Advanced Topics in Multicore Architecture & Software Systems

Out-of-Order Execution

Orientation

First half of the course: background

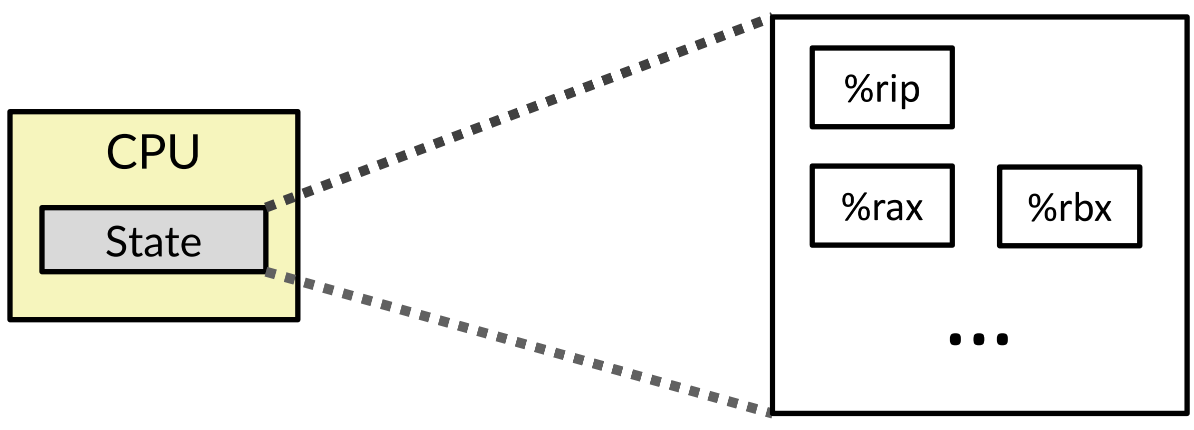
- Processors (**out-of-order**) and speculative execution
- Correctness of concurrent algorithms (linearizability)
- Cache coherence
- Memory consistency (processors & PL)

Processor state

The CPU state has a well-defined structure, which can be manipulated by instructions

State is mainly represented as *registers* (say, 64-bit in size)

- Program counter (PC aka IP or RIP in x86)
- RAX, RBX, … (in x86)



Instruction set architecture

Definition of possible state manipulations, for example:

- `ADD rX, rY, rZ` rX = rY + rZ
- `BRANCH  rC, rD` if (rc != 0) PC = rD

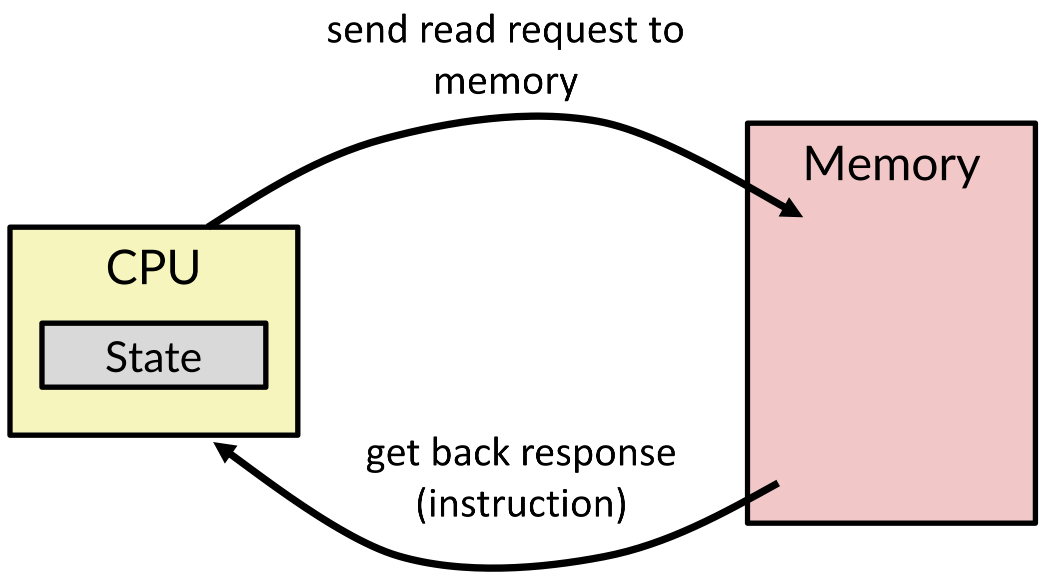
Plus small number of instructions for communicating with "outside" the state machine:

- `LOAD rA, rV` rV = Memory[rA]
- `STORE rA, rV` Memory[rA] = rV

Program execution

Processor state machine algorithm is simply:

1. Read next instruction from the memory address stored in the PC
2. Advance PC to next instruction ("PC += size-of-instruction")
3. Perform the instruction (= change processor state)
4. Go to 1



Parallelizing sequential code

Main observation behind CPU performance advances:
sequential code has parallelism

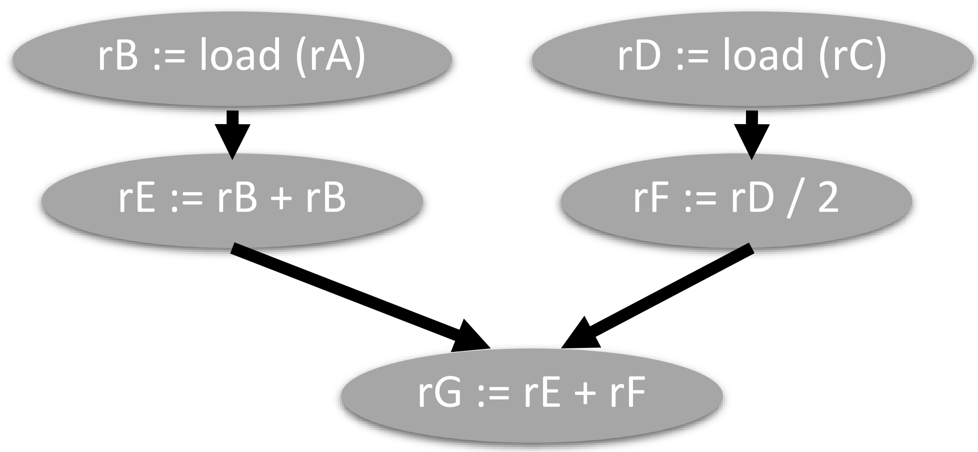
rB := load(rA)

rD := load(rC)

rE := rB + rB

rF := rD / 2

rG := rE + rF



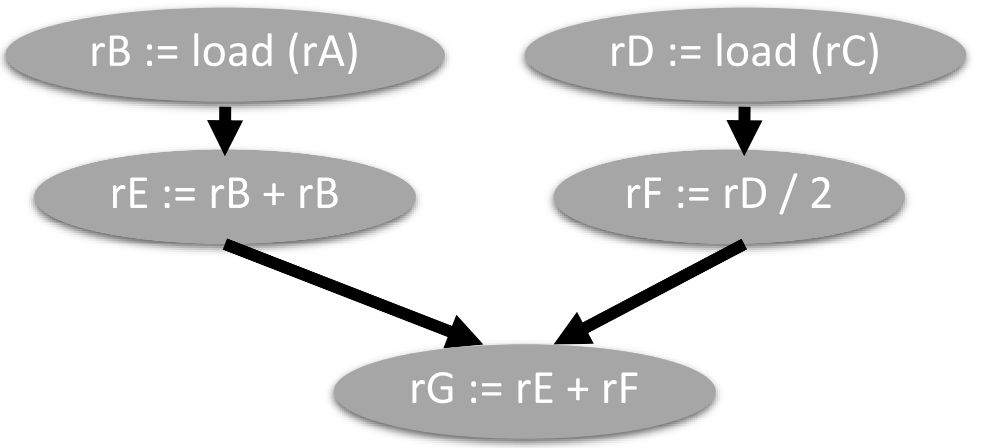
instruction-level parallelism (ILP)

Parallelizing sequential code

Instr B **data-depends (A->B)** on instr *A* if:

- the output of A is an input (operand) of B
- for some C, A->C and C->B (transitivity)

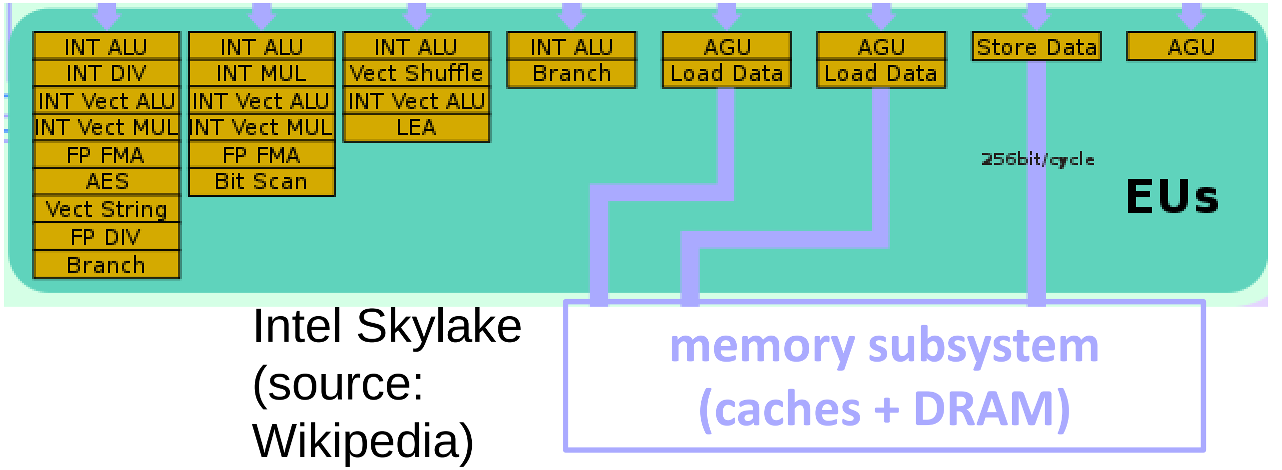
Data-flow graph:



Parallelizing sequential code

CPU can execute instructions without data dependencies in parallel:

- By having multiple execution units (e.g., ADDs)
- By having multiple memory operations in-flight
- Etc.

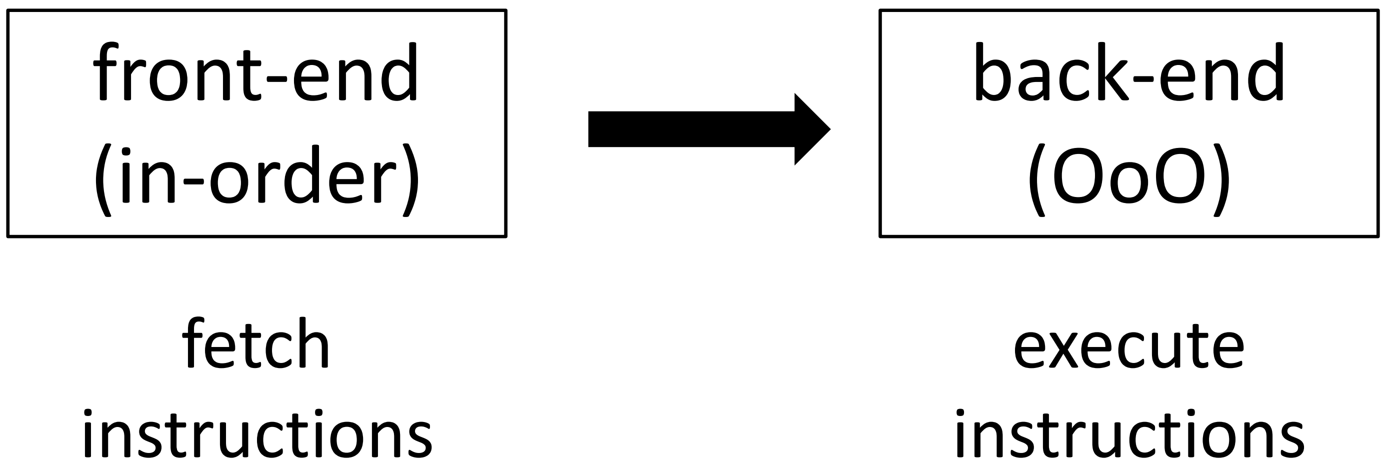


⇒ **Out-of-order (OoO) execution**

Intel Skylake (source: Wikipedia)

OoO execution

CPU builds the data-flow graph implicitly, as instructions are fetched



Breaking dependencies

Instr B **data-depends (A->B)** on instr *A* if:

- the output of A is an input (operand) of B
- for some C, A->C and C->B (transitivity)

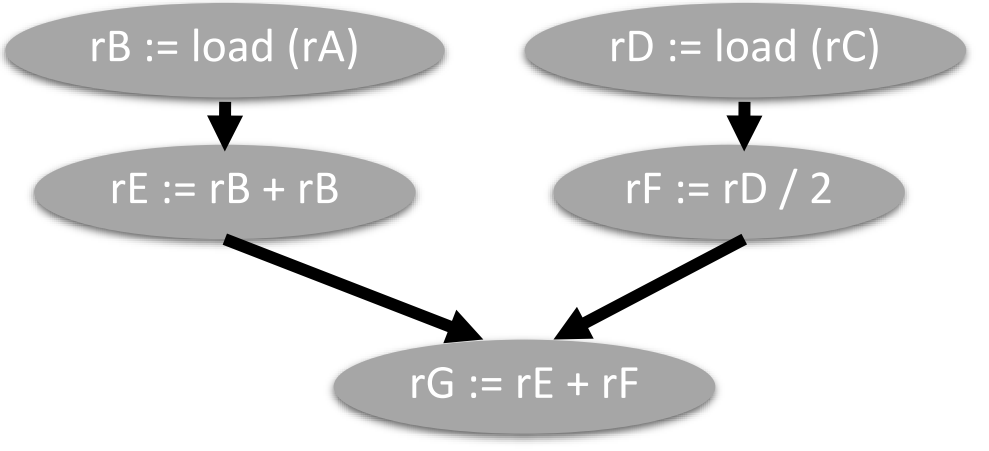
rB := load (rA)

rD := load(rC)

rE := rB + rB

rF := rD / 2

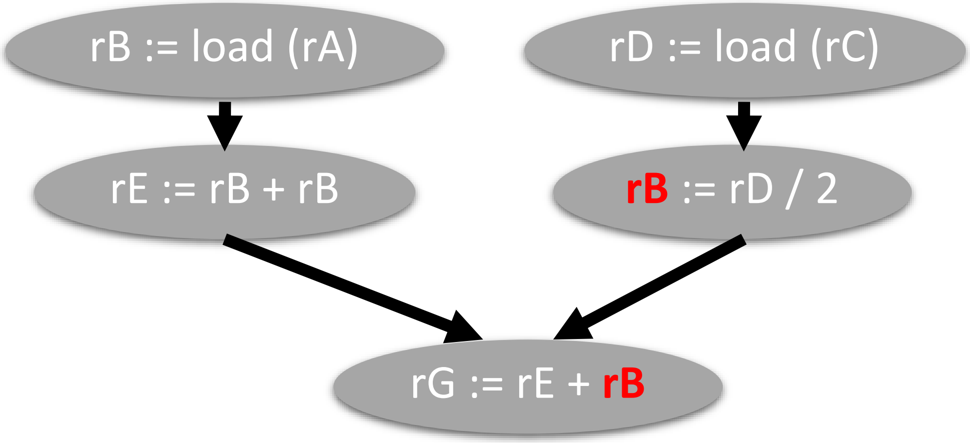
rG := rE + rF



Breaking dependencies

Instr B **data-depends (A->B)** on instr *A* if:

- the output of A is an input (operand) of B
- for some C, A->C and C->B (transitivity)



rB := load (rA)

rD := load(rC)

rE := rB + rB

rB := rD / 2

rG := rE + **rB**

Register renaming

CPU builds the data-flow graph implicitly, as instructions are fetched

Implicit graph built using **register renaming**. CPU has more physical registers than logical (ISA) registers. Instructions entering the back-end have their registers mapped to physical registers, which captures the data-dependencies.

Instructions are tracked in a reorder buffer (ROB) until execution

Register renaming

register mapping table		
rA: r0	rD: r40	rG: r75
rB: r1	rE: r22	
rC: r8	rF: r17	

register file

	data / ready
r0:	0xf00 / 1
r1:	0xbar / 1
r2:	0xbee / 1
r3:	
r4:	
r5:	
r6:	
r7:	
r8:	0xc00f / 1
…	

Register renaming

register mapping table

rA: r0	rD: r40	rG: r75
rB: r1	rE: r22	
rC: r8	rF: r17	

register file

data / ready

r0: 0xf00 / 1
r1: 0xbar / 1
r2: 0xbee / 1
r3:
r4:
r5:
r6:
r7:
r8: 0xc00f / 1
…

fetched

rB := load (rA)

ROB

Register renaming

register mapping table

rA: r0	rD: r40	rG: r75
rB: **r2**	rE: r22	
rC: r8	rF: r17	

register file

data / ready

r0: 0xf00 / 1
r1: 0xbar / 1
r2: ------- / 0
r3:
r4:
r5:
r6:
r7:
r8: 0xc00f / 1
...

fetched

rB := load (rA)

ROB

r2 := load (r0)

Can execute
(input is available)

Register renaming

register mapping table

rA: r0	rD: r40	rG: r75
rB: **r2**	rE: r22	
rC: r8	rF: r17	

fetched

rB := load (rA)

rD := load(rC)

ROB

r2 := load (r0)

register file

	data / ready
r0:	0xf00 / 1
r1:	0xbar / 1
r2:	------- / 0
r3:	
r4:	
r5:	
r6:	
r7:	
r8:	0xc00f / 1
…	

Register renaming

register mapping table

rA: r0	rD: **r3**	rG: r75
rB: **r2**	rE: r22	
rC: r8	rF: r17	

fetched

rB := load (rA)

rD := load(rC)

ROB

r2 := load (r0)

r3 := load (r8)

register file

data / ready

r0: 0xf00 / 1
r1: 0xbar / 1
r2: ------- / 0
r3: ------- / 0
r4:
r5:
r6:
r7:
r8: 0xc00f / 1
...

Register renaming

register mapping table

rA: r0	rD: **r3**	rG: r75
rB: **r2**	rE: r22	
rC: r8	rF: r17	

fetched

rB := load (rA)

rD := load(rC)

rE := rB + rB

ROB

r2 := load (r0)

r3 := load (r8)

register file

data / ready

r0: 0xf00 / 1

r1: 0xbar / 1

r2: ------- / 0

r3: ------- / 0

r4:

r5:

r6:

r7:

r8: 0xc00f / 1

…

Register renaming

register mapping table

rA: r0	rD: **r3**	rG: r75
rB: **r2**	rE: **r4**	
rC: r8	rF: r17	

register file

data / ready

r0: 0xf00 / 1
r1: 0xbar / 1
r2: ------- / 0
r3: ------- / 0
r4: ------- / 0
r5:
r6:
r7:
r8: 0xc00f / 1
...

fetched

rB := load (rA)
rD := load(rC)
rE := rB + rB

ROB

r2 := load (r0)
r3 := load (r8)
r4 := r2 + r2

Can't execute (input isn't available, can be executed once r2 becomes ready

Scheduling logic picks instructions to execute in each cycle

Register renaming

register mapping table

rA: r0	rD: **r3**	rG: r75
rB: **r2**	rE: **r4**	
rC: r8	rF: r17	

fetched

rB := load (rA)

rD := load(rC)

rE := rB + rB

rB := rD / 2

ROB

r2 := load (r0)

r3 := load (r8)

r4 := r2 + r2

register file

data / ready

r0: 0xf00 / 1

r1: 0xbar / 1

r2: ------- / 0

r3: ------- / 0

r4: ------- / 0

r5:

r6:

r7:

r8: 0xc00f / 1

…

Register renaming

register mapping table

rA: r0	rD: **r3**	rG: r75
rB: **r5**	rE: **r4**	
rC: r8	rF: r17	

register file

data / ready

r0: 0xf00 / 1
r1: 0xbar / 1
r2: ------- / 0
r3: ------- / 0
r4: ------- / 0
r5: ------- / 0
r6:
r7:
r8: 0xc00f / 1
...

fetched

rB := load (rA)
rD := load(rC)
rE := rB + rB
rB := rD / 2

ROB

r2 := load (r0)
r3 := load (r8)
r4 := r2 + r2
r5 := r3 / 2

Register renaming

register mapping table		
rA: r0	rD: **r3**	rG: r75
rB: **r5**	rE: **r4**	
rC: r8	rF: r17	

register file

data / ready

r0: 0xf00 / 1
r1: 0xbar / 1
r2: ------- / 0
r3: ------- / 0
r4: ------- / 0
r5: ------- / 0
r6:
r7:
r8: 0xc00f / 1
...

fetched

rB := load (rA)
rD := load(rC)
rE := rB + rB
rB := rD / 2
rG := rE + rB

ROB

r2 := load (r0)
r3 := load (r8)
r4 := r2 + r2
r5 := r3 / 2

Register renaming

register mapping table

rA: r0	rD: **r3**	rG: **r6**
rB: **r5**	rE: **r4**	
rC: r8	rF: r17	

fetched

rB := load (rA)

rD := load(rC)

rE := rB + rB

rB := rD / 2

rG := rE + rB

ROB

r2 := load (r0)

r3 := load (r8)

r4 := r2 + r2

r5 := r3 / 2

r6 := r4 + r5

register file

data / ready

r0: 0xf00 / 1

r1: 0xbar / 1

r2: ------- / 0

r3: ------- / 0

r4: ------- / 0

r5: ------- / 0

r6: ------- / 0

r7:

r8: 0xc00f / 1

...

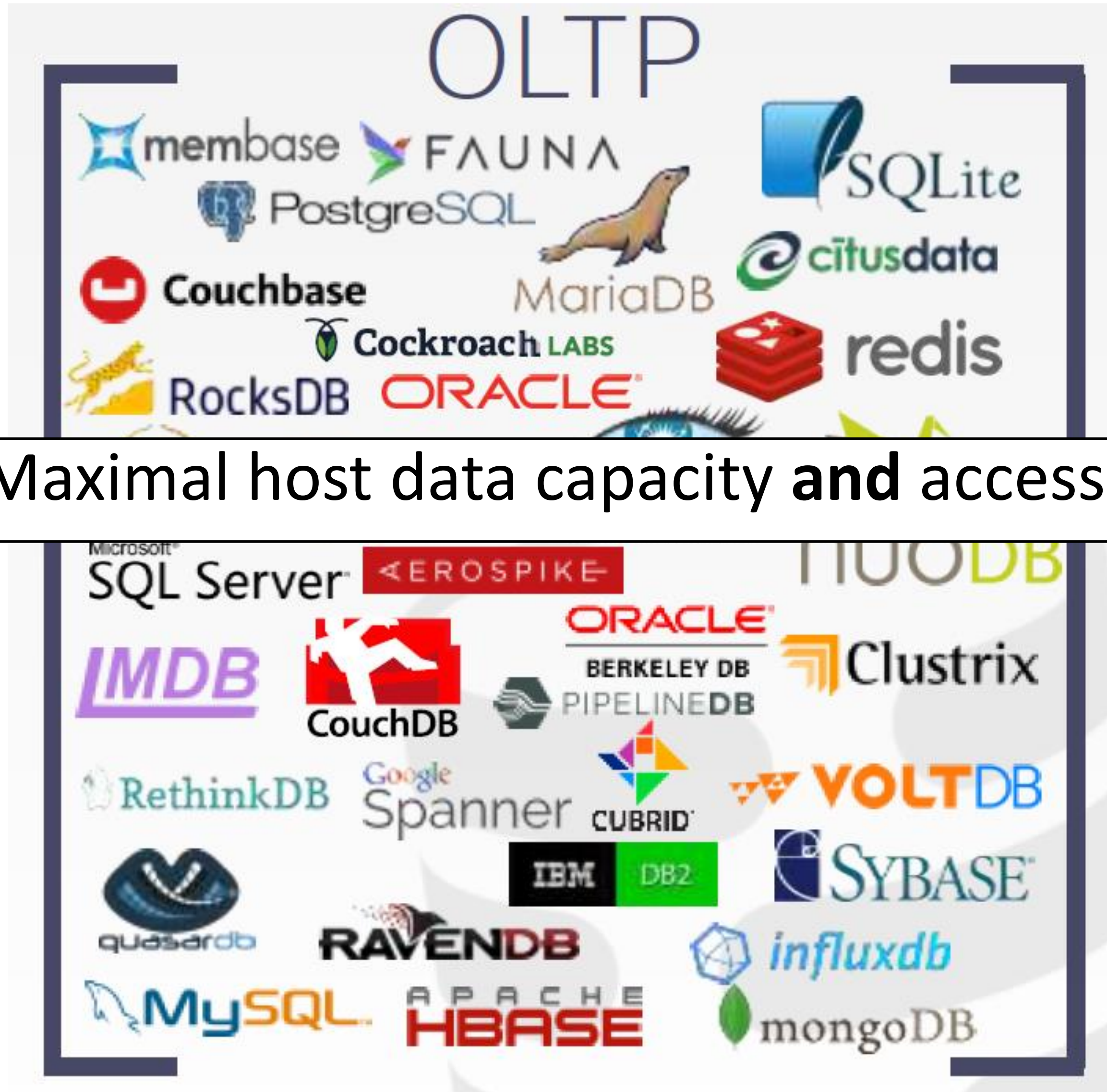
Benefitting from OoO

This part

Classic data structures are based on simple computation models, that don’t take OoO execution into account

We will see how OoO execution of memory operations, which is called **memory-level parallelism (MLP)** can be used to break a **fundamental** data structure speed barrier and speed up data access

Motivation: data deluge



Want: Maximal host data capacity **and** access speed

Outline

Running example: in-memory DB ordered index

- $\boxed{\text{Definition}}$
- Challenges
- Opportunity: MLP
- **Cuckoo Trie**: MLP-first index design
- What’s next

In-memory ordered index

Ordered index:

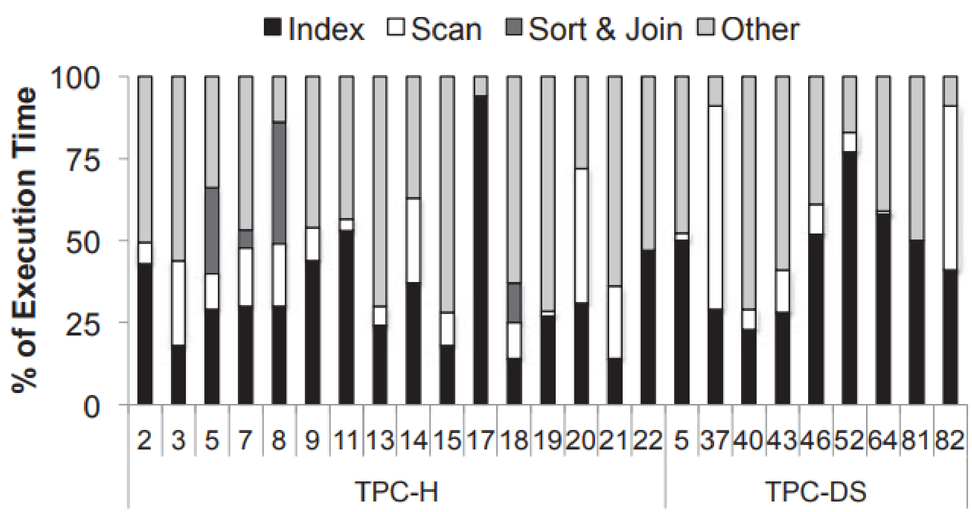
- Mapping of *keys* to *values* (e.g., DB rows)
- Insert, delete, and lookup of keys
- Range scan: list all keys from *start* to *end* in sorted order

Example: “List cities w/ population between 1M and 3M’’

Indexes should be fast

DB queries can spend up to 94% (35% on average) of their time on indexing.

[Kocberber et al., MICRO-46]



Total execution time breakdown

access speed
(faster=better)

Indexes should be memory efficient

Index size can equal/exceed dataset size!

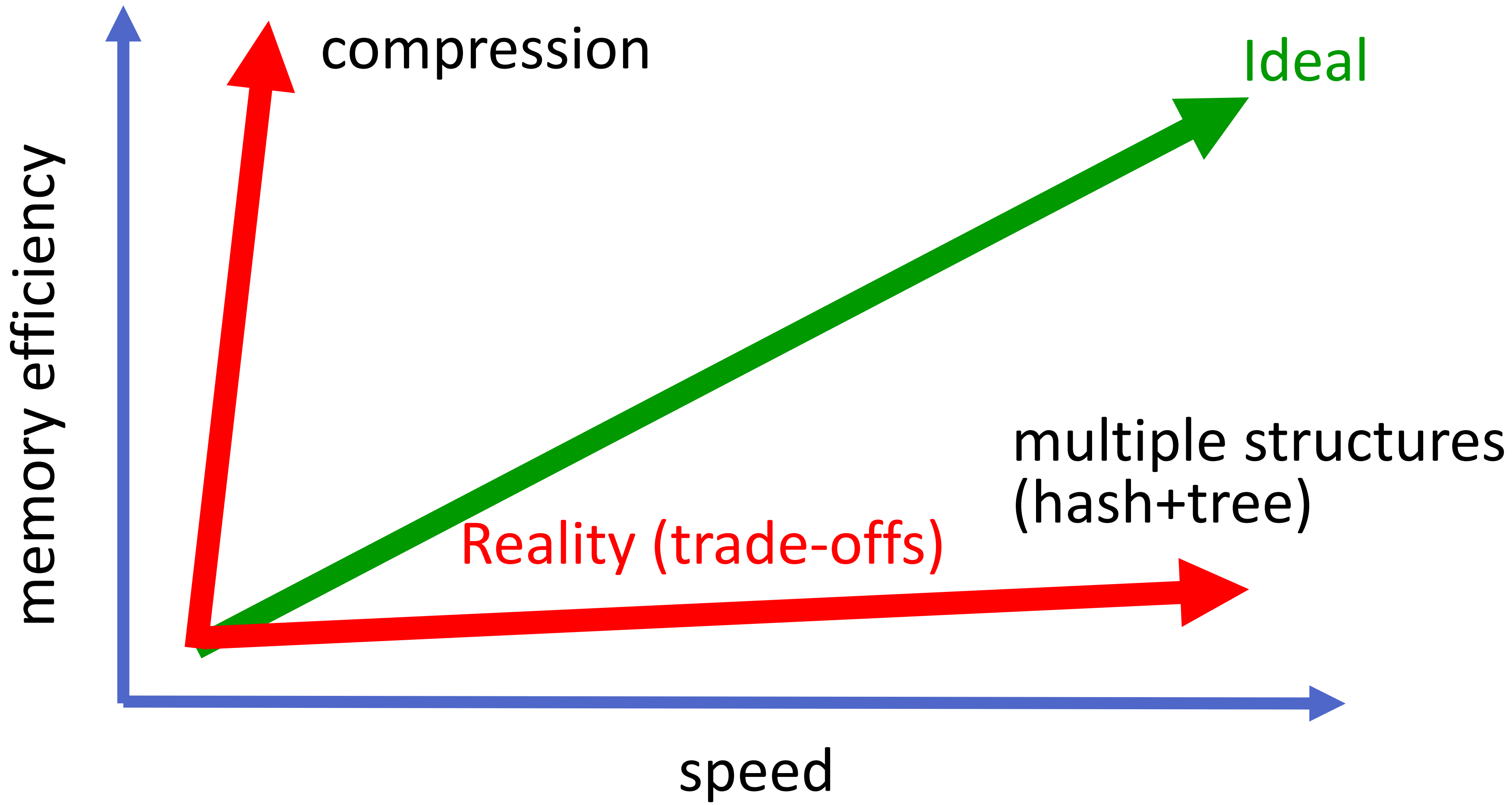
[Zhang et al., SIGMOD'16]

	Tuples	Primary Indexes	Secondary Indexes
TPC-C	42.5%	33.5%	24.0%
Articles	64.8%	22.6%	12.6%
Voter	45.1%	54.9%	0%

Table 1: Percentage of the memory usage for tuples, primary indexes, and secondary indexes in H-Store using the default indexes (DB size $\approx$ 10 GB).

memory efficiency
(smaller=better)

Index design involves trade-offs



Outline

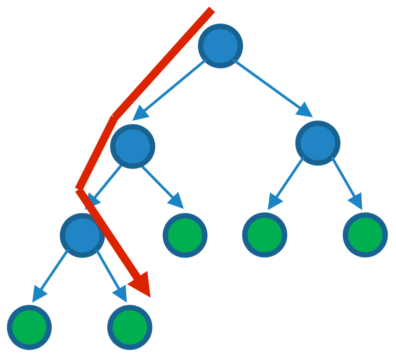
Running example: in-memory DB ordered index

- Definition
- Challenges
- Opportunity: MLP
- **Cuckoo Trie**: MLP-first index design
- What’s next

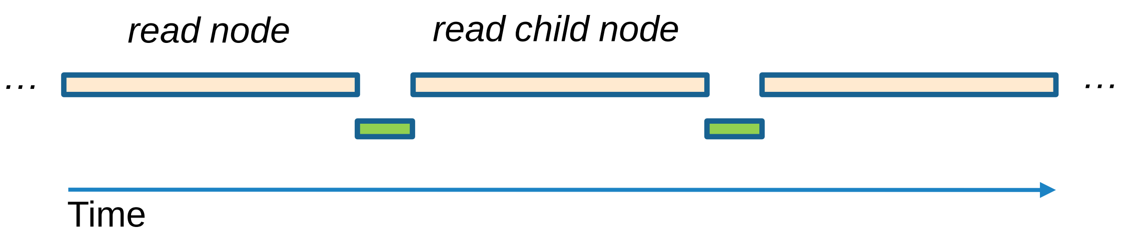
Indexes bottlenecked on memory

Ordered indexes are hierarchical: B-trees, tries, …

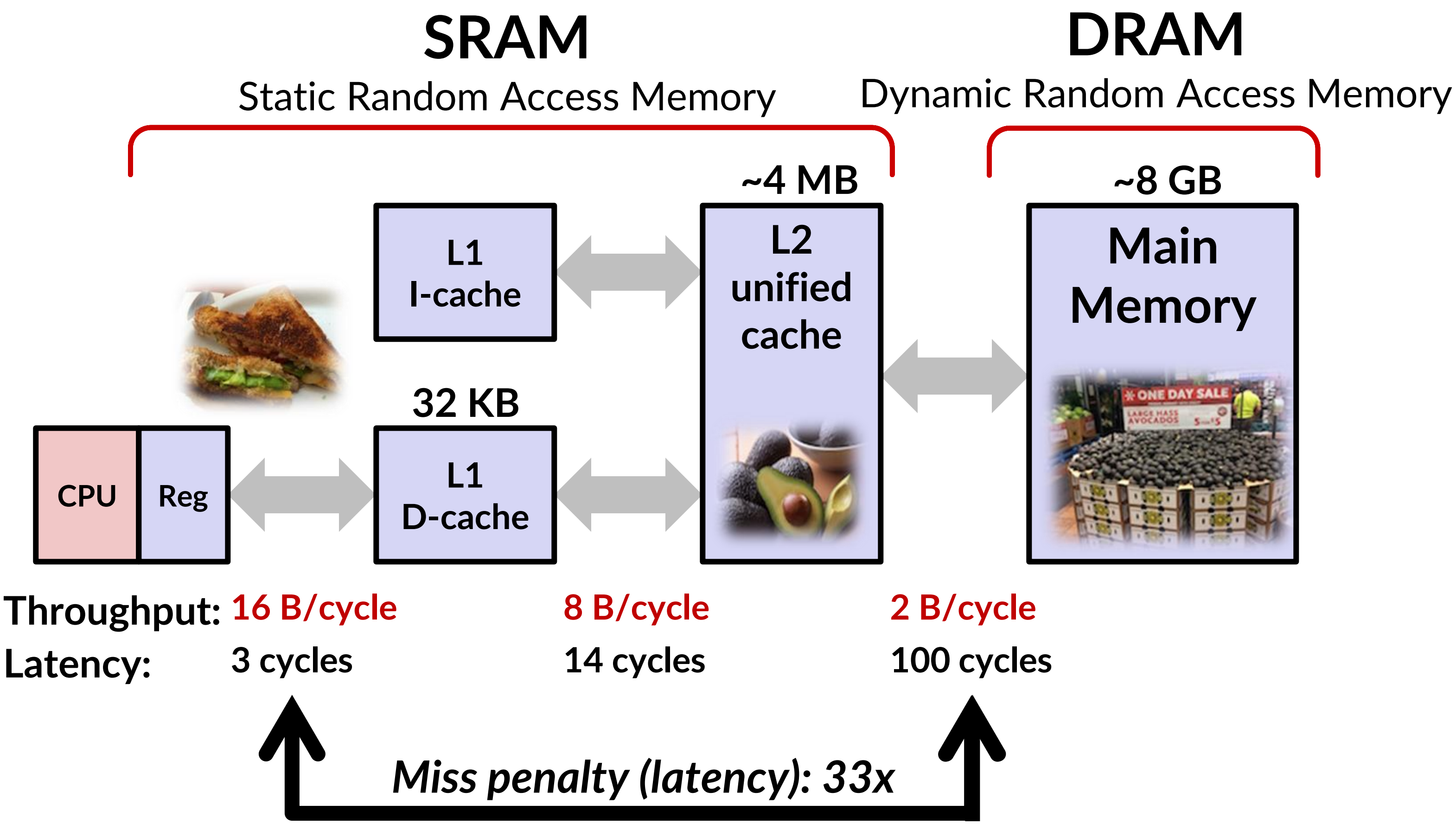
Pointer-chasing



Memory access is slow



Memory access is slow



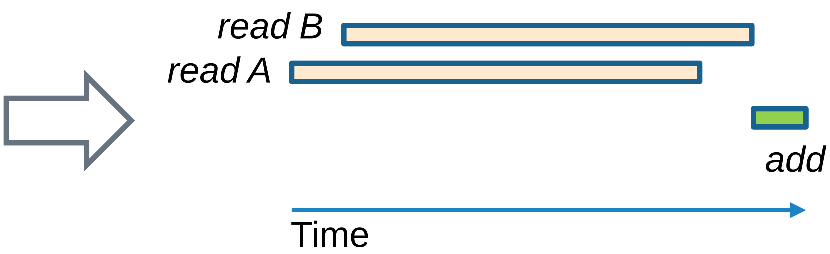
Memory-level parallelism (MLP)

Out-of-order instruction execution to the rescue

⇒ Can **overlap** execution of memory reads

```
// x = *mem1 + *mem2;

READ A from Mem1
READ B from Mem2
ADD x, A, B
```



… as long as they're **independent**

MLP = average number of parallel memory accesses

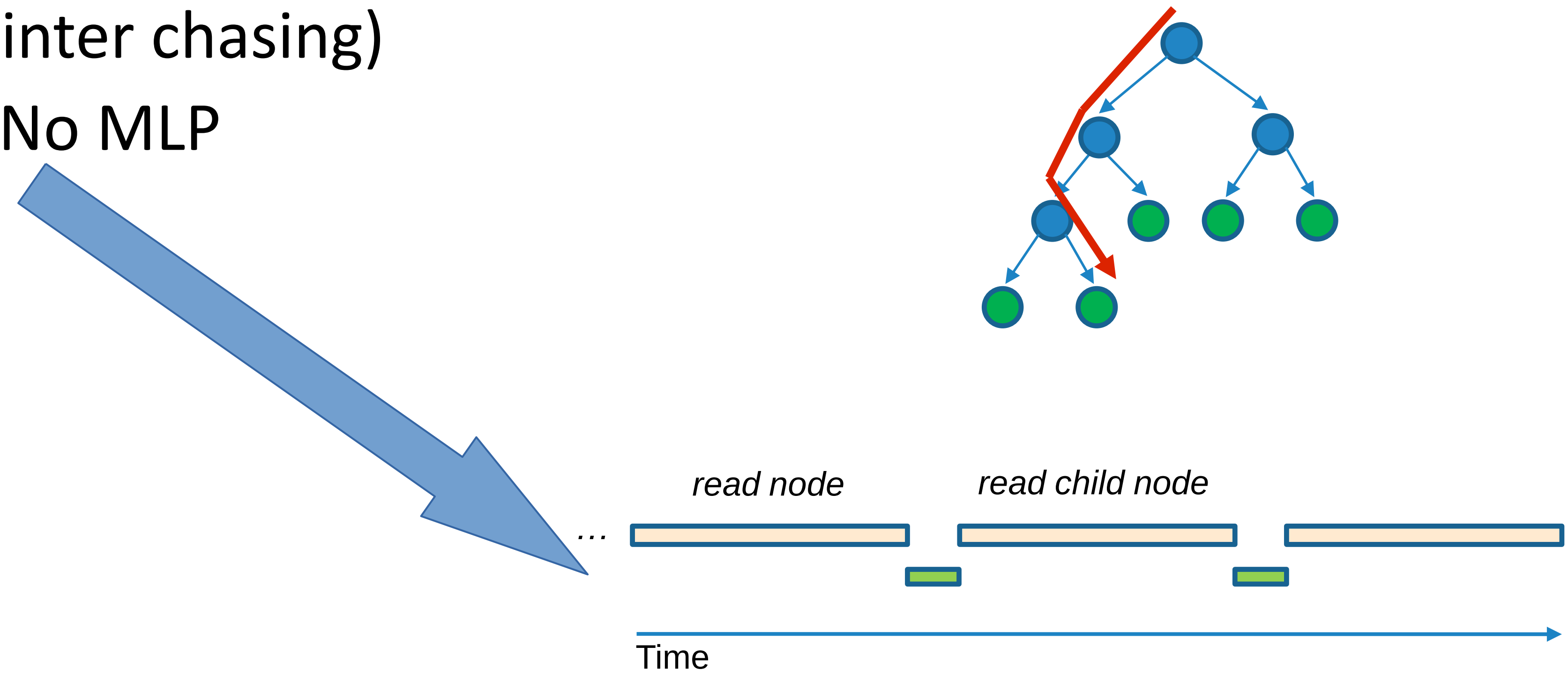
MLP in trees (or lack thereof)

Trees have dependent reads:

Child address is stored in parent

(Pointer chasing)

$\Rightarrow$ No MLP



Outline

Running example: in-memory DB ordered index

- Definition
- Challenges
- Opportunity: MLP
- **Cuckoo Trie**: MLP-first index design
- What's next

Cuckoo Trie (CT)

Index designed with MLP as a **primary consideration**

- Adar Zeitak's MSc thesis

Memory efficient

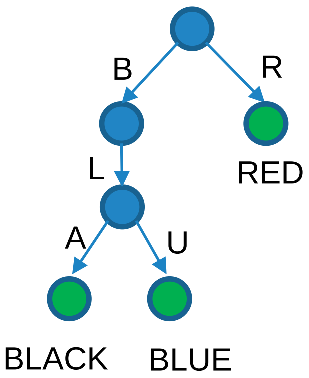
Thread-safe and scalable

Tries

Keys = sequence of *symbols*

Nodes = Prefixes of keys, Leaves = keys

Each node is an array of *d* children, corresponding to all next possible symbols

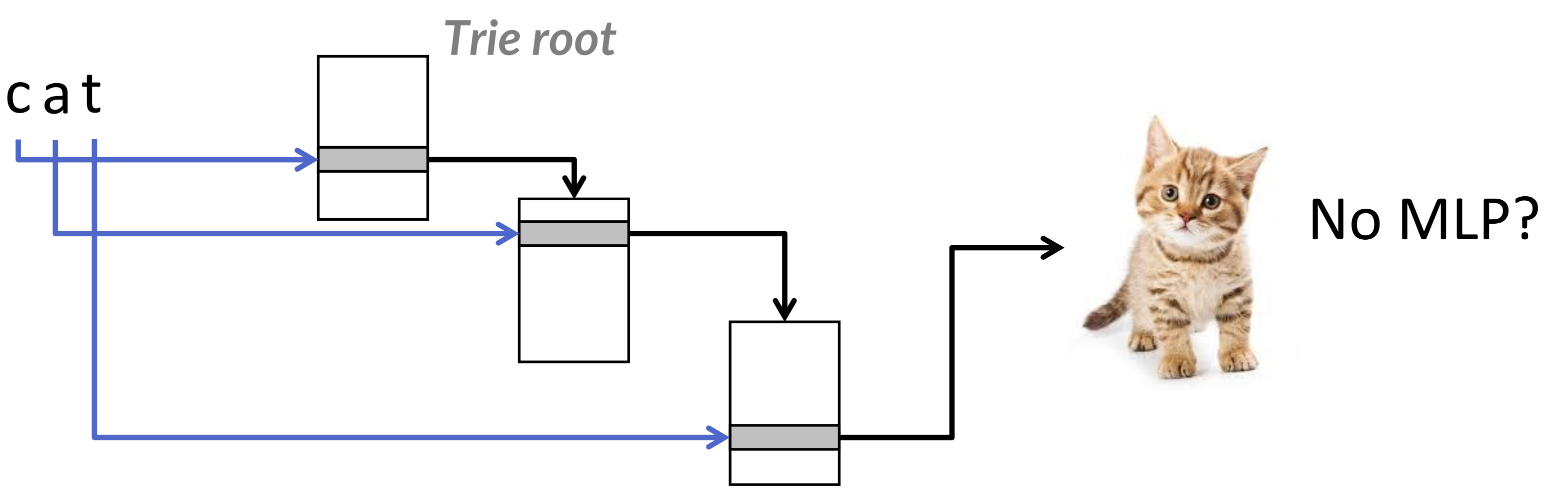


Tries

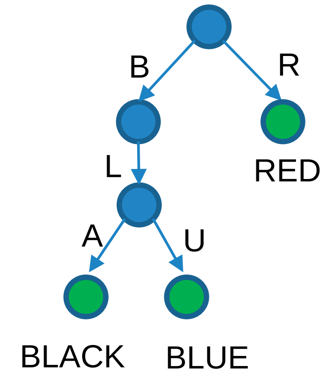
Keys = sequence of *symbols*

Root corresponds to the empty string

Each node is an array of *d* children, corresponding to all next possible symbols



Cuckoo Trie: main idea



Start with a trie

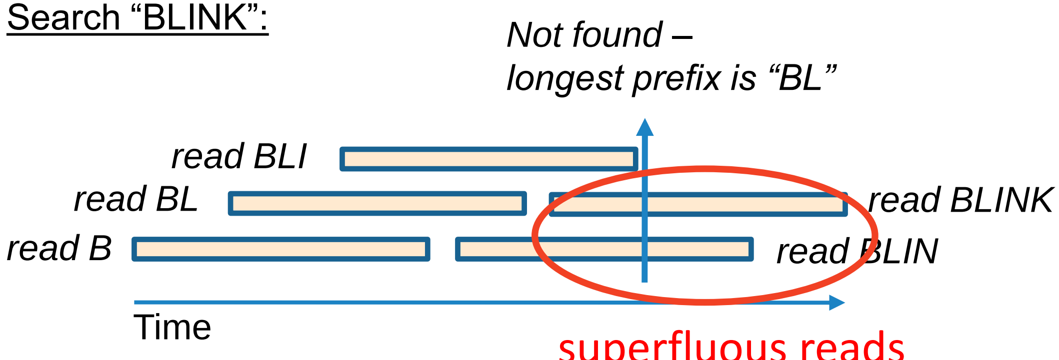
Store nodes in **bucketized** cuckoo hash table (BCHT), keyed by their name (prefix)

Key	Value
(empty)	
R	
BL	
B	
BLA	
BLU	

⇒ No pointers!

⇒ Can read nodes **in parallel**

Search “BLINK”:



Cuckoo Trie search code

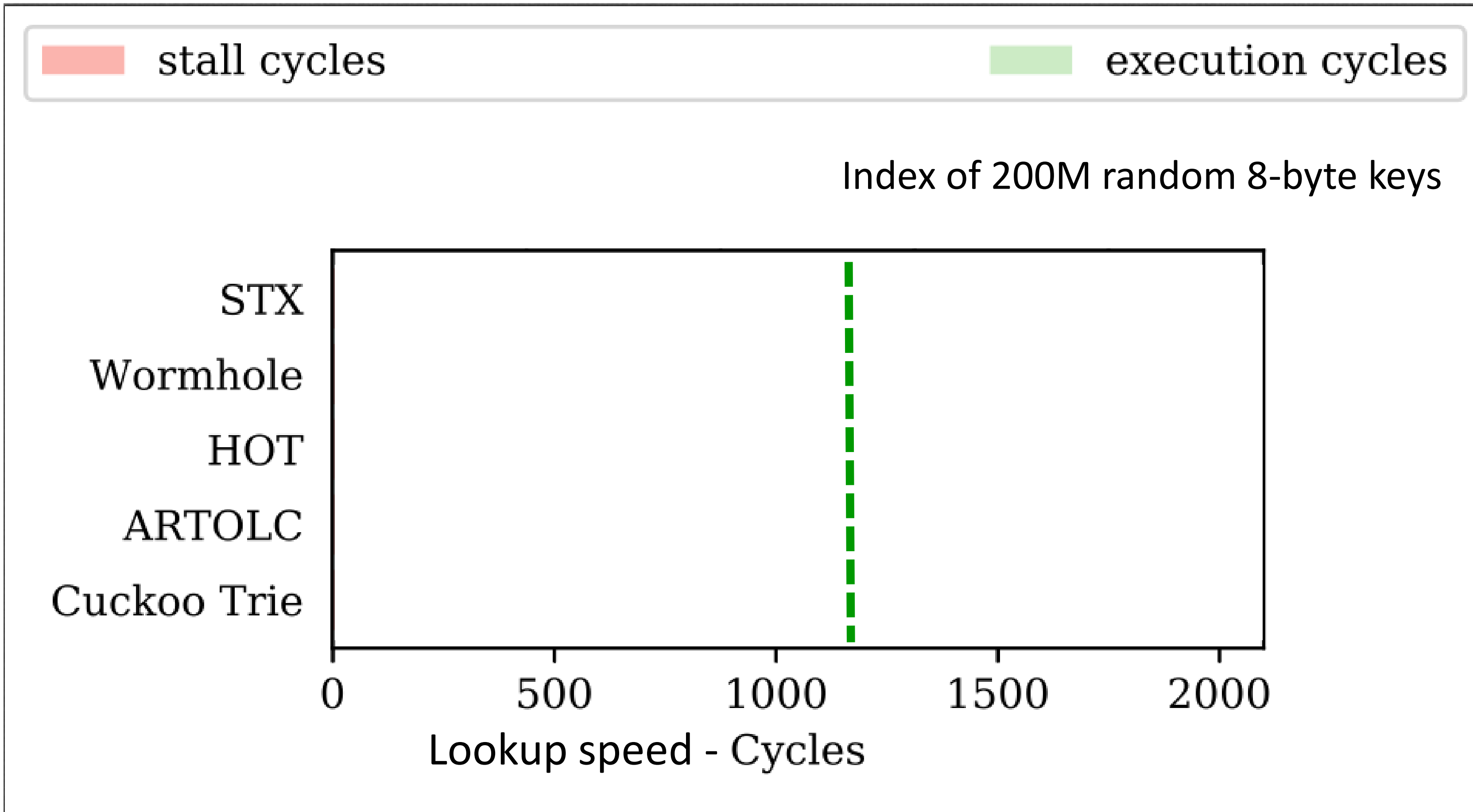
Algorithm 1 Cuckoo Trie search path traversal

Parameters:
H - the hash function
D - the prefetch depth

```
function SEARCH(k)
    restart:
    for i ← 1 to D do
        prefetch BUCKETS(H(k[: i]))          ▷ two BCHT buckets of k[: i]
    node ← Root
    depth ← 0
    for i ← 0 to #k − 1 do
        prefetch BUCKETS(H(k[: D + i + 1]))
        node′, depth′ ← FINDCHILD(node, depth, k, i)
        if node′ = null then
            return node                       ▷ reached a leaf
        node, depth ← node′, depth′
    return node
```

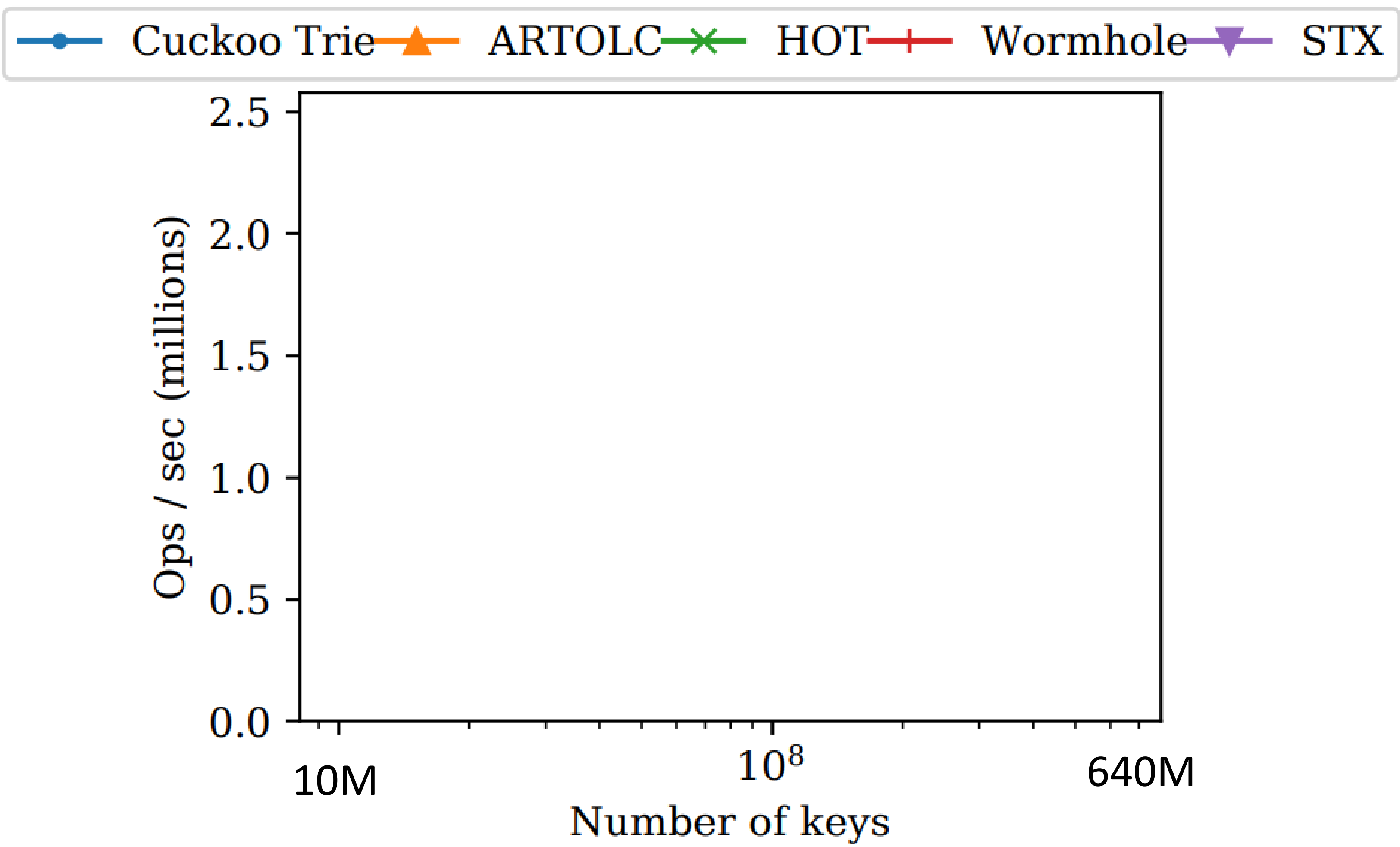
Cuckoo Trie MLP impact

CT lookup faster than other indexes stall on DRAM



MLP ⇒ Data set size scalability

CT scales with data set size



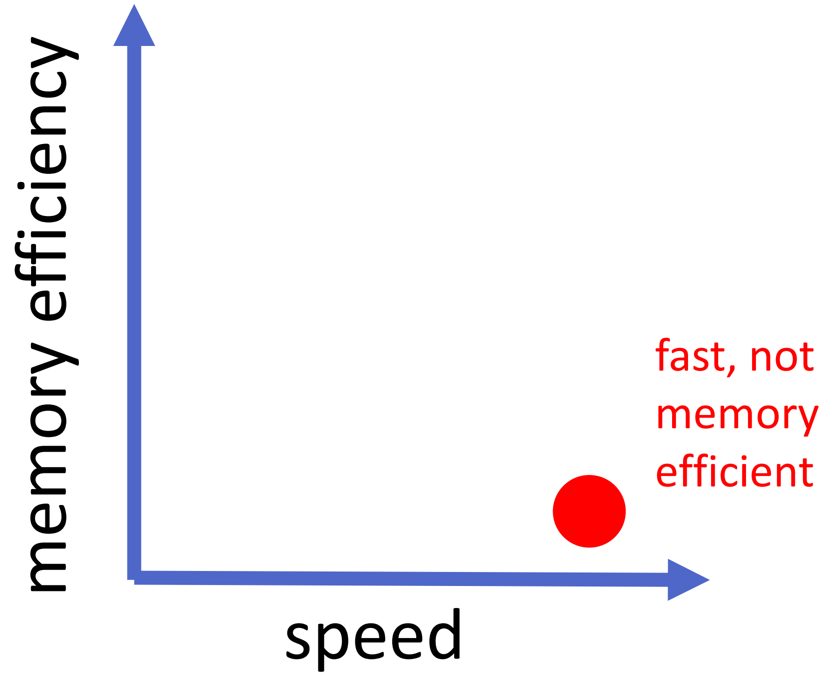
Challenge: memory efficiency

Problem: hash table *keys*

- Each hash entry stores a key (prefix)

⇒ Large, variable-sized entries?

Key	Value
(empty)	
R	
BL	
B	
BLA	
BLU	



Idea: use key redundancy

Problem: hash table *keys*

- Each hash entry stores a key (prefix)

⇒ Large, variable-sized entries?

Solution: use key redundancy (prefix!)

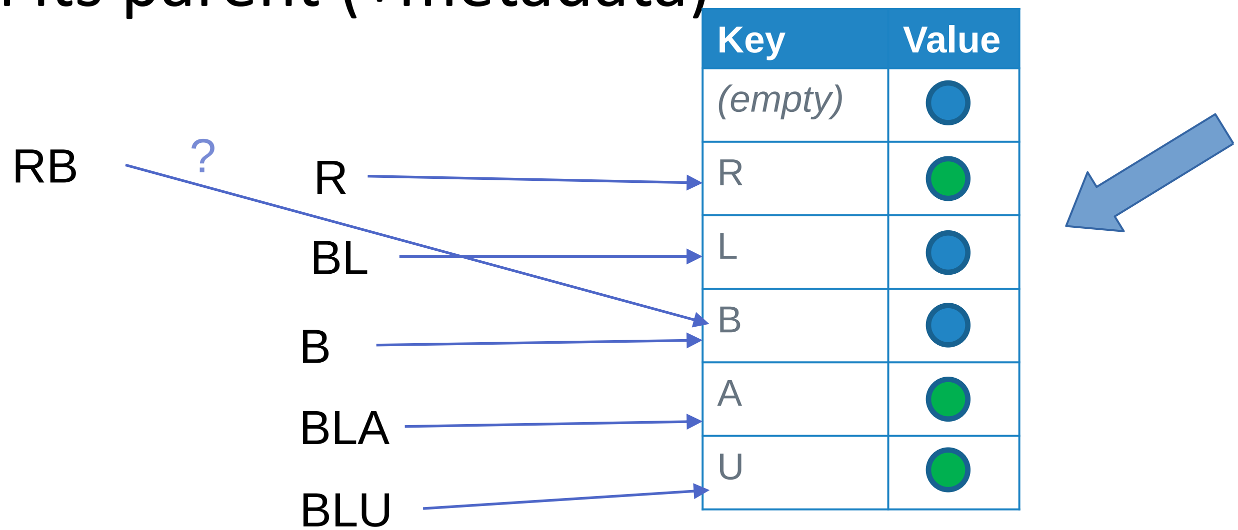
⇒ Constant-size hash table entries

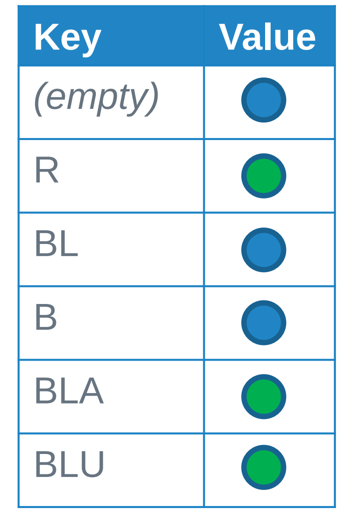
Making this work is non-trivial

⇒ <u>Exploiting MLP doesn't come "for free"</u>

Using key redundancy

Each node stores the **transfer symbol** from its parent (+metadata)





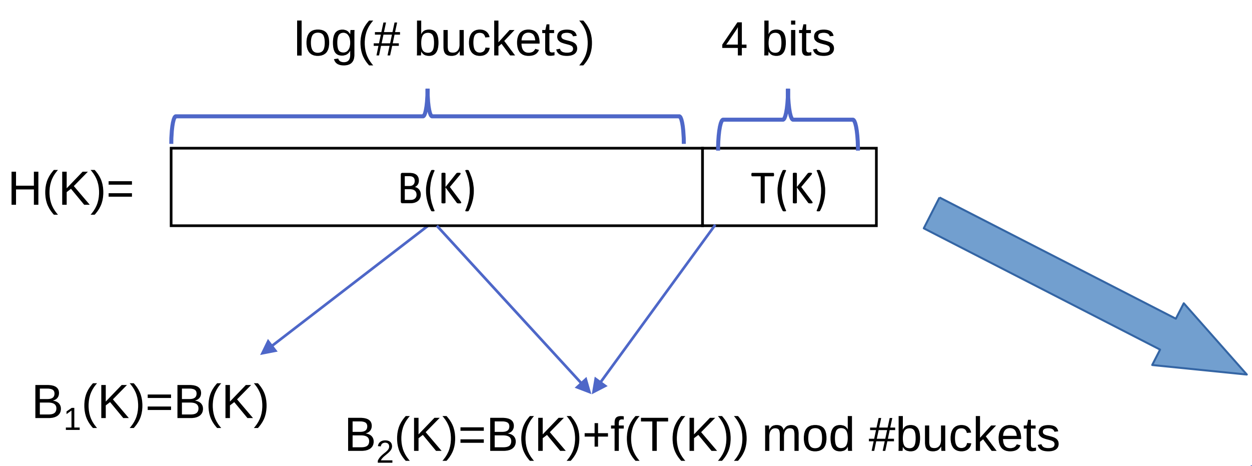
Problems (when does cuckoo hash need the key?):

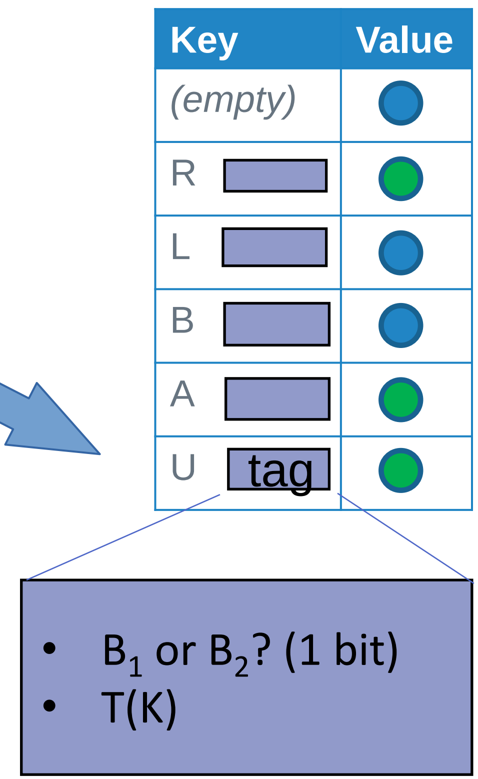
1) Relocations
2) Resolving hash collision (identifying a match)

Solving relocations

Idea:

- Derive key buckets from one hash function (not two)
- Store information required to compute alternate bucket in entry

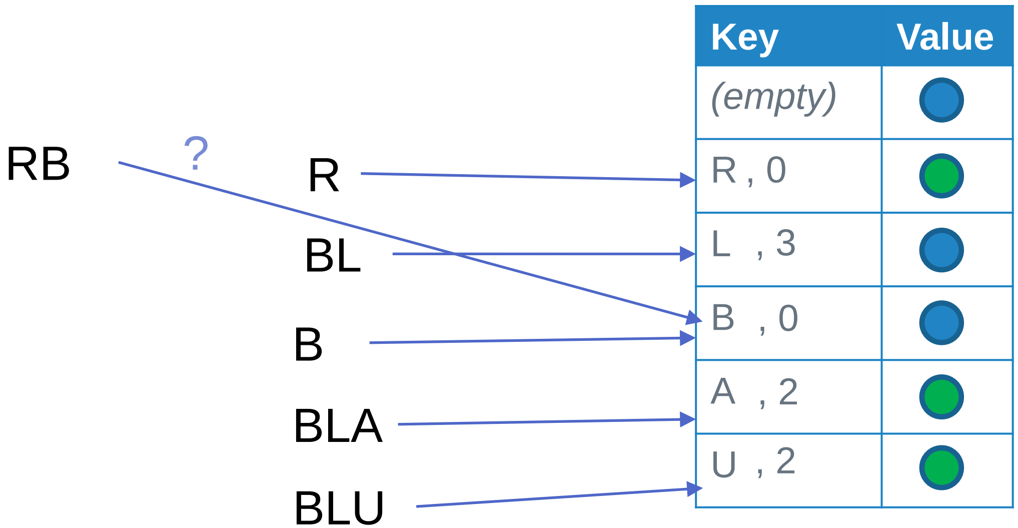




$\Rightarrow$ Can compute entry's alternate bucket from entry

Solving entry verification

Idea: Conceptually, node identifies its parent



☹ Storing explicit parent pointers isn't memory efficient

Solving entry verification

We're searching for node named *X*.

Assume we found its parent, *P*, named *X[:#X-1]*.

To verify key *K* in an entry is *X*, we need to:

1) Verify that *K[-1] = X[-1]*
2) The parent of *K* is the node *P* we already found

Solving entry verification

We're searching for node named *X*.
Assume we found its parent, *P*, named *X[:#X-1]*.

To verify key *K* in an entry is *X*, we need to:

1) **Verify that *K[-1]* = *X[-1]* (transfer symbol in entry)**
2) The parent of *K* is the node *P* we already found

Solving entry verification

We're searching for node named *X*.

Assume we found its parent, *P*, named *X[:#X-1]*.

To verify key *K* in an entry is *X*, we need to:

1) Verify that *K[-1]* = *X[-1]*
2) **The parent of *K* is the node *P* we already found**

Use a special *peelable* hash function: given $H(k \circ c)$ and symbol *c*, can compute $H(k)$

Solving entry verification

To verify key *K* in an entry is *X*, we need to:

1) Verify that $K[-1] = X[-1]$
2) **The parent of K is the node P we already found**

Use a special *peelable* hash function: given $H(k \circ c)$ and symbol *c*, can compute $H(k)$

Each entry stores:

- **Color**, a unique integer chosen on insertion to be unique among entries with the same hash (bucket pair)
- The color of its parent

Need to verify that *K*.parentColor=*P*.color

Solving entry verification

function BUCKETS(h)
    $b_1 \leftarrow h/T$
    $b_2 \leftarrow (b_1 + M[h \ mod \ T]) \ mod \ S$
    return b_1, b_2

function ENTRIESFOR(h)
    $b_1, b_2 \leftarrow$ BUCKETS(h)
    $tag \leftarrow h \ mod \ T$
    $S_1 \leftarrow \{n \in B[b_1] \mid n.tag = tag \wedge n.is\_primary\}$
    $S_2 \leftarrow \{n \in B[b_2] \mid n.tag = tag \wedge \neg n.is\_primary\}$
    return $S_1 \cup S_2$

H(X) X[-1]

function SEARCHBYPARENT($hash, last\_sym, color$)
    for c **in** ENTRIESFOR($hash$) **do**
        if $c.last\_symbol = last\_sym$ **then**
            if $c.parent\_color = color$ **then**
                return c
    return null

parent's color

Correctness of entry verification

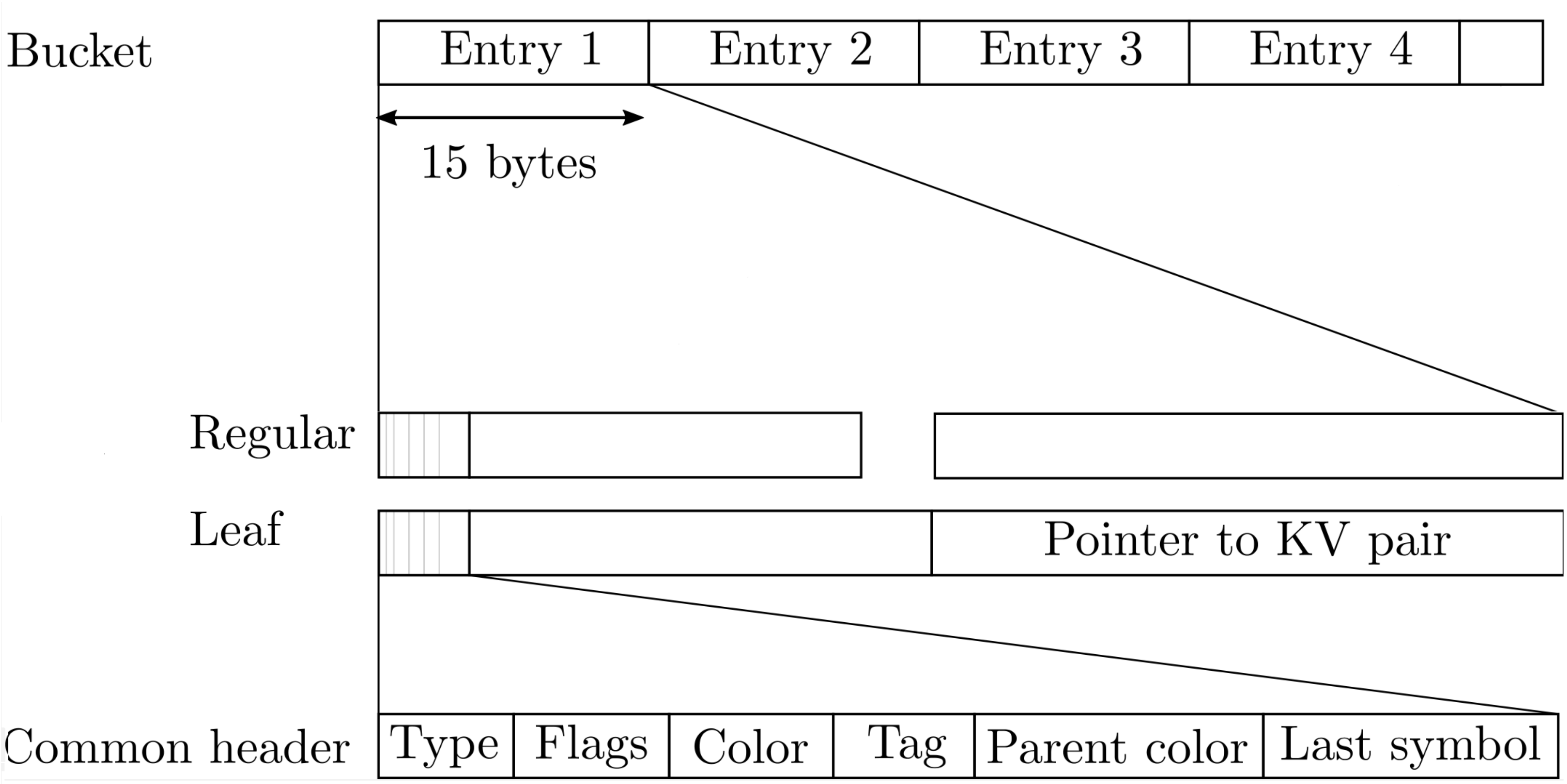
Verified that K.parentColor=X.color

$H(K) = H(X)$ (by checking tag)

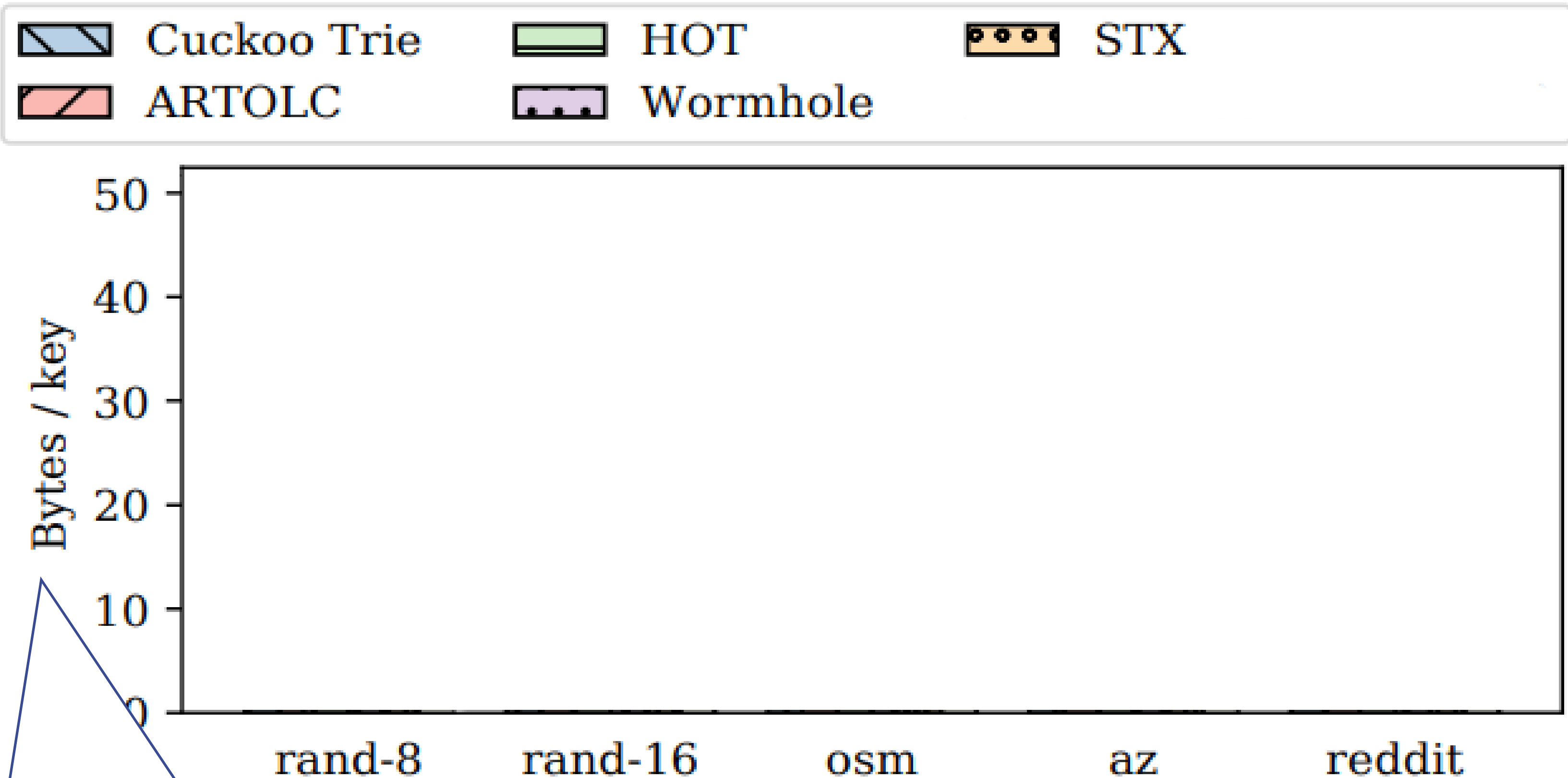
$\Rightarrow H(K[:-1]) = H(X[:-1])$ (by peelability, since transfer symbol matches)

$\Rightarrow K = X$ (by color match, since color is unique)

Entry format: putting it together

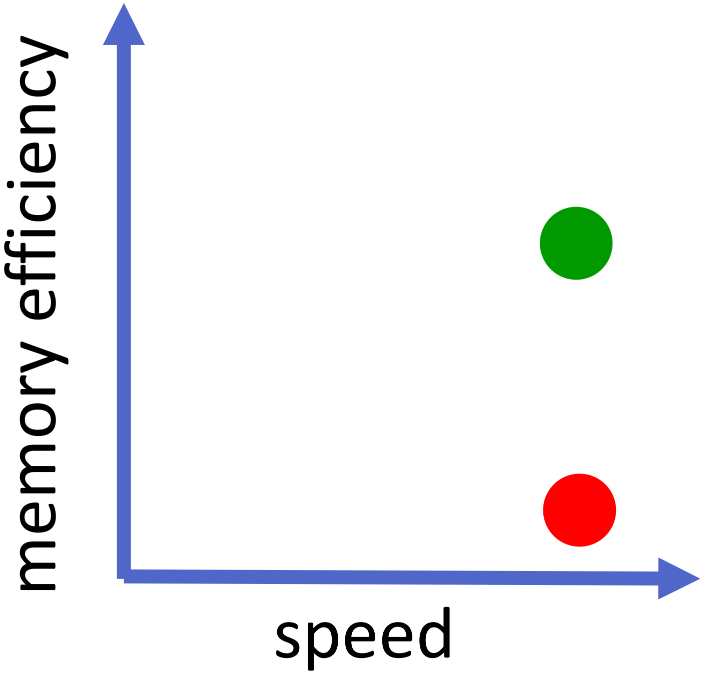


Cuckoo Trie memory efficiency



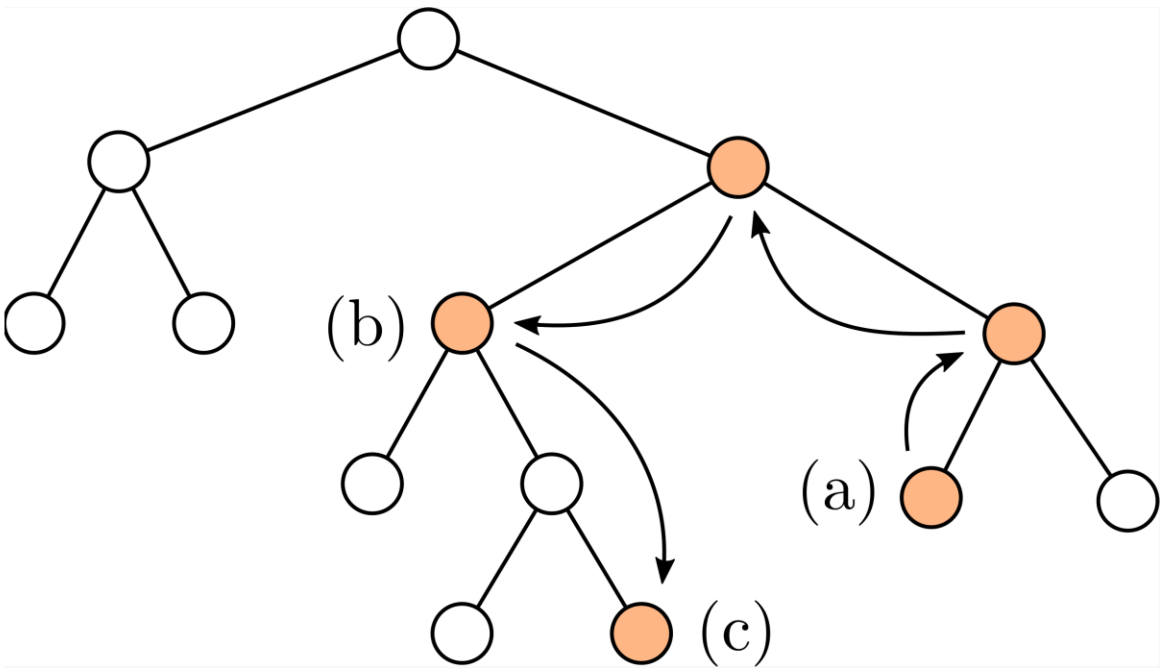
Includes hash load factor, multiple nodes per key, etc.

Cuckoo trie memory efficiency

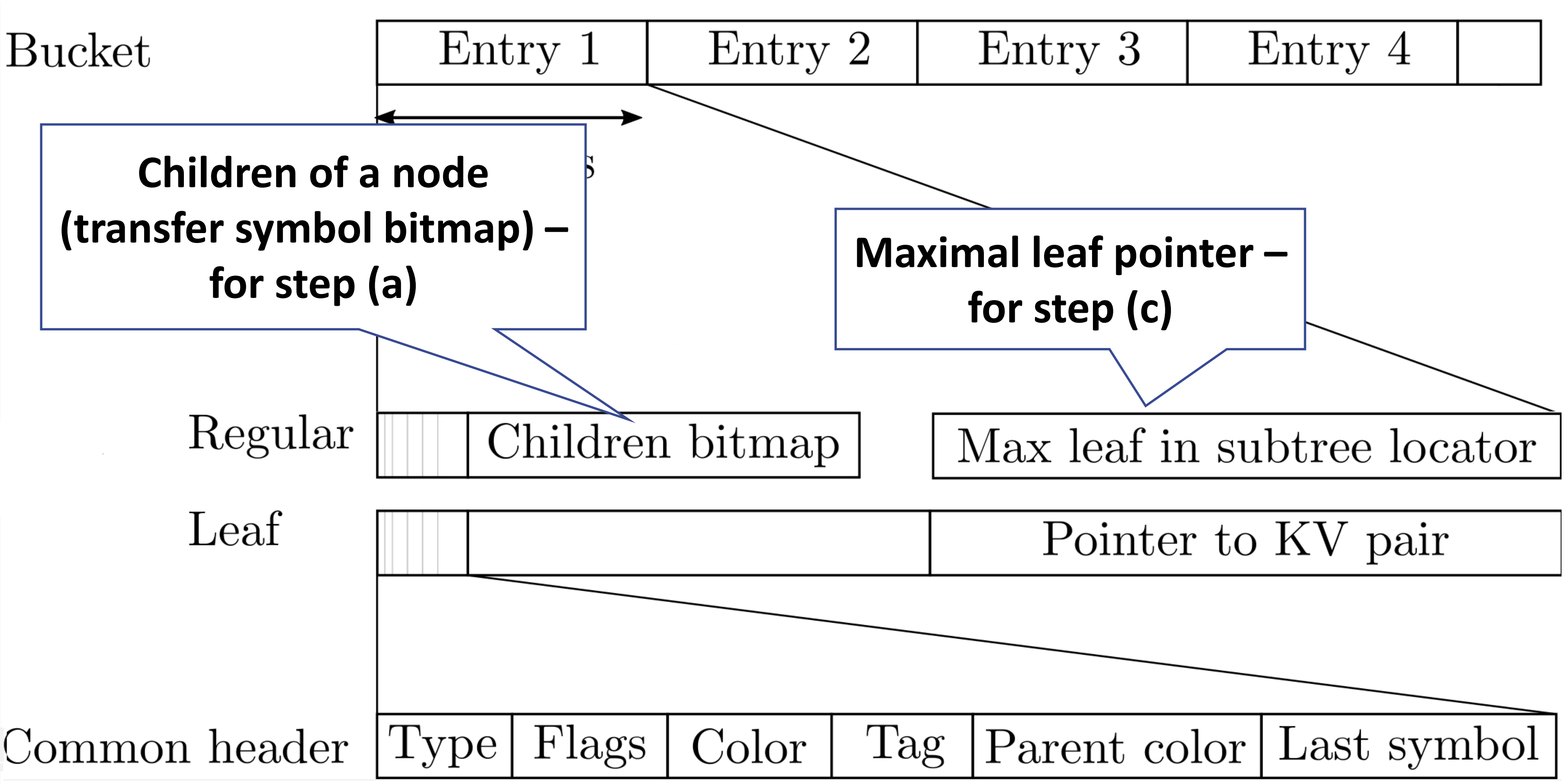


Range scans: Predecessor search

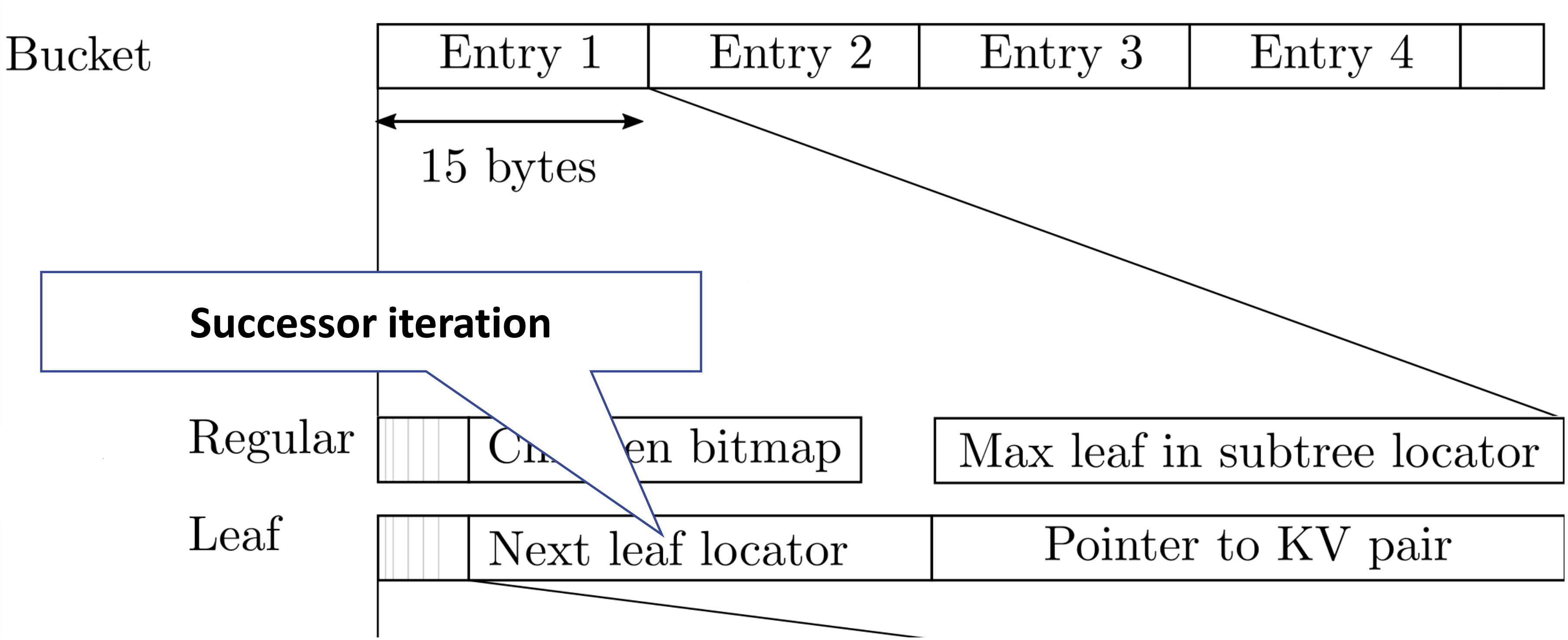
(a) Climb up until a node with a smaller child is found
(b) Go to this child
(c) Go to maximal child in its subtree



Range scans: Predecessor search



Range scans: Successor search



Successor iteration is sequential, but uses MLP by prefetching the next node at each step

$\Rightarrow$ Overlap client work with DRAM access

Cuckoo Trie: Multi-threading

Multi-core scalability

- Optimistic concurrency control protocol
- Traversal concurrent with updates

Good trade-off: No other index is both faster and smaller

Outline

Running example: in-memory DB ordered index

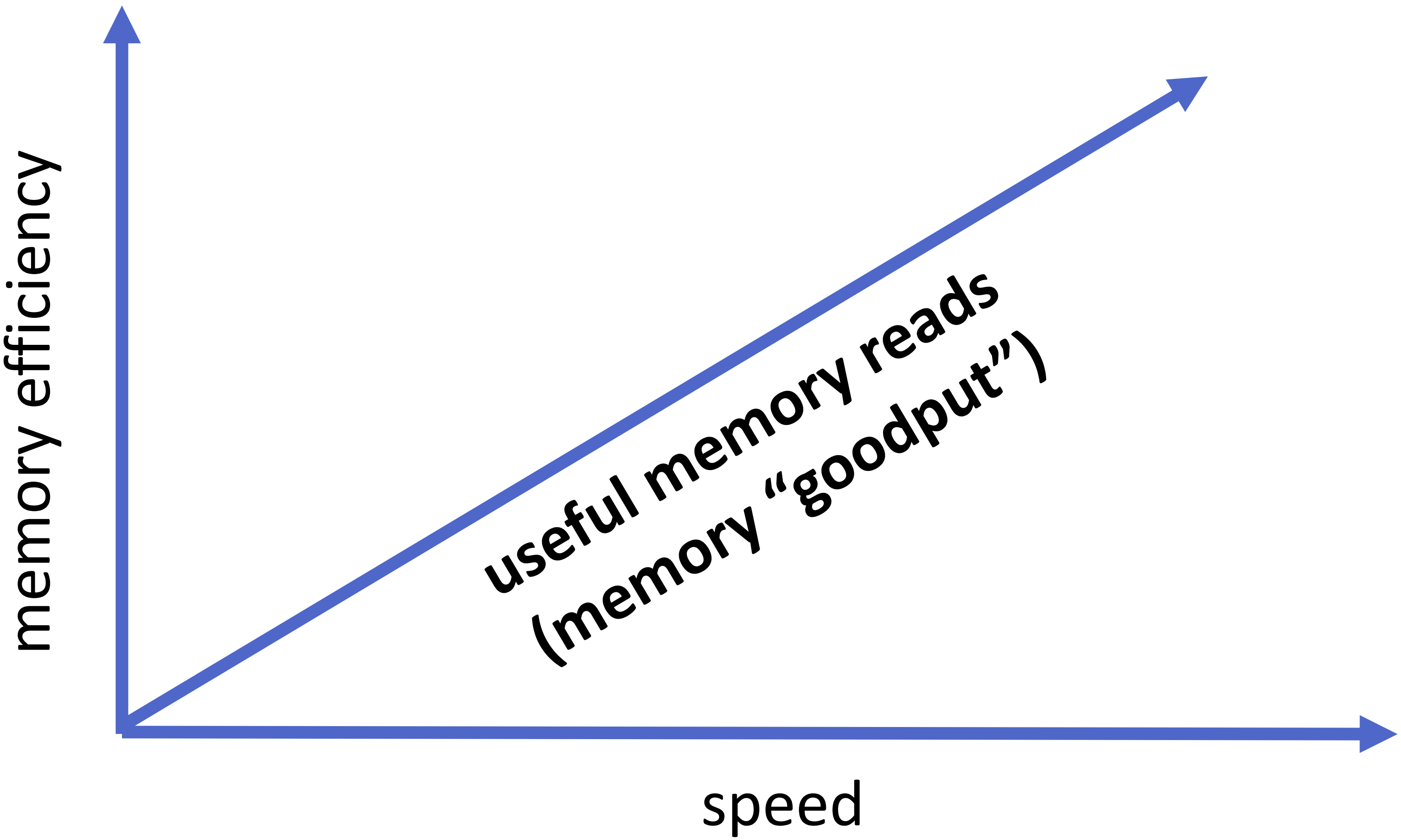
- Definition
- Challenges
- Opportunity: MLP
- **Cuckoo Trie**: MLP-first index design
- $\boxed{\text{What's next}}$

What's next?

- MLP is not a silver bullet
  - Opens a new world of trade-offs

- Go beyond…
  - Indexing
  - DRAM

MLP extends the trade-off space

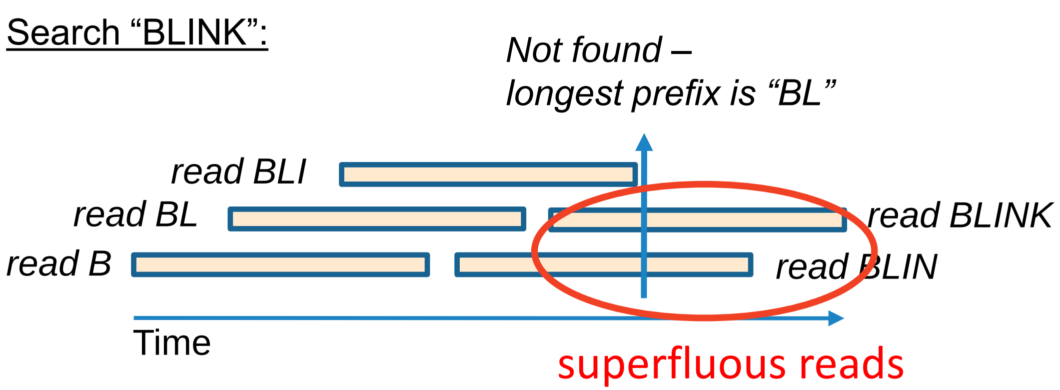


Cuckoo Trie limitations

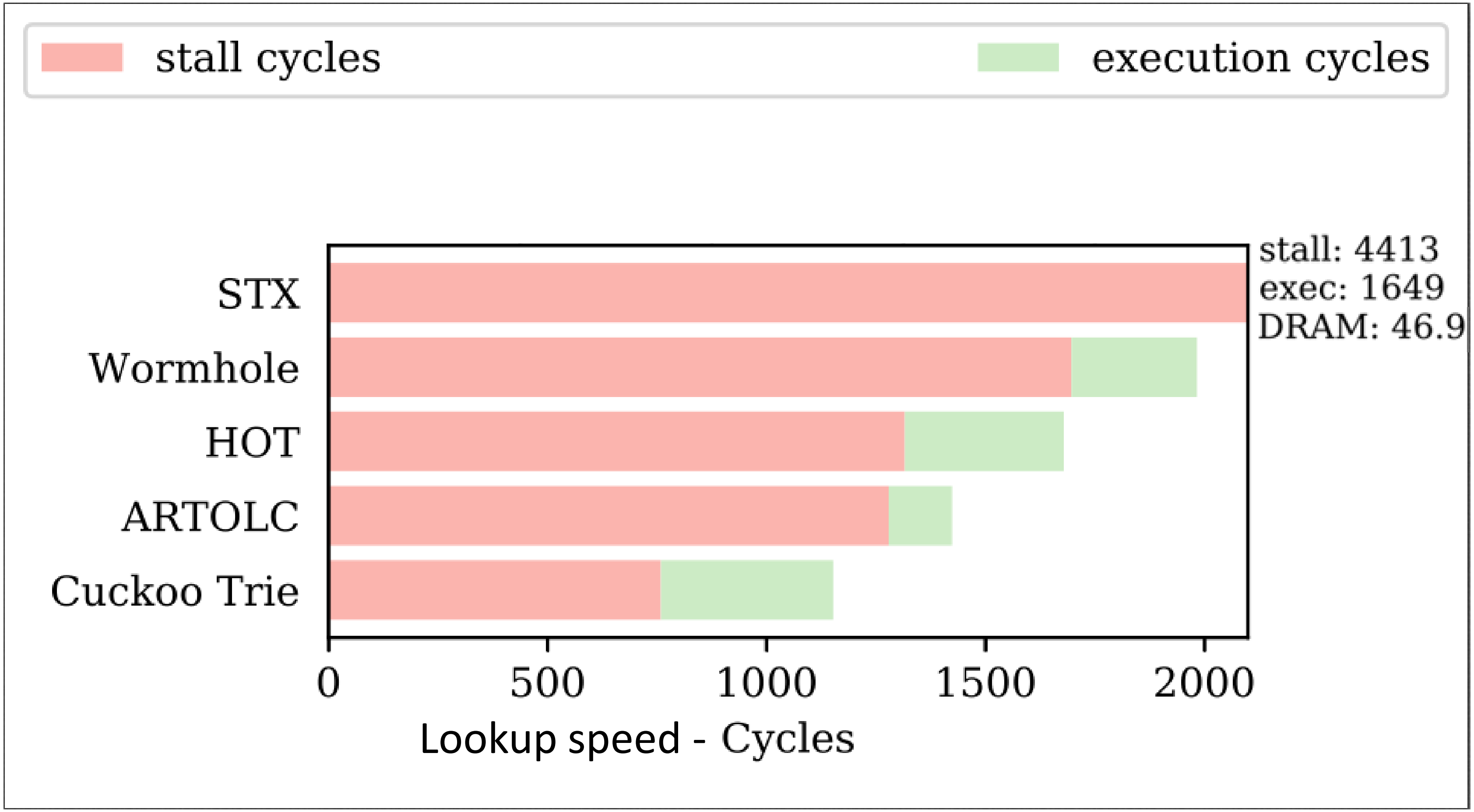
No dependent reads

So all reads performed up front

$\Rightarrow$ **Superfluous reads**



Cuckoo Trie memory bandwidth



Index of 200M random 8-byte keys

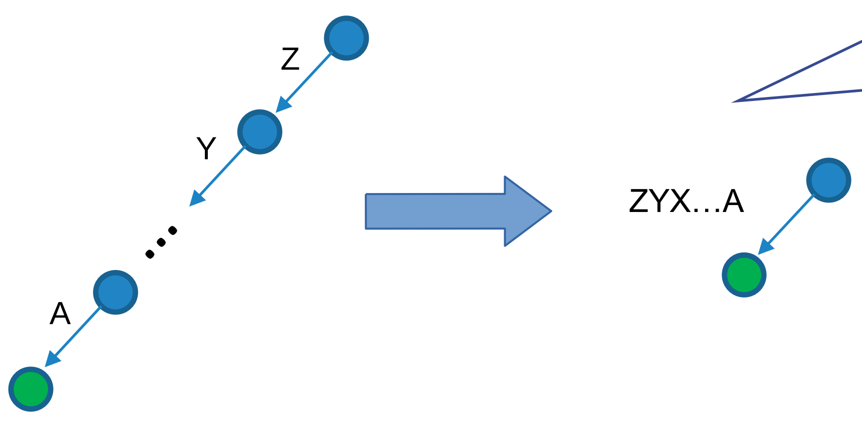
Cuckoo Trie limitations

No dependent reads

So all reads performed up front

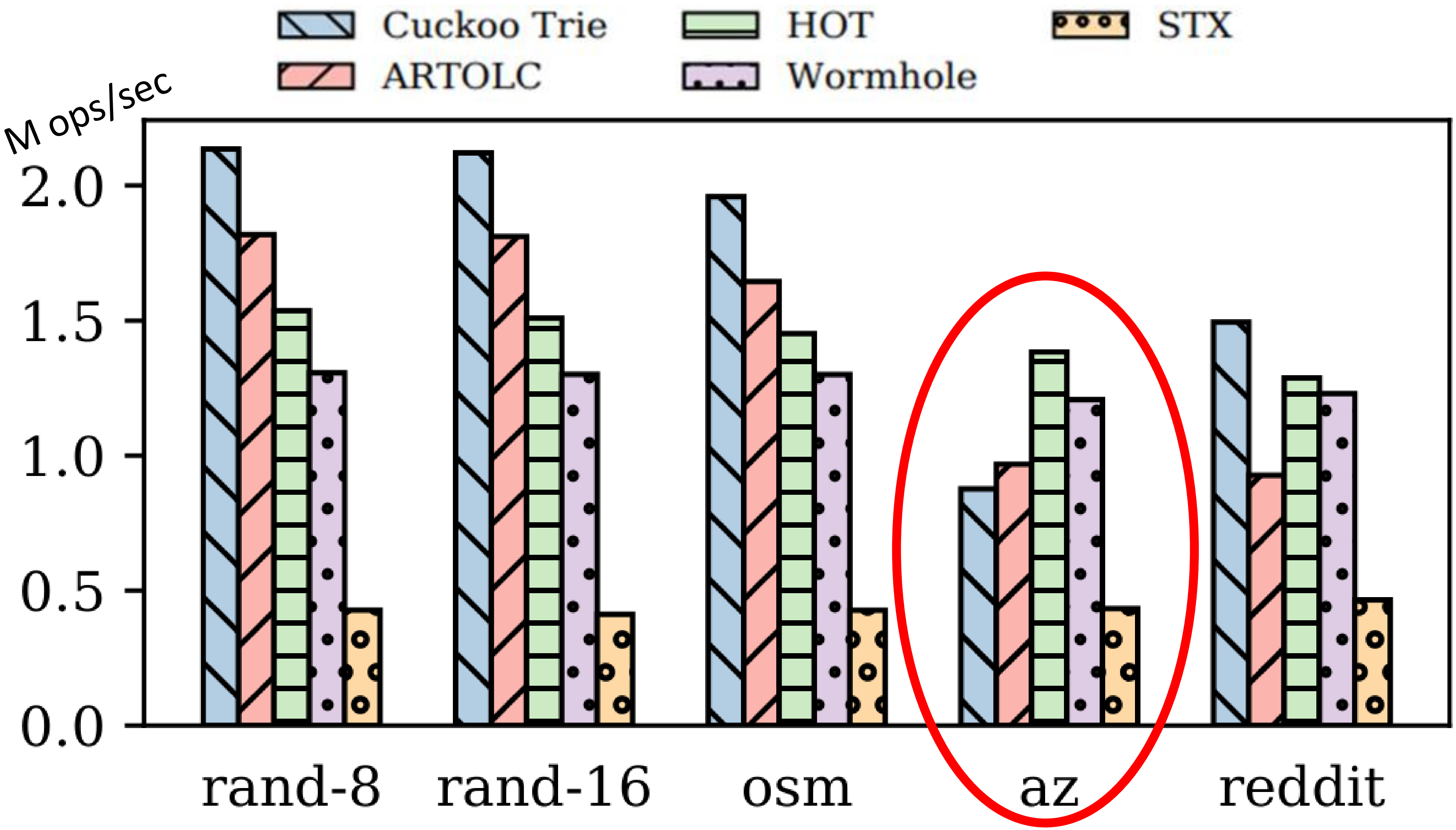
$\Rightarrow$ **Certain optimizations less effective**

- Path compression



Cuckoo Trie has path compression but it only saves space, not time

Lookups for different key lengths



Conclusion

System design should **explicitly target MLP**

Exploiting MLP may be **not trivial**

Next steps:

- Go beyond indexing
- Remote memory (RDMA, CXL) / block storage (SSD)
  - Higher latency and (effectively) lower bandwidth
  - All can sustain (forms) of MLP