Adaptive Batch Update in TCAM (ABUT) : How Collective Optimization Beats Individual Ones

Ying Wan[1], Haoyu Song[2], Yang Xu[3], Chuwen Zhang[1], Yi Wang[4,5], and Bin Liu[1,5]

[1]Tsinghua University, China, [2]Futurewei Technologies, USA
[3]Fudan University, China [4]Southern University of Science and Technology, China
[5]Peng Cheng Laboratory, China

Email: wany16@mails.Tsinghua.edu.cn

Outline

- Background
- Related work
- Two key algorithms of ABUT:
  - Incremental rule grouping
  - Optimal TCAM placement calculation
- Performance evaluation
- Conclusion

Background —— TCAM for rule tables

- **The *de facto* industry standard for rule tables**
  - line-speed lookup speed
  - flexible matching pattern
- **The placed rules are arranged in priority order**
  - TCAM returns the first matched rule
  - It is the highest-priority matched rule that counts

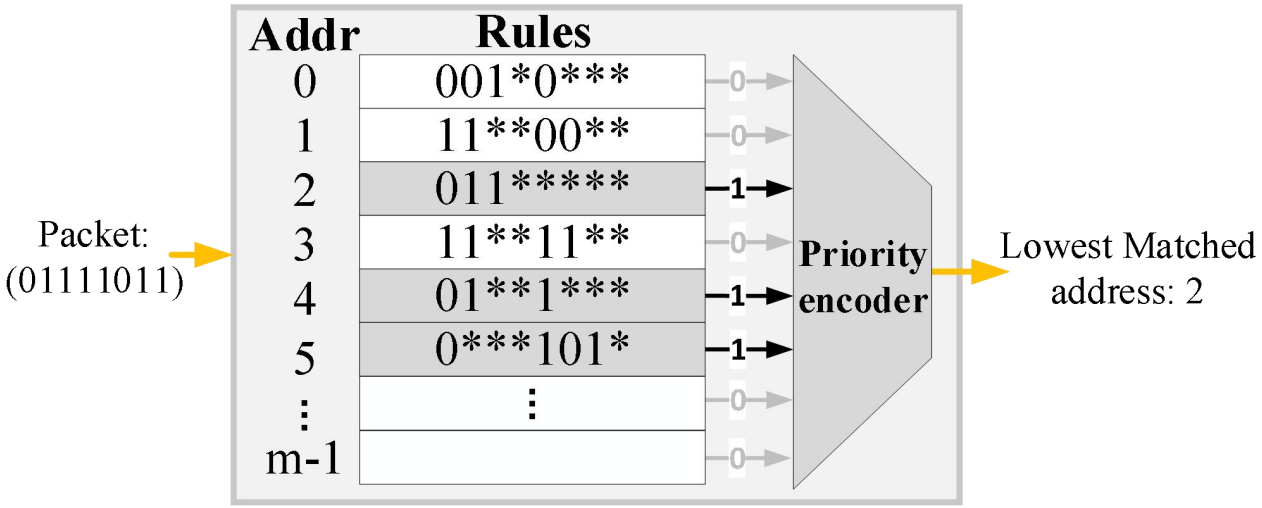

Fig.1. The mechanism of TCAM

Background —— TCAM update problem

- **A new rule insertion incurs the moves of existing rules**
  - rules must be placed by the topological order of the rule relation graph
  - design algorithms to compute the update scheme with fewer rule moves
- **Batch TCAM update requirements**
  - A high-level policy in SDN and IDN can be converted to multiple TCAM rules
  - TCAM applications present a batch update pattern (e.g., TCAM as a cache)
  - Conventional applications requires batch rule table updates (TE, IP source guard)
  - Individual updates accumulate to form a batch for switch (HP-5406, Pica-3290)

Fig.2. The rule overlapping relationship and the rule relation (DAG)

Background — challenges of TCAM update

- **Placement cost (t^p)**
  - Proportional to the number of rule moves
  - A large t^p indicates the long interrupt of the TCAM lookup (packet loss)

- **Compute cost (t^c)**
  - The time overhead to compute the rule move scheme
  - A large t^c fails to sustain the update requests (throughput and latency)

- Exist works either optimize t^p or t^c for individual update
  - Cumulative placement and compute cost

Related works

- Individual update algorithms
  - Per-group priority: PLO(HOTI'2000), FFU(Globecom 2006)
  - Per-rule priority
    - Single chain: Cao(HOTI'2000), Γ_{cao}(TON'2018)
    - Hybrid chain: FastRule(JSAC'2019)
    - Range chain: RuleTris(ICDCS'2016), Γ_{bh}(TON'2018)
- "Semi-Batch" update algorithms:
  - CoPTUA(TOC'2004), Hermes(CoNext'2017), COLA(INFOCOM'2020)

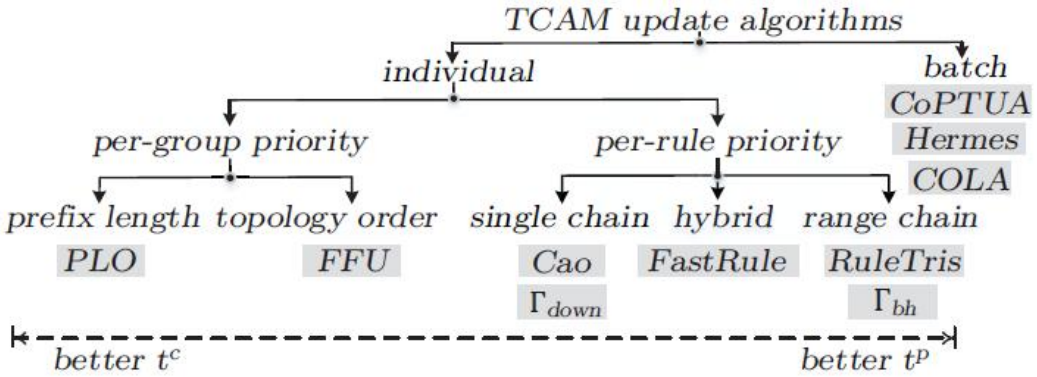

Fig. 3. Classification of TCAM update algorithms.

ABUT——The target and architecture

- Target:
  - A batch contains n rules to be inserted

$$T_b^c < \sum_{i=1}^{n} t_i^c, \quad T_b^p < \sum_{i=1}^{n} t_i^p,$$

  - $\{T_b^c, T_b^p\}$ and $\{\sum_{i=1}^{n} t_i^c, \sum_{i=1}^{n} t_i^p\}$ is the batch and individual update cost
- Architecture
  - Rule grouping algorithm
    - Incremental grouping rules
  - Batch update algorithms
    - placing rules according to their grouping IDs (gid)
    - dynamic programming to compute the optimal placement

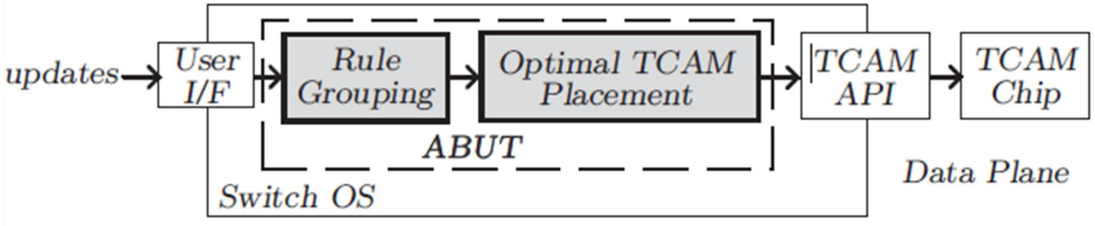

Fig.4. The architecture of TCAM

ABUT——incremental rule regrouping

- topological-order based rule grouping
  - optimal in terms of the number of resulting groups

$$r.gid = \begin{cases} 0 & r \ has \ no \ successor \\ max\{r'.gid\} + 1 & r' \in successors \ of \ r \end{cases} \quad (1)$$

- regroup the rules that are possibly changed due to updates
  - regroup rules in their priority order
  - the change of $A.gid$ only affect $B.gid$(B is the predecessor of A)
    - **insertion rule *A***: regroup A
    - **deleted rule *A***: regroup B if and only if $B.gid == A.gid + 1$
    - **regrouped rule *A*** ($A.gid$ is changed from x to y):
      - If $x > y$: regroup B if and only if $B.gid <= y$
      - If $x < y$: if and only if $B.gid == x + 1$

Example——incremental rule regrouping

- **DAG**: rule relation graph
  - add the new rules ($E,\ F_0,\ F_1$), mark the new rules
  - remove the rules to be deleted ($C_0,\ C_1,\ C_2$), mark its specific predecessors(B)
- **L_R:** a link list that link the rules in their priority reverse order
  - Regroup a rule after all its successors has been assigned the correct GIDs
- Only regroup the encountered marked rules
  - mark its s

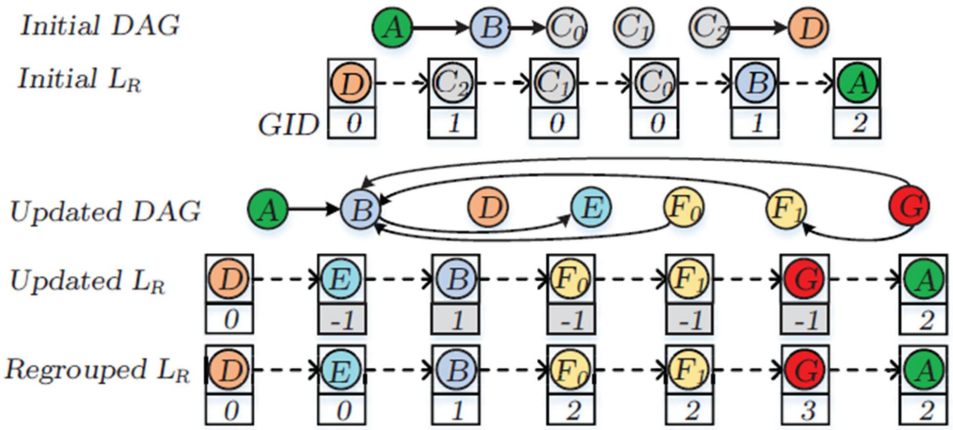

Fig.5. An example of the fast incremental regrouping

ABUT——optimal TCAM placement calculation

- Rules are placed in their GID orders(The legitimate placement N_L):
  - $m,\ n,\ k,\ N_i$ is the number of TCAM entries, rules, groups, rules of $i-th$ group, respectively

$$N_L = (\prod_{i=0}^{k-1} N_i!) * \binom{n}{m} = (\prod_{i=0}^{k-1} N_i!) * \frac{m!}{n!(m-n)!} \qquad (2)$$

- dynamic programming to minimize placement cost (write and nullify operations)
  - $T[i].gid,\ R[j].gid$: the GID of the rule in the entry $T[i]$ and the (j-1)-th rule
  - $C[i][j]$: the minimum cost if $T[0:i-1]$ are used to place $R[0:j-1]$.

$$\mathrm{C}[i][j] = min \begin{cases} \mathrm{C}[i\text{-}1][j] + (\mathrm{T}[i\text{-}1].gid \neq \text{-}1) \\ \mathrm{C}[i\text{-}1][j\text{-}1] + (\mathrm{T}[i\text{-}1].gid \neq \mathrm{R}[j\text{-}1].gid) \end{cases} \qquad (3)$$

  - Adaptive empty entry distribution without extra cost
    - The two ways in Eq.3 give the same cost

Example——optimal TCAM placement calculation

- Placement cost
  - The minimized TCAM write and nullify operations
- Batch updates
  - Delete $C0$, $C1$, and $C2$
  - Insert E, $F0$, and $F1$
- any path from $C[0][0]$ to $C[m][n]$ represents an optimal placement.

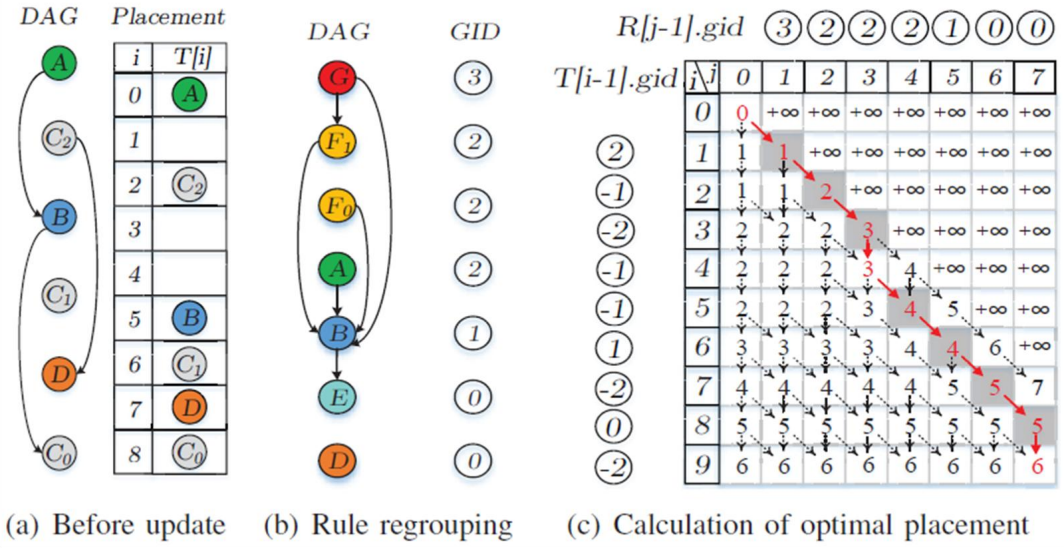

Fig.6. An example of dynamic programming for optimal TCAM placement

Performance evaluation——Experiment setup

- **Compare objects**
  - Single Chain (SC)[1]
  - Range Chain (RC)[1]
  - Batch update: COLA_SC, COLA_RC
- **Metric**
  - Rule grouping performance
  - Effect of empty entry distribution
  - Performance on LPM and multi-field rule tables
  - Scalability on batch size (θ), TCAM capacity (m), TCAM fill-rate (δ)
- **Dataset**
  - CAIDA ($cd1{\sim}cd10$) and Stanford($sf1,\ sf2$)
  - ACL($acl1{\sim}acl5$) and FW($fw1{\sim}fw5$)
  - Openflow rules ($of1,\ of2$)

TABLE I
RULE TABLES USED FOR PERFORMANCE EVALUATION

Type	Name	Source	Feature	Field #	Size
LPM	cd1 - cd10	CAIDA	real	1	~1M
LPM	sf1, sf2	Stanford	real	1	~90K
ACL	acl1 - acl5	ClassBench	synthetic	5	10K
Firewall	fw1 - fw5	ClassBench	synthetic	5	10K
IP Chain	ipc1, ipc2	ClassBench	synthetic	5	10K
Openflow	of1, of2	ClassBench-ng	synthetic	9	10K

Performance evaluation——Experiment results

- **Rule groups**

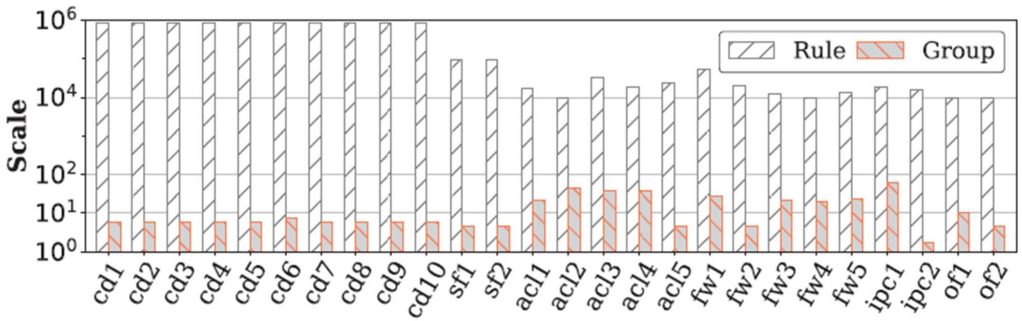

Fig.7. The number of rules vs. the number of groups

- **Time consumption**

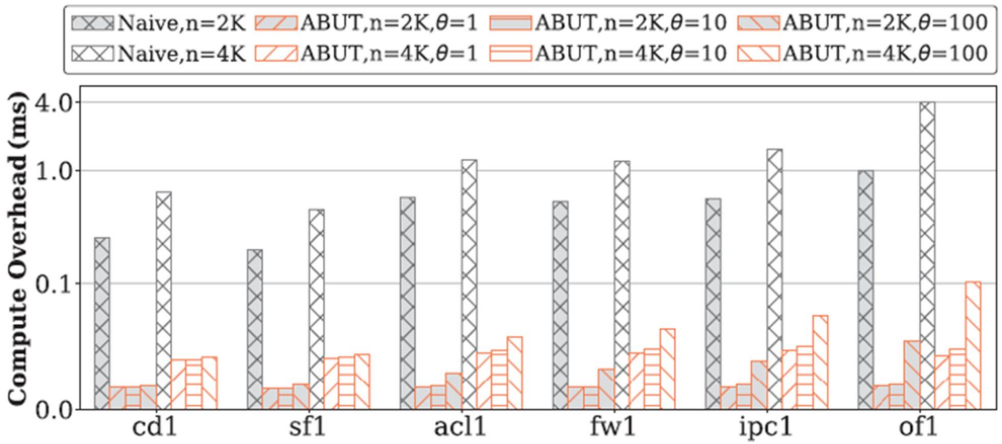

Fig.8. naïve regrouping vs. ABUT’ s incremental regrouping

Performance evaluation——Experiment results

- **Effect of empty entry distribution**

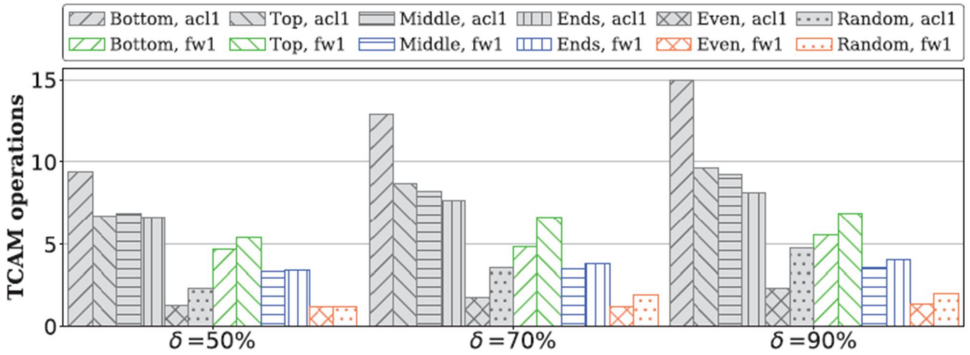

Fig.9. empty distribution strategies under m=4K and θ =50

- **Performance on LPM tables (TCAM as a cache)**

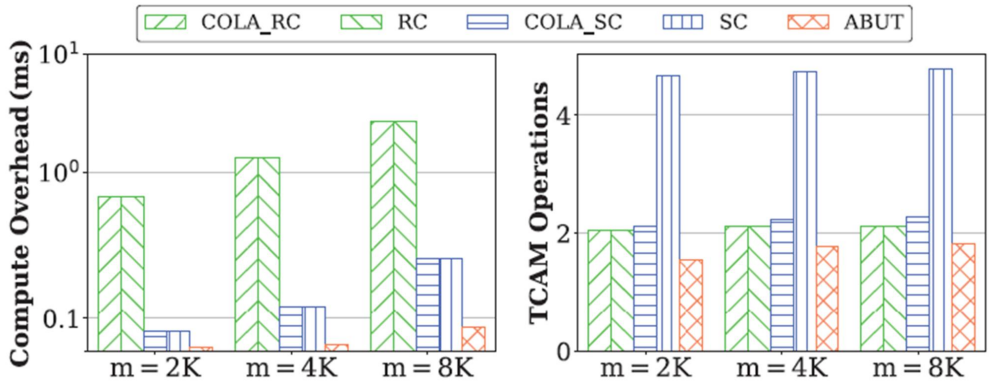

Fig.10. Update performance on LPM tables

Performance evaluation——Experiment results

- **Performance on multi-field rule tables**

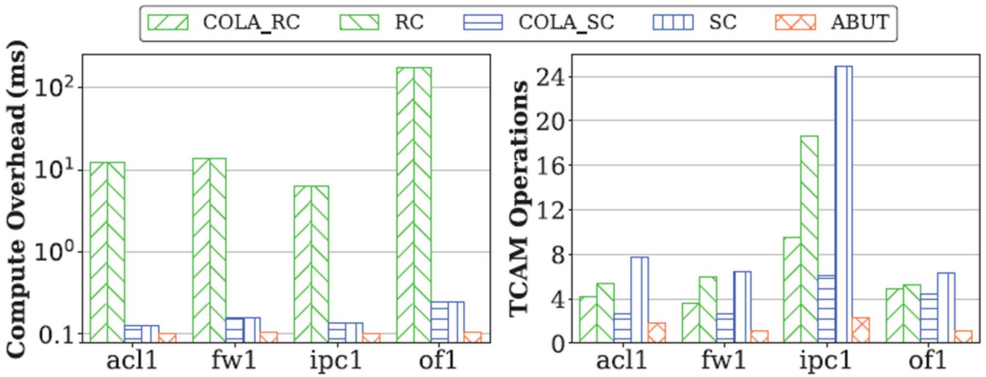

Fig.11. Update performance on multi-field tables

- **Scalability on batch size (θ)**

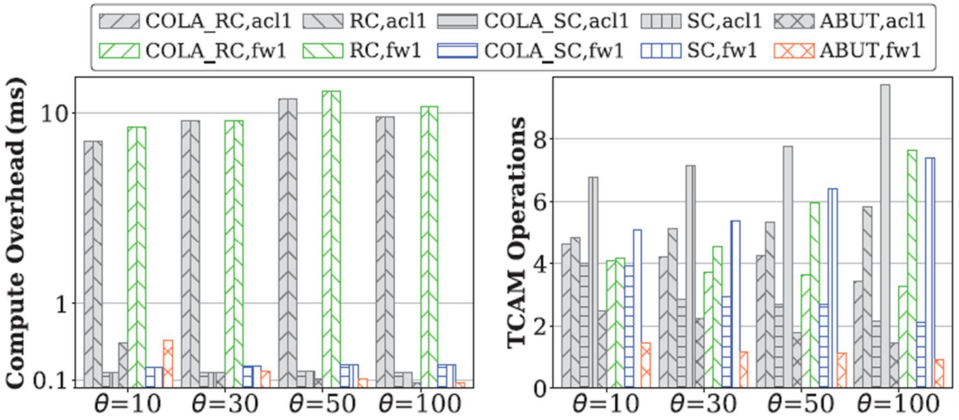

Fig.12. Update performance with θ for m =4K and δ=80%

Performance evaluation——Experiment results

- **Scalability on TCAM size (m)**

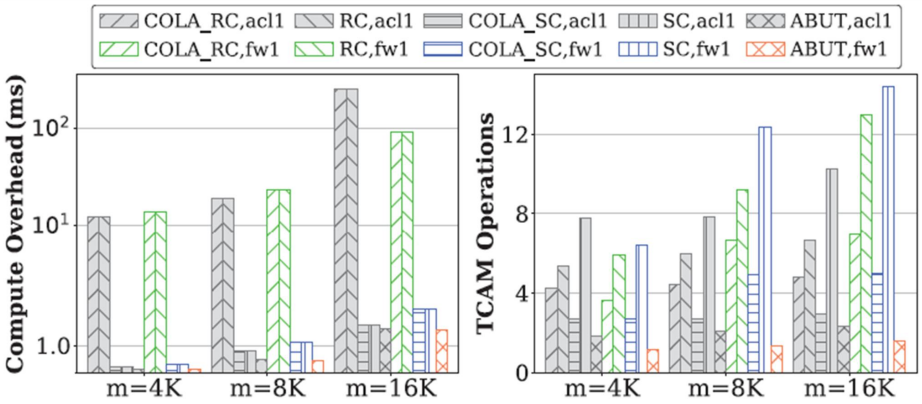

Fig.13. Update performance with m for θ =50 and δ=80%

- **Scalability on TCAM fill-rate (δ)**
  - $\delta < 100\%$, inserte θ=50 rules
  - $\delta = 100\%$, randomly delete θ=50 rules before inserting θ=50 rules

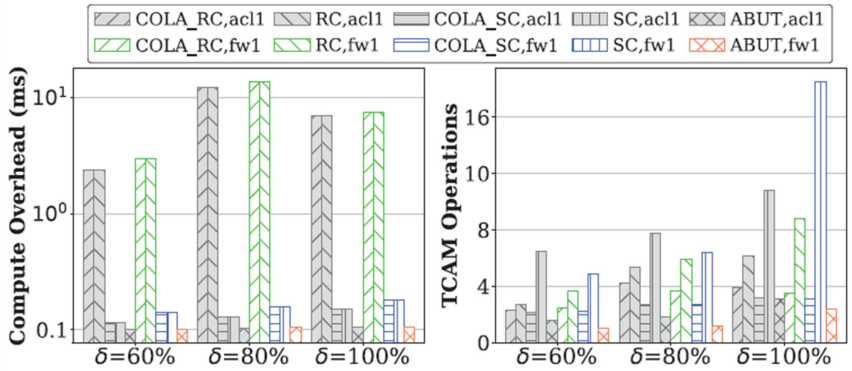

Fig.14. Update performance with δ for θ =50 and m=4K

Conclusion

- ABUT is the first true TCAM batch update algorithm
  - Grouping rules and maintain group orders
  - Dynamic programming for optimal TCAM placement
  - Adaptive empty entry distribution

Thank You!

Q & A