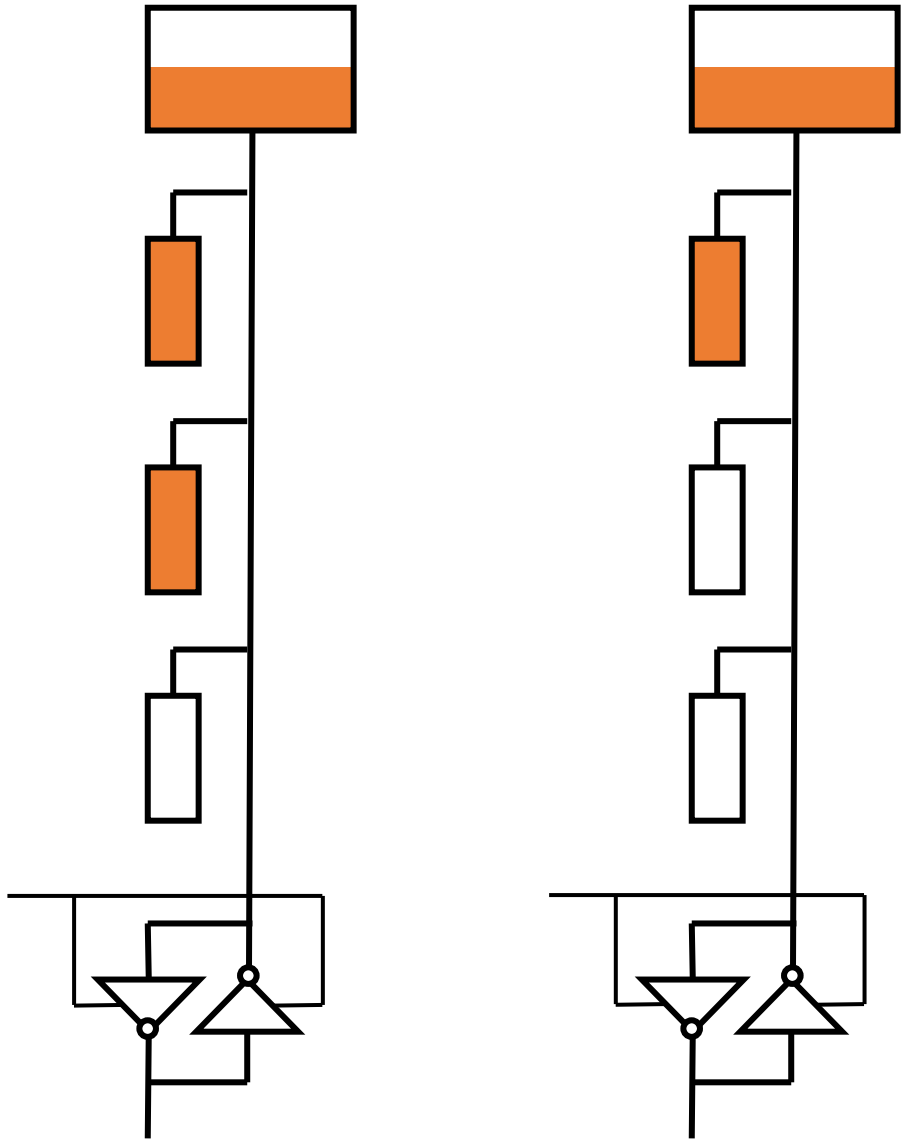


Logical Operations

Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology

MICRO 2017

- Open 3 rows at the same time
- Charge sharing forces line towards majority state



Logical Operations

AND

Open a constant 0 row in addition to the 2 operands.
Majority high is only possible if both operands are high.

OR

Open a constant 1 row in addition to the 2 operands.
Majority high is achieved if any of the operands are high.

Problems With Previous Ideas

- Requires redesign of memory or system architecture
  - DRAM is a highly optimized low margin business → Expensive to change

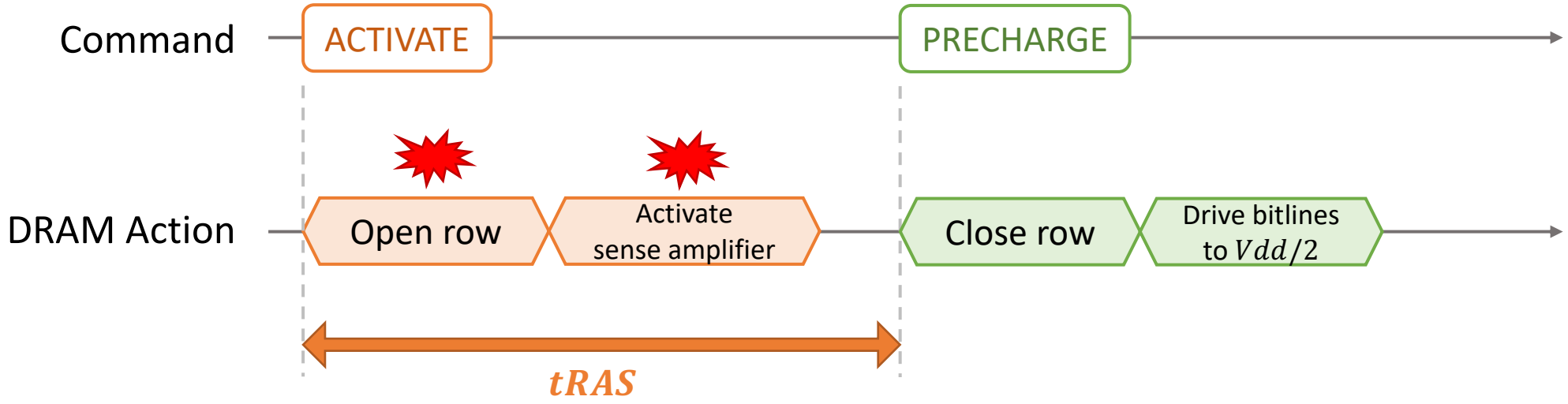
Novelty, Key Approach and Ideas

Novelty

- Is it possible to perform in-memory copy and AND/OR computations using standard off-the-shelf commodity DRAM?
- Provide a framework to allow for arbitrary computation using the available DRAM commands.

Key Approach

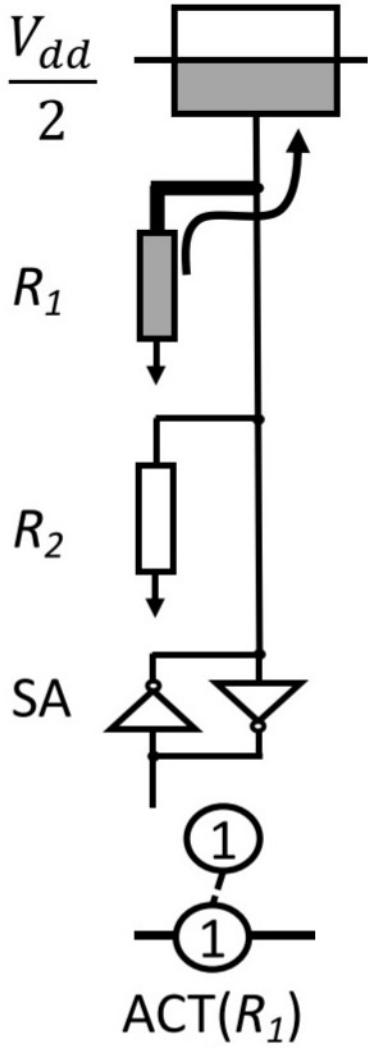
- Purposefully violate DRAM timing constraints to achieve non standard state which will perform in memory computations

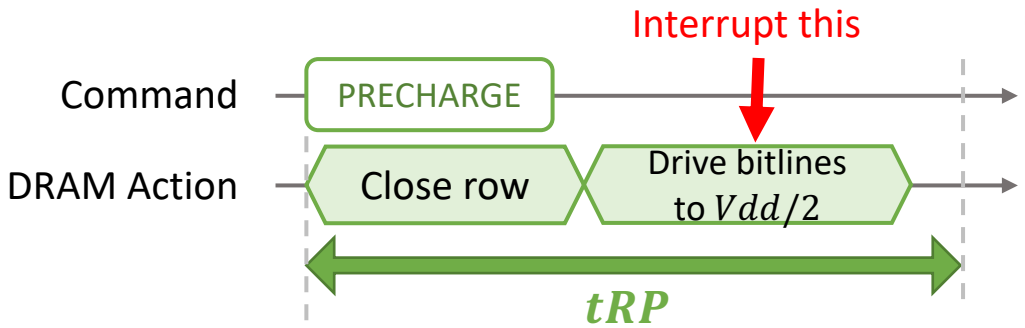


Mechanisms

Row Copy in Practice

- Precharge is required. Otherwise subsequent activate is ignored
- Make T2 short to prevent the bitline from being pulled to Vdd/2

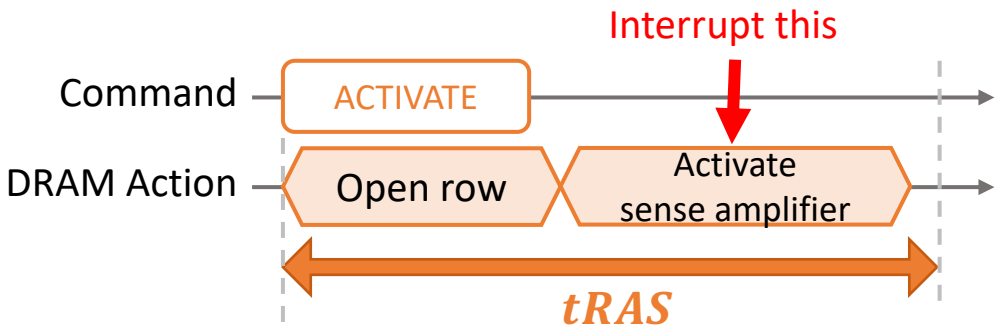




What About AND/OR?

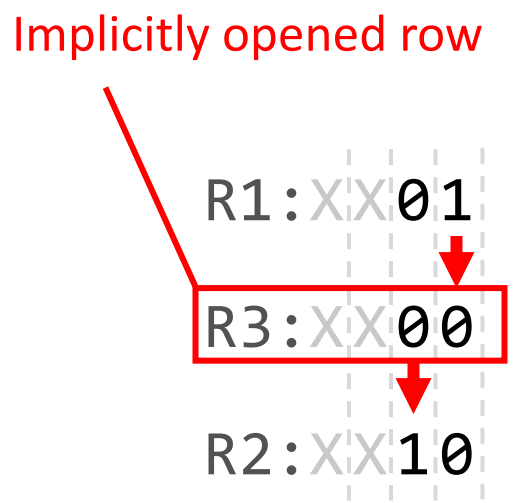
- Rows need to all be opened before sense amplifiers get activated
  - Row copy only has one row open at any time and sense amplifiers are activated in between

- Violate $tRAS$ to prevent the activation of the sense amplifiers



How to Open 3 Rows at the Same Time?

- While switching between selected rows, address needs to be updated
  - Select the right rows and be fast enough → intermediate rows get opened



Through testing the authors determined that address changes affect the opened rows as follows:

Considering the least significant 2 bits

R1 (From)	R2 (To)	R3 (Implicitly opened)
01	10	00
10	01	11

Charge Sharing

Ideally as along as all 3 rows are open the majority state should be the result

Reality however is different

- Rows aren’t all opened at the same time
  - The row that is opened first has the more time to affect the bitline

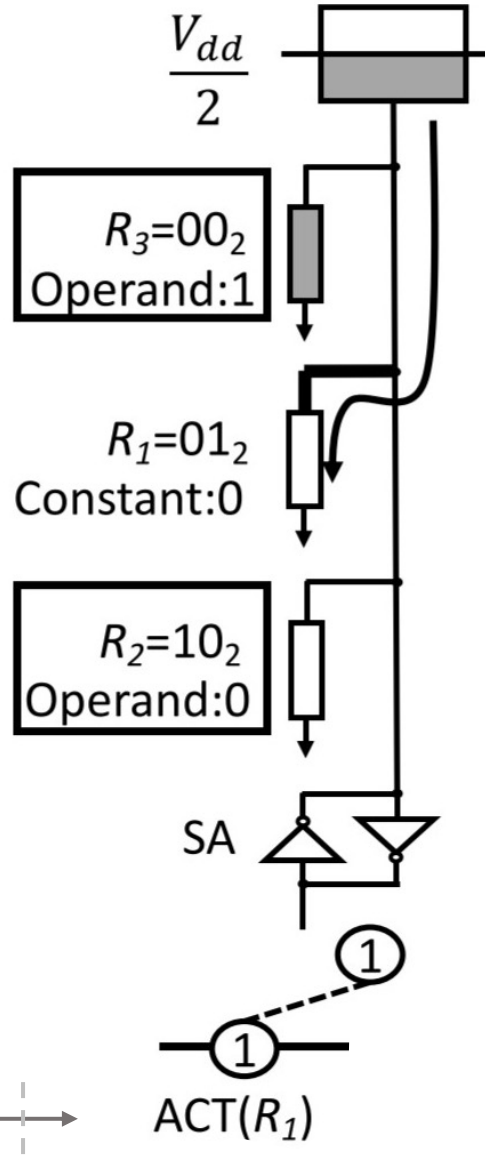
Charge Sharing Results

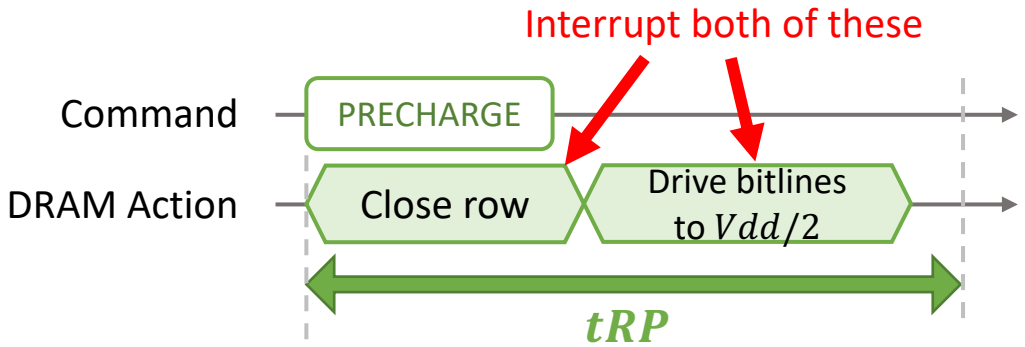
R3 \ R1 R2	00	01	10	11
0	0	0	X	1
1	0	1	1	1

Use this combination for OR between R2 and R3

Logical AND in Practice

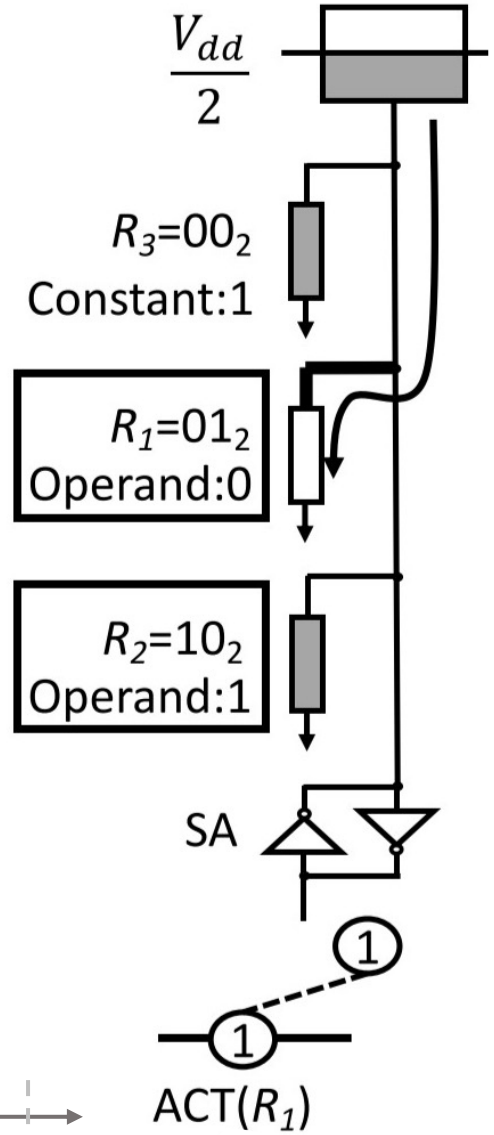
- Interrupt activation of sense amplifiers with low T1
- Prevent closing of the activated rows by having no delay between the necessary PRECHARGE and the next ACTIVATE

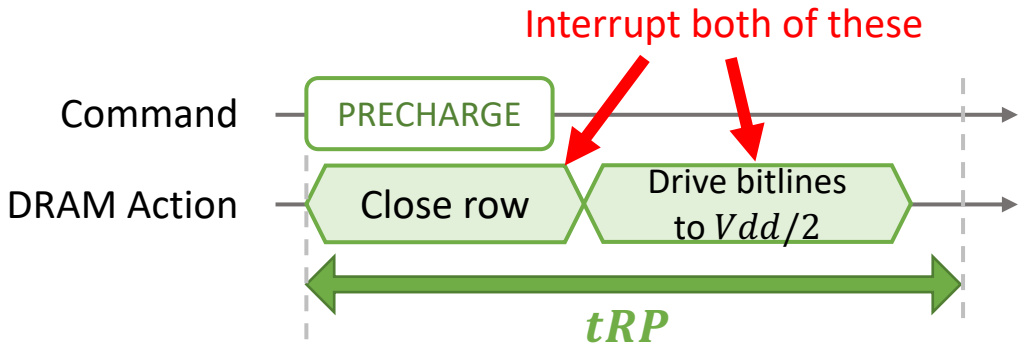




Logical OR in Practice

- Interrupt activation of sense amplifiers with low T1
- Prevent closing of the activated rows by having no delay between the necessary PRECHARGE and the next ACTIVATE





“Bad” Rows

- Some rows are unusable for in-memory operations
  - Manufacturing variations can cause problems with unequal capacitance
  - Row remapping is employed when some rows are faulty causing addresses to not match up
  - Deterioration during use can cause rows to become unusable even it they once were usable

Detect and exclude those rows from operations in software

Is This Enough for General Computation?

AND, OR are not enough to represent all formulas in boolean algebra

We need negation for functional completeness

The Negation Issue

It is possible to transform any formula with negation in such a way that the only negated terms are the inputs variables

see conjunctive normal form / disjunctive normal form

$$A \oplus B = (A \vee B) \wedge \overline{(A \wedge B)} = (\bar{A} \wedge B) \vee (A \wedge \bar{B})$$

As long as we have the negated values of the input operands we can compute anything in boolean algebra

→ Precompute this on the CPU

The Negation Issue

$$(A \vee B) \oplus B$$ We want to compute this directly without transformation

$$A \vee B = C \quad \rightarrow \quad C \oplus B$$ We need $\bar{C}$ to compute $C \oplus B$

$$\bar{C} = \overline{A \vee B} = \bar{A} \wedge \bar{B}$$

Require any value to have both its non inverted as well as its inverted representation available

$$\boldsymbol{val} = (\boldsymbol{v}, \bar{\boldsymbol{v}})$$

Any computation we do has to compute both its result as well as the inverse of that result

The Negation Issue

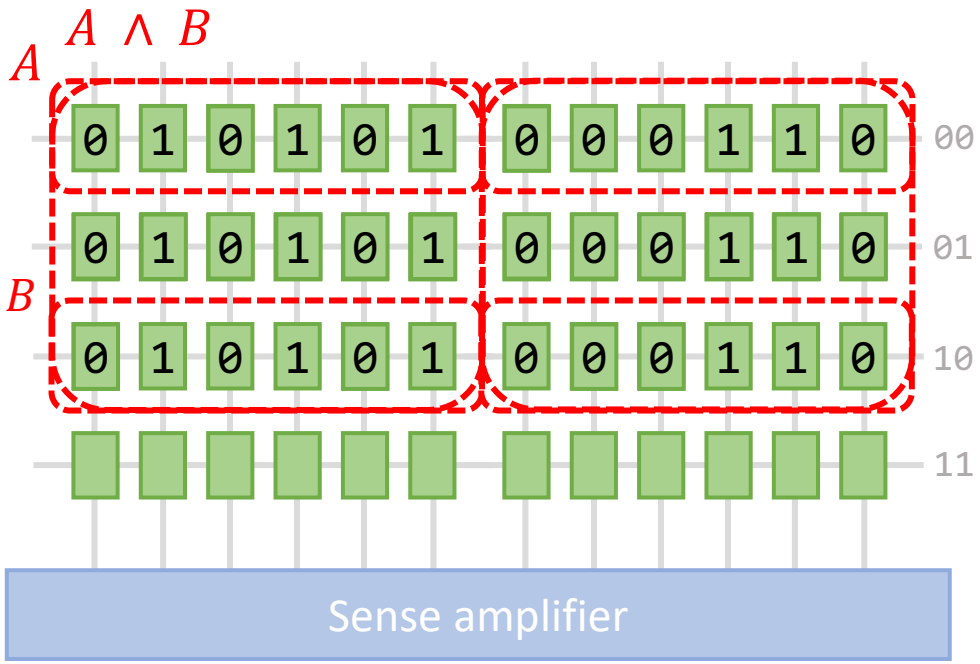
$$\begin{aligned}
\text{NOT: } \neg(A, \bar{A}) &= (\bar{A}, A) \\
\text{AND: } (A, \bar{A}) \wedge (B, \bar{B}) &= (A \wedge B, \bar{A} \vee \bar{B}) \\
\text{OR: } (A, \bar{A}) \vee (B, \bar{B}) &= (A \vee B, \bar{A} \wedge \bar{B}) \\
\text{NAND: } \neg((A, \bar{A}) \wedge (B, \bar{B})) &= (\bar{A} \vee \bar{B}, A \wedge B) \\
\text{XOR: } (A, \bar{A}) \oplus (B, \bar{B}) &= ((\bar{A} \wedge B) \vee (A \wedge \bar{B}), (\bar{A} \vee \bar{B}) \wedge (A \vee B))
\end{aligned}$$

The Negation Issue

- CPU needs to compute the inverse at the beginning
- Memory usage is doubled
- Every computation needs to calculate both the non inverted result as well as the inverted one.
  - Approximately double the computations are required
  - This can be improved by logical reduction of complex operations

Bit-Parallel Arithmetic

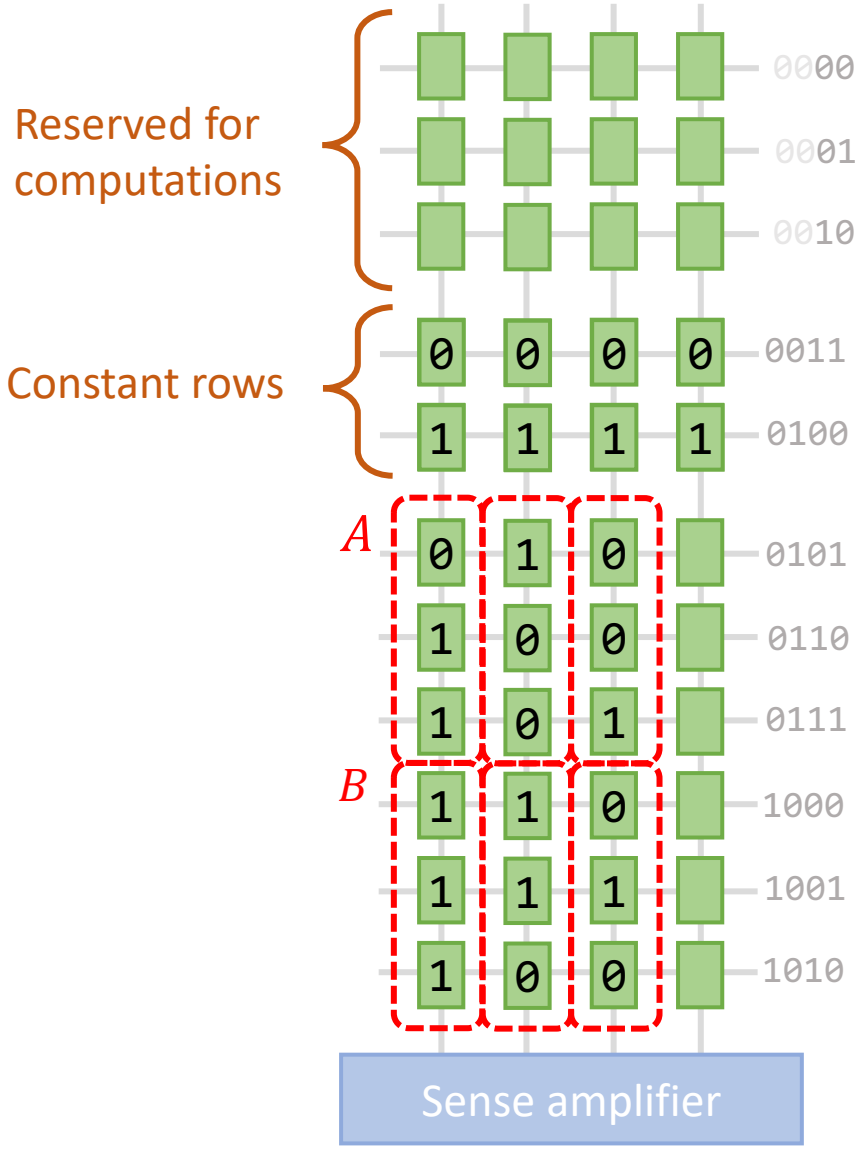
- Data is laid out in rows. Which is how data is typically laid out.
- Any operation is applied to all bits of a value in parallel
  - Can compute AND/OR of up to 8kB in one operation
  - Data can be directly accessed and used by the CPU
  - There is *no way* to perform bit shift which is needed for general purpose computations



This is a big problem

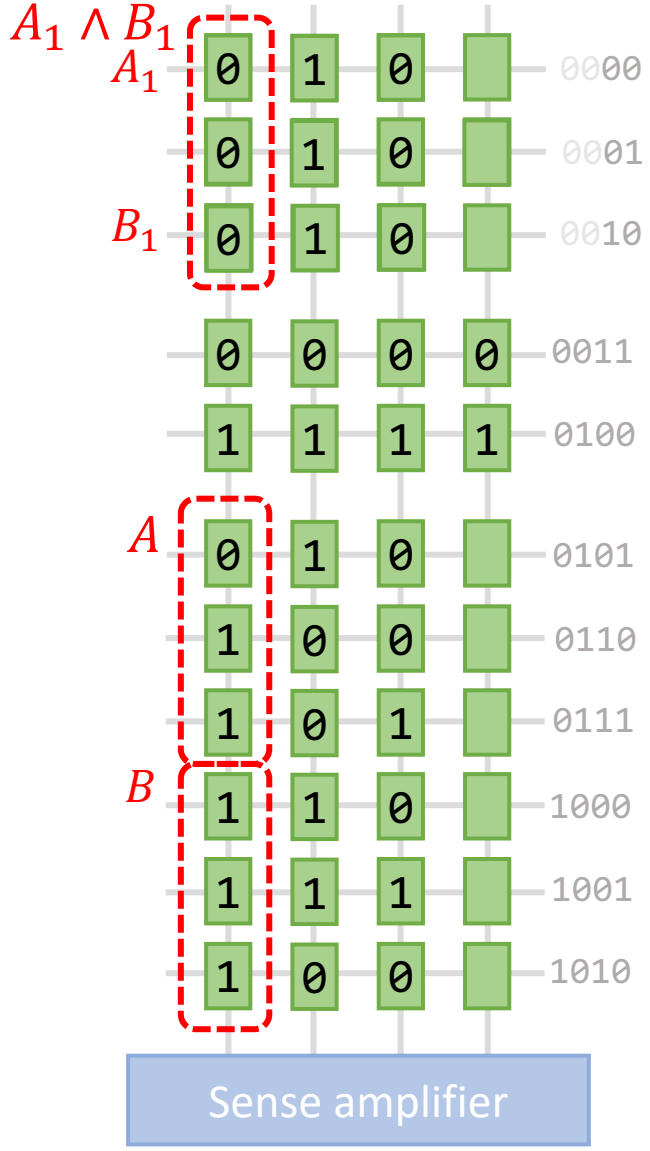
Bit-Serial Arithmetic

- Data is laid out in columns instead of rows



Bit-Serial Arithmetic

- Data is laid out in columns instead of rows
- Perform operations on each bit in a serial manner
  - Allows very fine grained control of operations
  - Can compute up to 64k values in parallel
  - Latency is significantly higher
  - Translation needed to read results back into the CPU (very expensive without special hardware)
  - Memory available is limited by number of rows



Constraints

- Operands are overwritten by operations
  - Copy needed before operation if operands are required again
- Operands must be in specific locations
  - Need to be in specific location relative to each other
- Bit-serial operations require extensive data level parallelism to be efficient
- All computations need to be in the same subarray
  - Inter subarray requires expensive copy through the memory controller and bus (what were trying to eliminate in the first place)
- Increased memory usage needed for arbitrary computations
  - Constant 0 and 1 rows are required
  - Need both inverted and non inverted data (can be optimized)

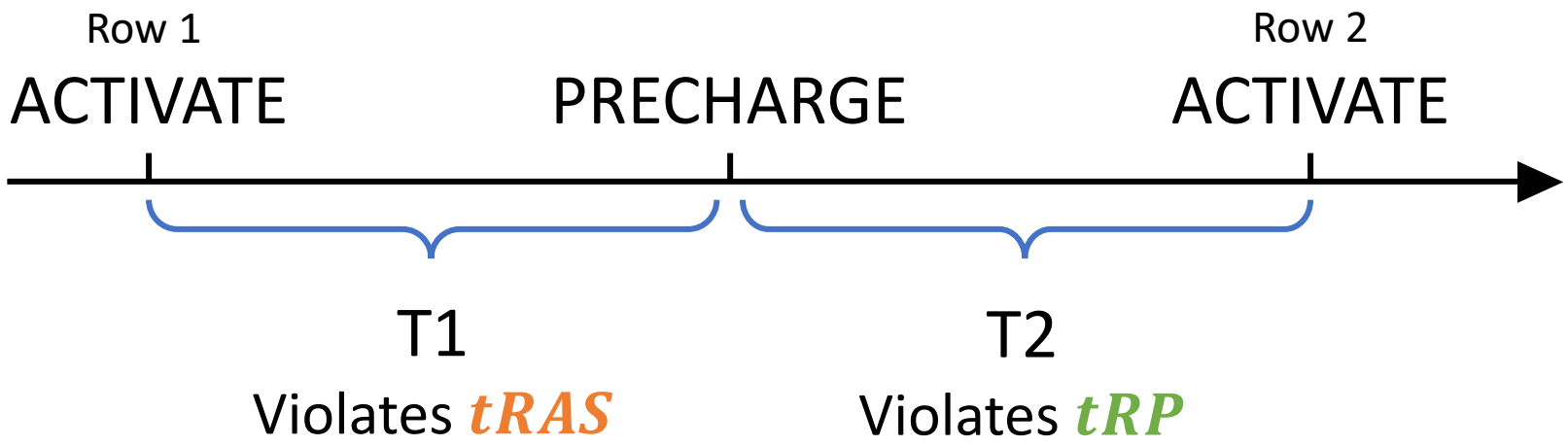
Methodology and Evaluation

Methodology

- Host system with FPGA acting as custom memory controller
- FPGA is running modified SoftMC, open source memory controller
- Data bus limited to either 400MHz or 800MHz
  - Timings in results are quantized to multiples of 2.5ns
- Issued instructions are buffered on the FPGA
  - Prevents PCIe bus latency issues

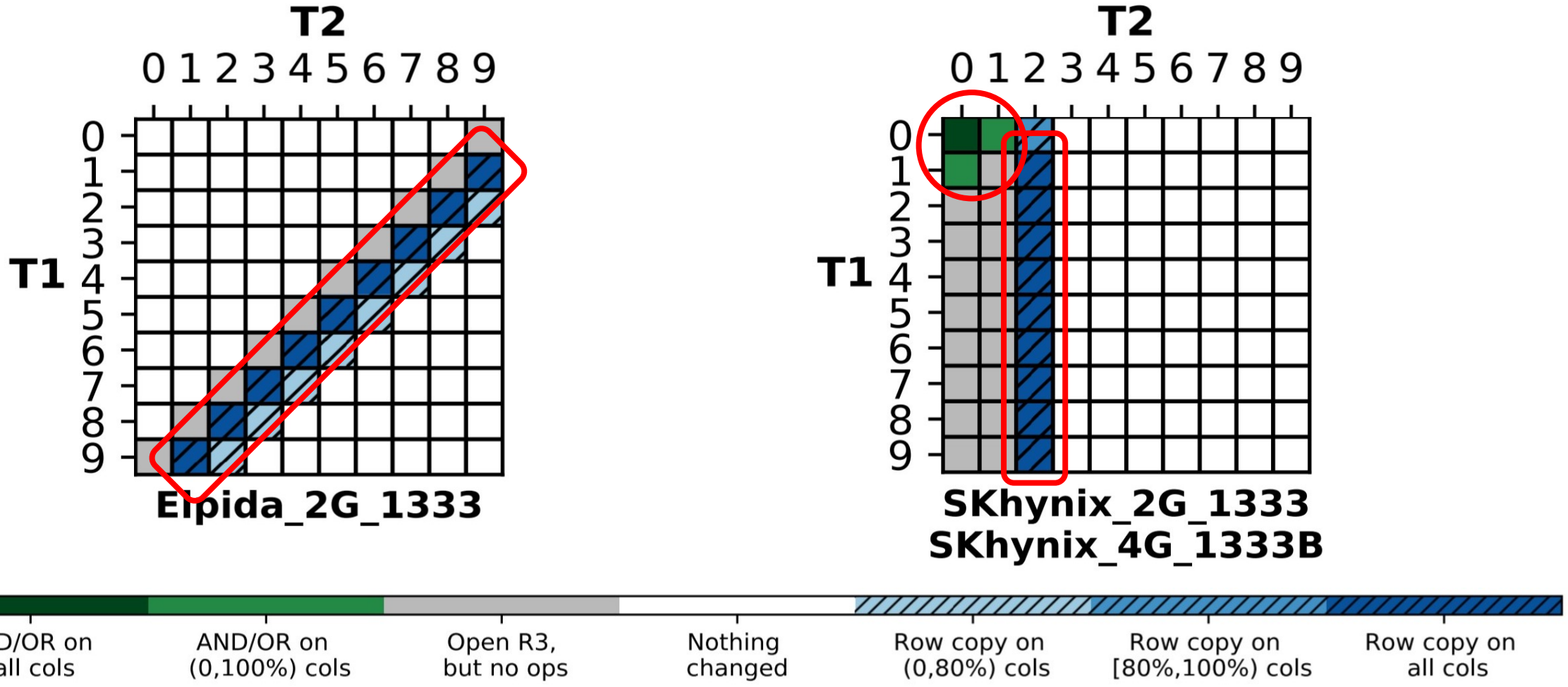
- 32 different DDR3 modules from 7 different vendors are evaluated.
- Reliability/feasibility is primary testing goal

Reminder

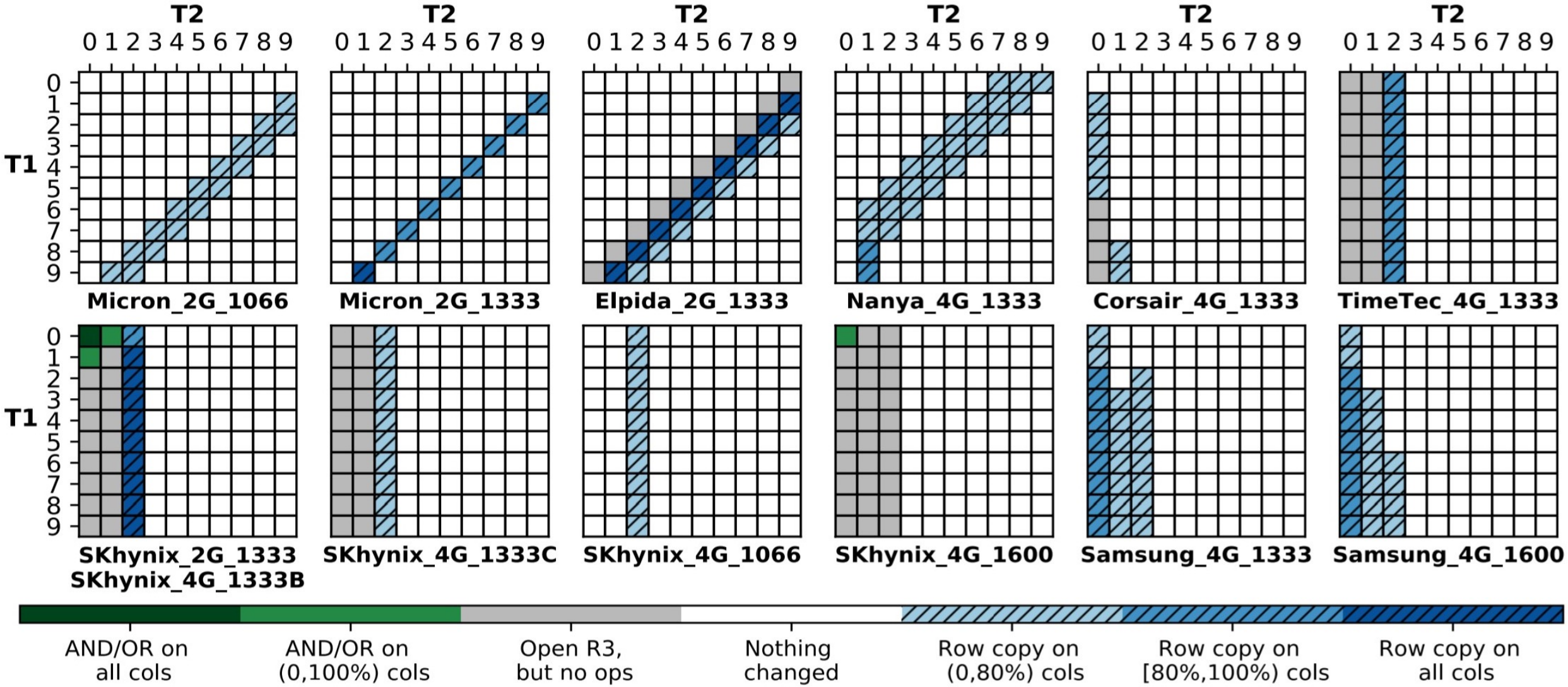


- Delay will be displayed as the number of idle clock cycles. 1 cycle is equivalent to 2.5ns

Results


T2
0 1 2 3 4 5 6 7 8 9
T1
0 1 2 3 4 5 6 7 8 9
Elpida_2G_1333
T2
0 1 2 3 4 5 6 7 8 9
T1
0 1 2 3 4 5 6 7 8 9
SKhynix_2G_1333
SKhynix_4G_1333B
AND/OR on all cols
AND/OR on (0,100%) cols
Open R3, but no ops
Nothing changed
Row copy on (0,80%) cols
Row copy on [80%,100%) cols
Row copy on all cols

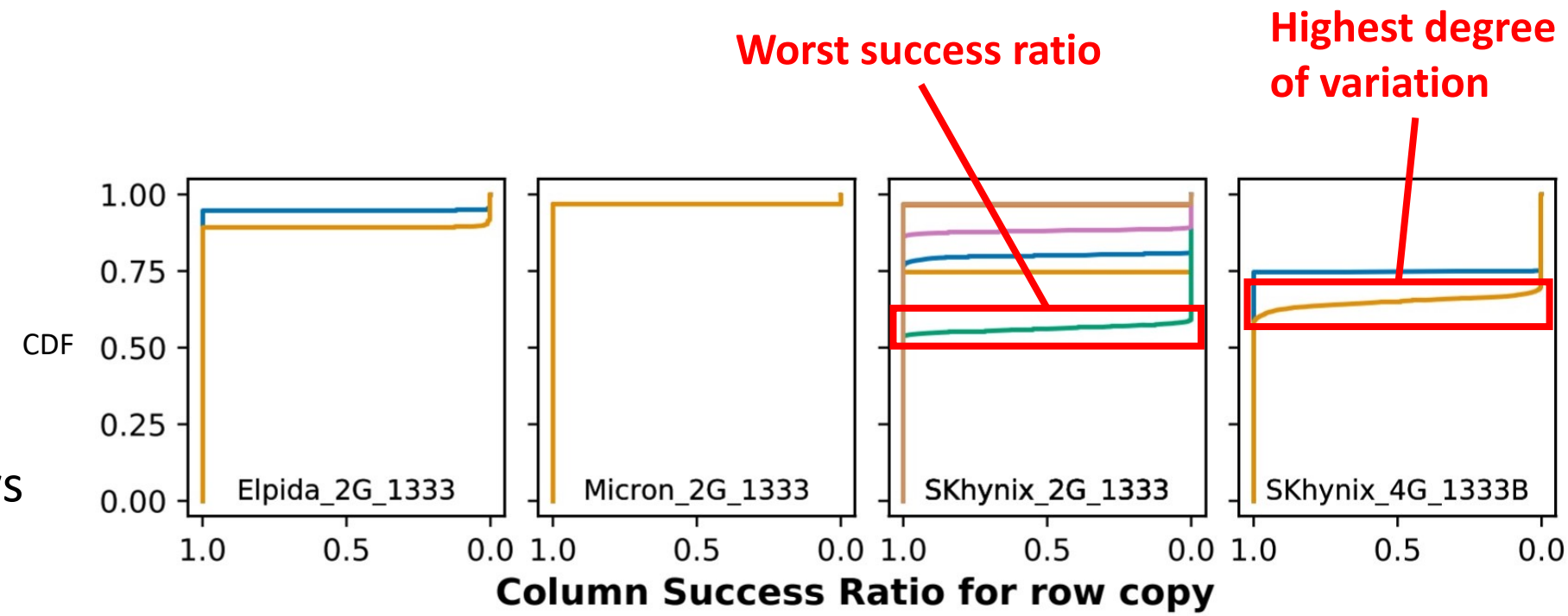
All Results


T2
0 1 2 3 4 5 6 7 8 9
T1
0 1 2 3 4 5 6 7 8 9
Micron_2G_1066
Micron_2G_1333
Elpida_2G_1333
Nanya_4G_1333
Corsair_4G_1333
TimeTec_4G_1333
SKhynix_2G_1333
SKhynix_4G_1333B
SKhynix_4G_1333C
SKhynix_4G_1066
SKhynix_4G_1600
Samsung_4G_1333
Samsung_4G_1600
AND/OR on all cols
AND/OR on (0,100%) cols
Open R3, but no ops
Nothing changed
Row copy on (0,80%) cols
Row copy on [80%,100%) cols
Row copy on all cols

Reliability

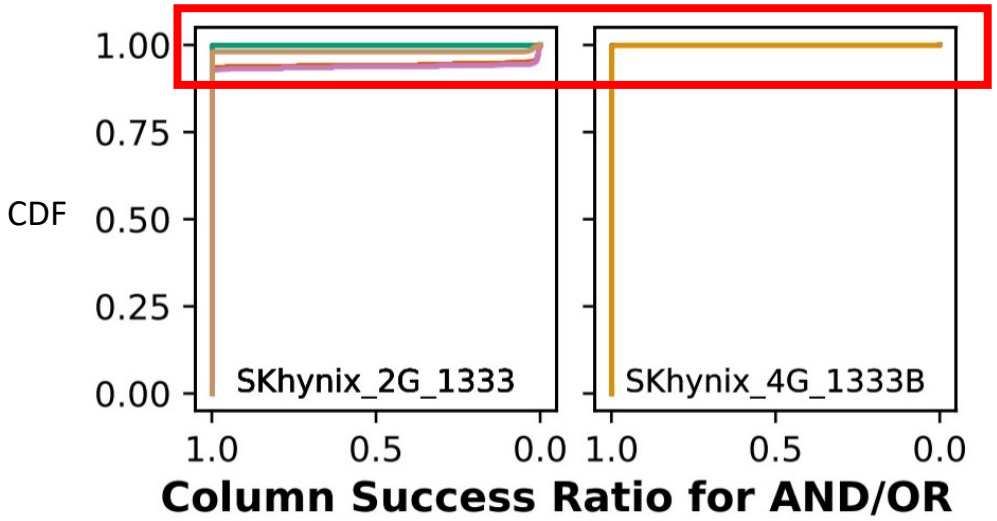
Fast success ratio drop is desirable

Makes it easy to determine suitable rows



If a cell is capable of performing a operation then reliability is very high

- 53.9% to 96.9% of rows have a 100% success rate when performing row copy
- 92.5% to 99.98% of rows have a 100% success rate when performing AND/OR



Performance

Number of clock cycles needed to perform the operation on 1 row (i.e. 1 bit)

Row copy	SHIFT	AND	OR	XOR	ADD
18	36	172	172	444	1332

- Throughput is dependent on high parallelism
- Assuming that all columns are used:
  - Row copy has a peak bandwidth of 182 GB/s
  - 8-bit AND/OR has a peak throughput of 19 GOPS
  - 8-bit ADD has a peak throughput of 2.46 GOPS

Energy Efficiency

- Energy usage was modelled using VAMPIRE
- Energy cost model assumes all DRAM commands finish
  - In this case though commands are interrupted
  - This is likely a conservative estimate

- 347x more energy efficient than vector unit for copying
- 48x more energy efficient than vector unit for 8-bit AND/OR
- 9.3x more energy efficient than vector unit for 8-bit ADD

Executive Summary

- **Problem**: Memory latency and bandwidth are a significant power and performance bottleneck in modern computing systems
- **Goal**: Perform in-memory computing using current off-the-shelf commodity DRAM and demonstrate a framework for arbitrary computations.
- **Key Approach**: Violate DRAM timing constraints to achieve non standard state
- **Evaluation**: Test feasibility and reliability of using 32 different modules from 7 different vendors. Performance and efficiency of AND/OR and 8-bit ADD are tested on modules that can reliably perform the operations.
- **Results**:
  - Nearly all chips can perform row copy on at least some rows
  - 3 modules were able to perform AND/OR operations
  - If an operation is possible then the reliability of the operation is high (between 92.5% to 99.98% of rows that can perform AND or OR have 100% success rate)
  - Peak throughput of 19 GOPS for 8-bit AND/OR and 2.46 GOPS for 8-bit ADD

Strengths of the Paper

- Almost no changes in DRAM are required
- Creative approach to solve the negation issue
- Allows highly efficient and performant parallel computations
- Memory can be used for both computing and storage
  - Very flexible as subarrays can be switched between being used for computing and storage
- Compiler optimizations could reduce many of the inefficiencies in the proposed solutions

- If nothing else this paper illustrated that the mechanisms presented by RowClone and Ambit are reliable and feasible

Weaknesses

- Only intra subarray computations are efficient
- End to end integration is required
  - Significant redesign of memory controllers are needed
  - System and application software need to implement the features
- Operations on many independent data is required for efficiency
  - What happens before and after one computation batch?
- Need to have negated values available
  - Doubled memory usage
  - Precomputation is needed

- Paper is unfortunately worded implying they invented the row copy and AND/OR operation mechanism. (They did prove it works without modifications)

Thoughts and Ideas

Enable Inter Subarray Bulk Data Copy

- Reduces the impact of duplicate data arising from negation framework
  - Computations are no longer limited to 253 bits since multiple subarrays can now be used for storage

Low-Cost Inter-Linked Subarrays (LISA):Enabling Fast Inter-Subarray Data Movement in DRAM

Add Translation Logic to Memory Controller

- Let memory controller handle translation between bit-serial and bit-parallel representation
  - No need for expensive translation on CPU

Remove Subarrays That Are Used for Computation From the Addressable Memory Pool

- Eliminates cache coherence issues
- CPU can’t process bit-serial data anyways so not much is lost

- With added effort this can be made dynamic

Discussion

Is it worth adding special purpose negation logic to DRAM over the presented framework?

What kind of software changes need to happen to use this?

Is it worth it for developers to support this in addition to specialized accelerators?

How widely applicable is this mechanism?

What optimizations could be enabled by performing operations in a bit-serial manner?