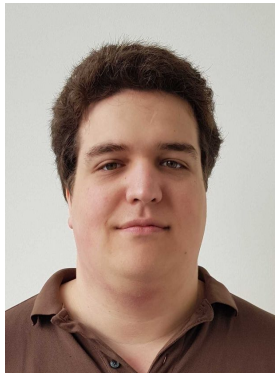
UNIVERSITÄT DUISBURG ESSEN

Universitätsklinikum Essen

technische universität dortmund

fakultät für informatik

Fundamentals of alignment-free sequence analysis: *k*-mer hashing



Part II: Hashing strategies, hardware considerations, summary

Jens Zentgraf & Sven Rahmann

ACM-BCB 2020

Hashing

- Set ("universe") U of possible keys
- Set of keys $K \subseteq U$ to be stored, $|K| = N$
- Hash table (array) with P slots
- Hash function h
  - $h: U \rightarrow \{0, \ldots, P - 1\}$

Hashing

- Universe U: all possible first names (of finite length)
- Set of keys $K = \{Anna, Franz, Bea, Fritz\}$
- Hash table with 6 slots
- Some function *h* mapping *U* to {0, 1, 2, 3, 4, 5}.

Hashing

h("Anna") = 3

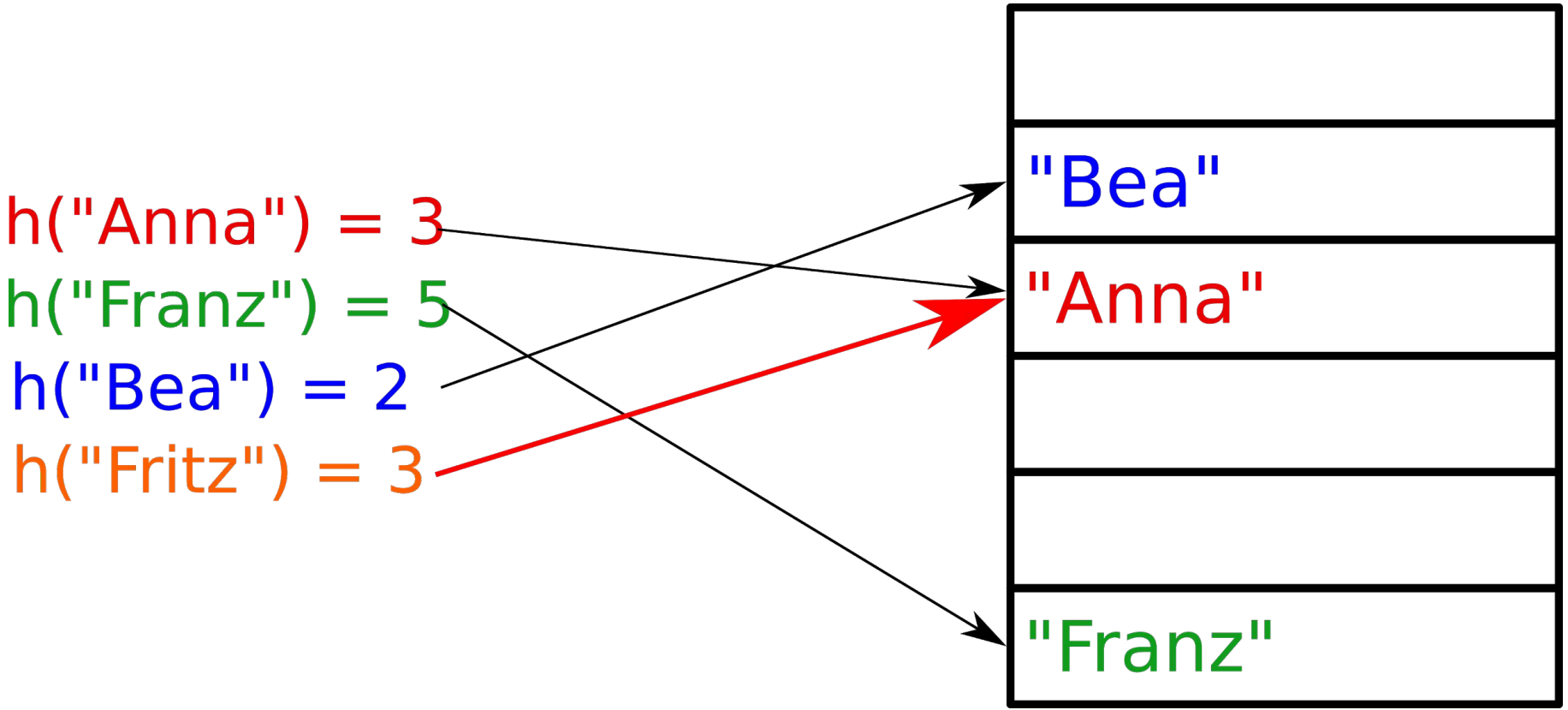
h("Franz") = 5

h("Bea") = 2

Hashing

h("Anna") = 3
h("Franz") = 5
h("Bea") = 2

Hashing



Collisions and strategies of collision resolution

Definition: collision

- Two elements are hashed to the same location
- $h(a) = h(b)$ for some $a \neq b$

Strategies of collision resolution:

- Chaining (also: separate chaining)
- Open addressing (also: closed hashing)

Hopscotch hashing

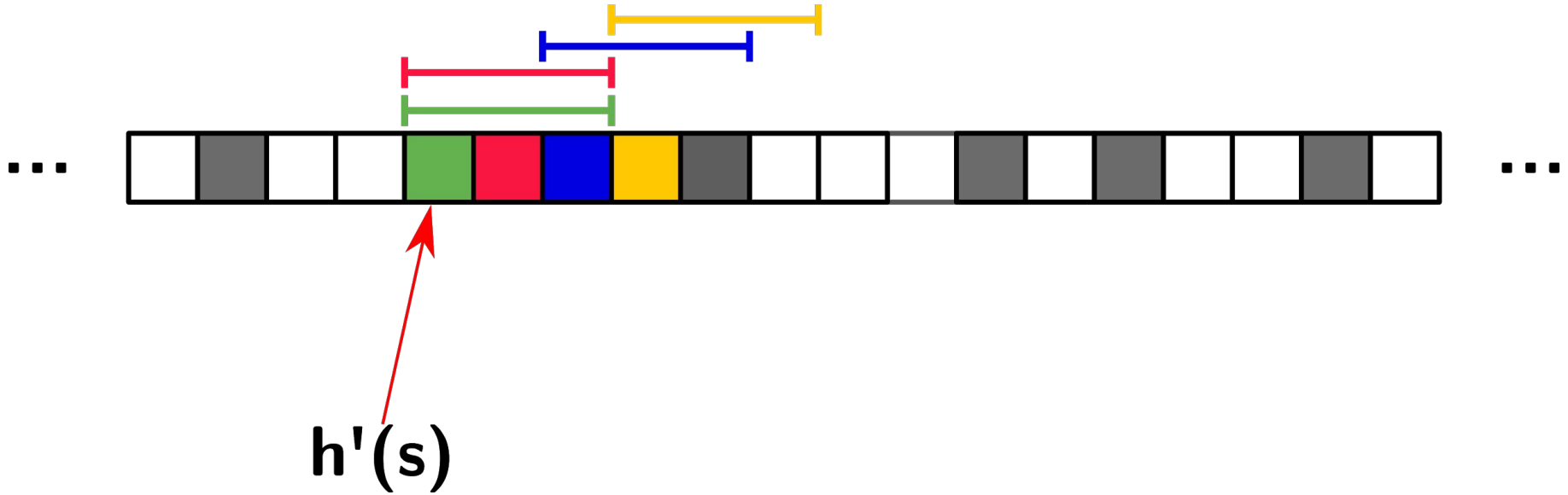
- One hash function $h(x)$
- Maximum range r to insert element
- Element can be inserted between $h(x)$ and $h(x)+r-1$
- Bit array of length r for each position: Which elements in [h(x), h(x)+r-1] belong to h(x) ?
- Insert: O(N)
- Lookup: O(r)

- Compute hash position $h(x)$
- If $h(x)$ is occupied:
  - Find next empty slot $y \geq h(x)$ (can be far away)
  - Find an element that can be moved to the empty slot:
    - Start at position $i = y - r - 1$
    - Check if element can be moved to y
    - If not, repeat for increasing i up to $y-1$
    - Fail if no element found.
  - Repeat with $y = i$ (new empty slot),till x can be inserted in $[h(x), h(x)+r-1]$.

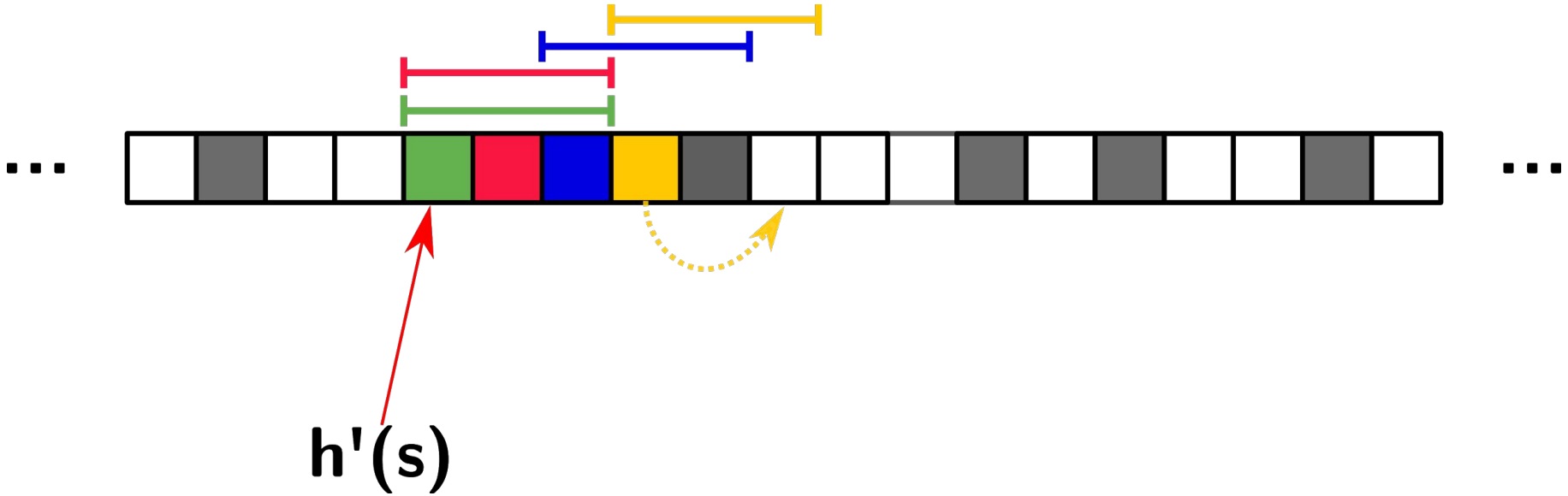
Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In Distributed Computing, pages 350–364, Berlin, Heidelberg, 2008. Springer.

Hopscotch hashing

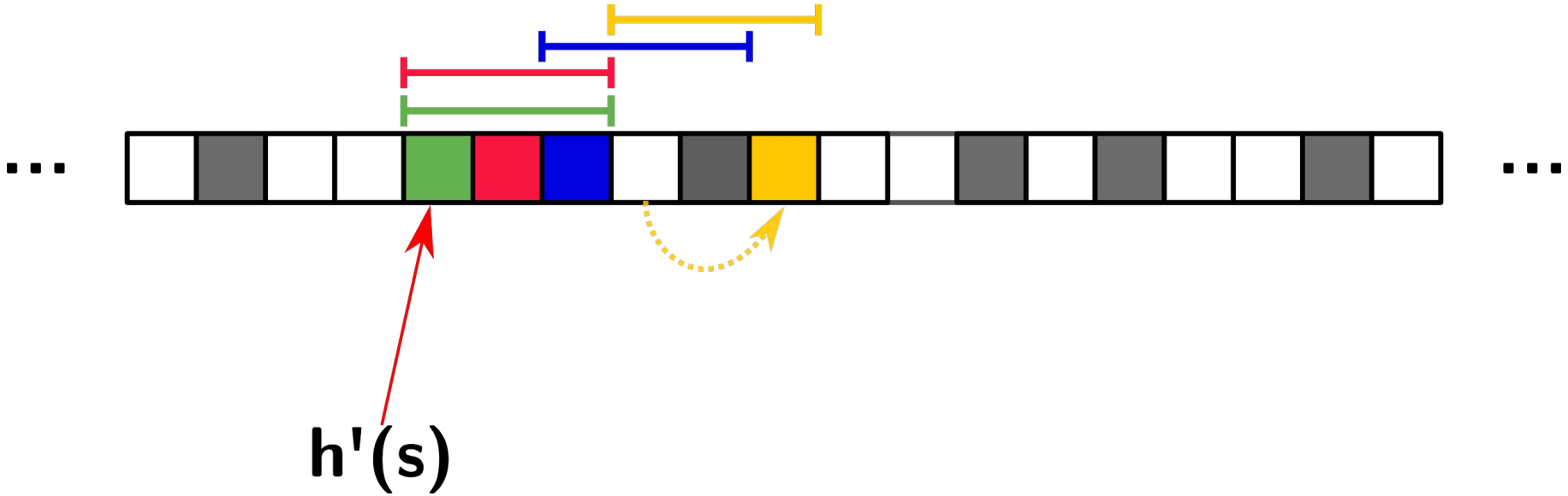
Example: $r = 3$



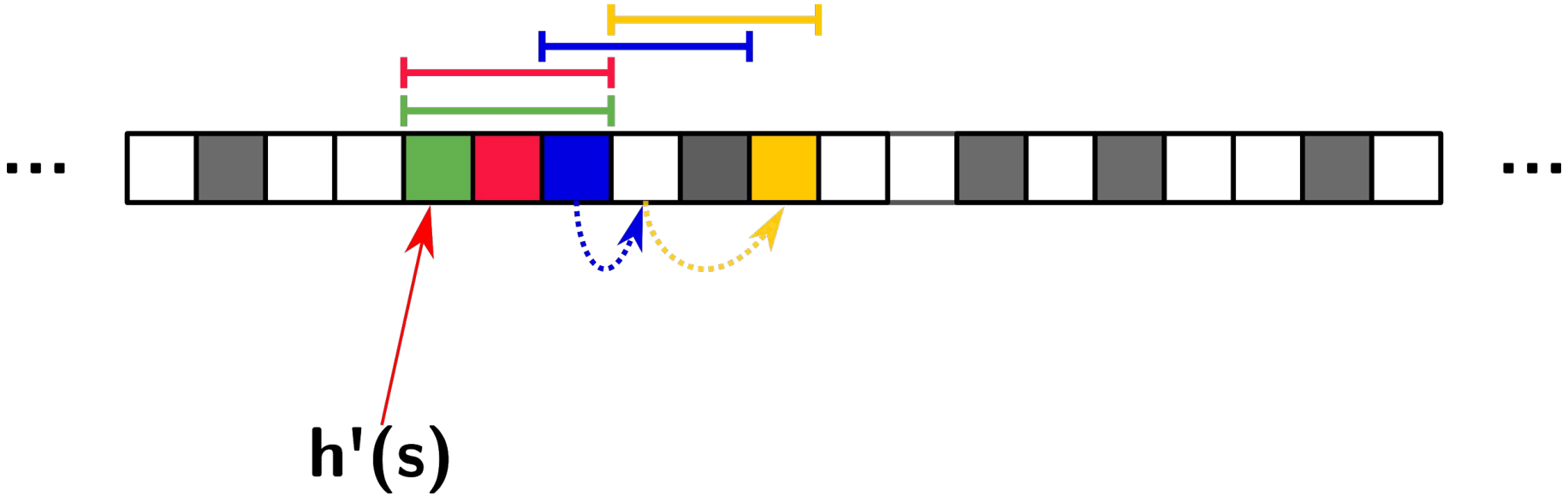
Hopscotch hashing



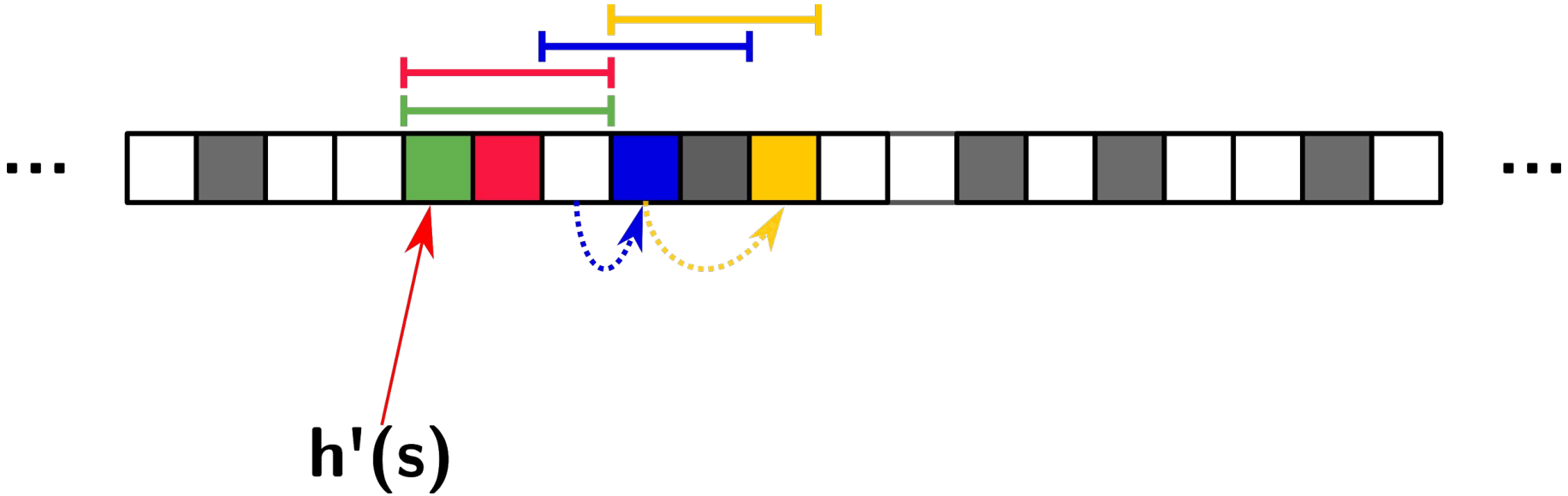
Hopscotch hashing



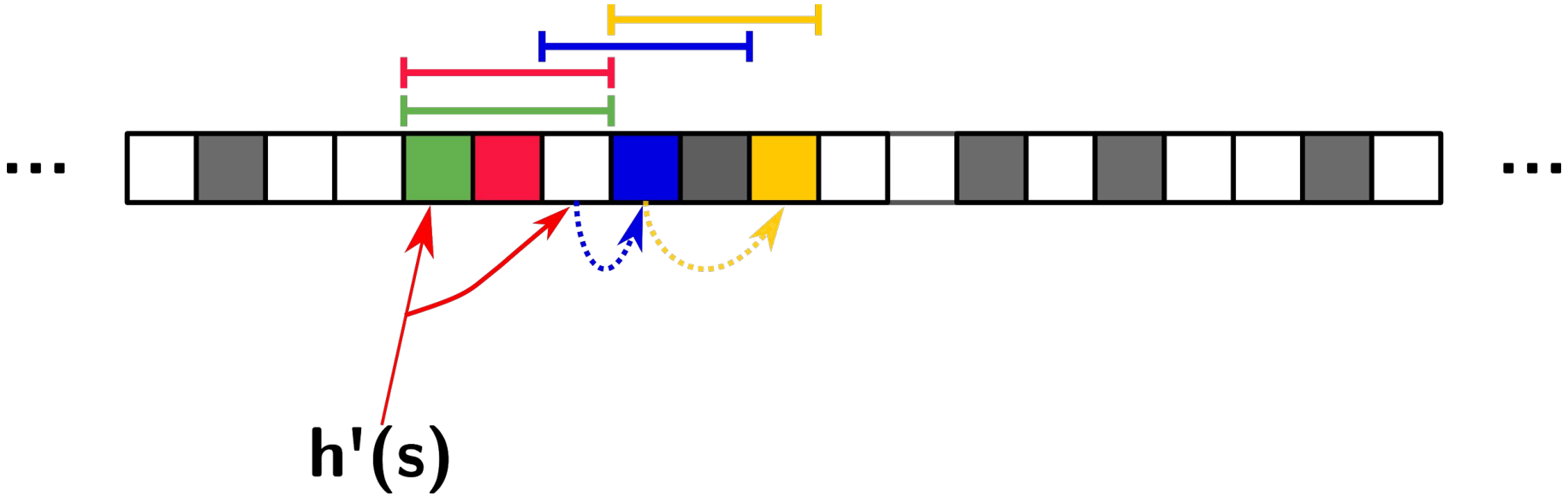
Hopscotch hashing



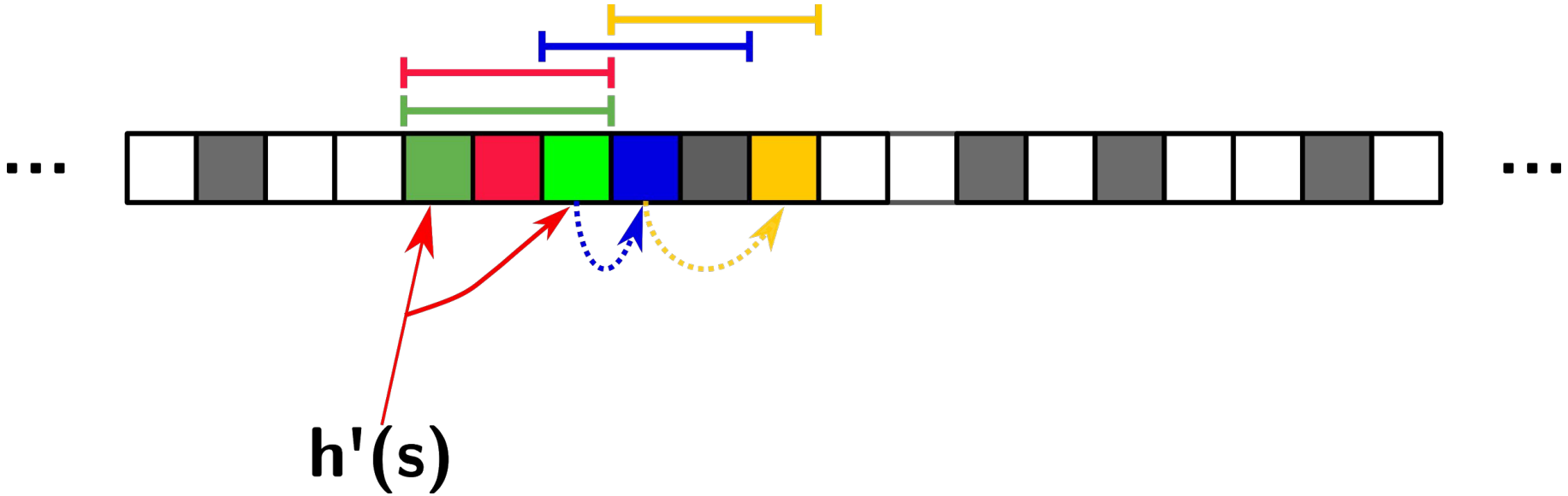
Hopscotch hashing



Hopscotch hashing



Hopscotch hashing



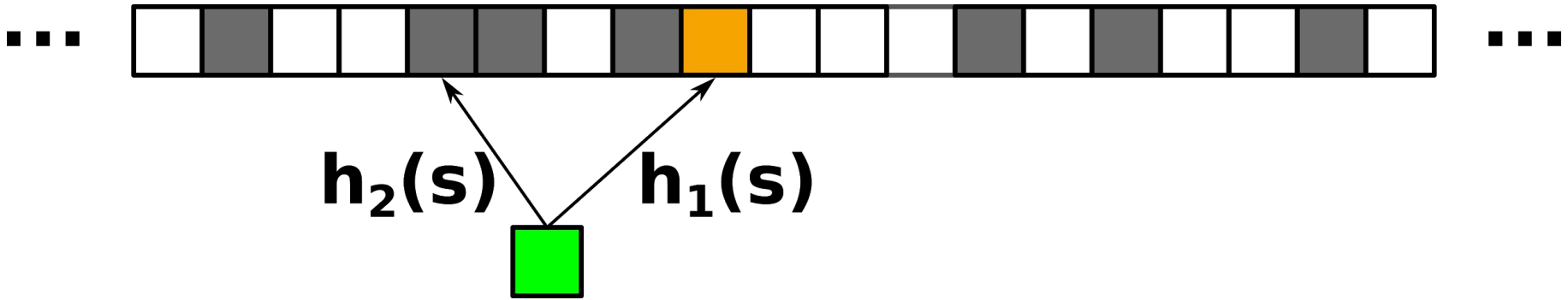
Cuckoo hashing

- Two hash functions h_1 and h_2
- Insert new element x with h_1
  - If position is occupied with element *y*, displace element *y*
  - Insert y with alternative hash function
- Parameter *w*: Maximum number of displacements

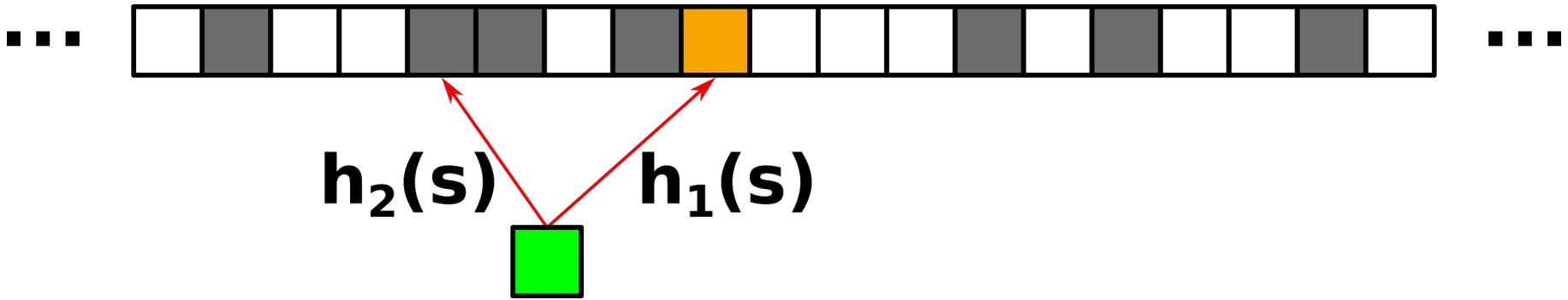
- Insert: O(*w*)
- Lookup: $2 \in O(1)$

Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. J. Algorithms, 51(2):122–144, 2004.

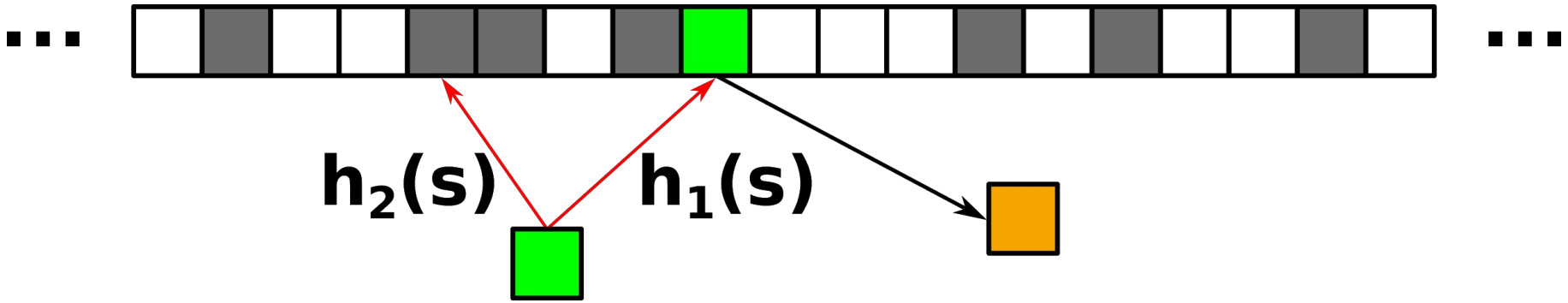
Cuckoo hashing



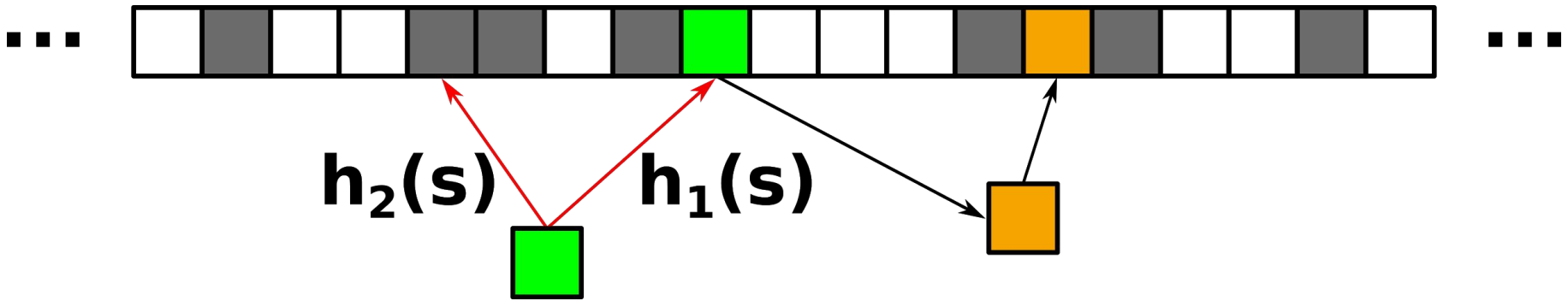
Cuckoo hashing



Cuckoo hashing



Cuckoo hashing

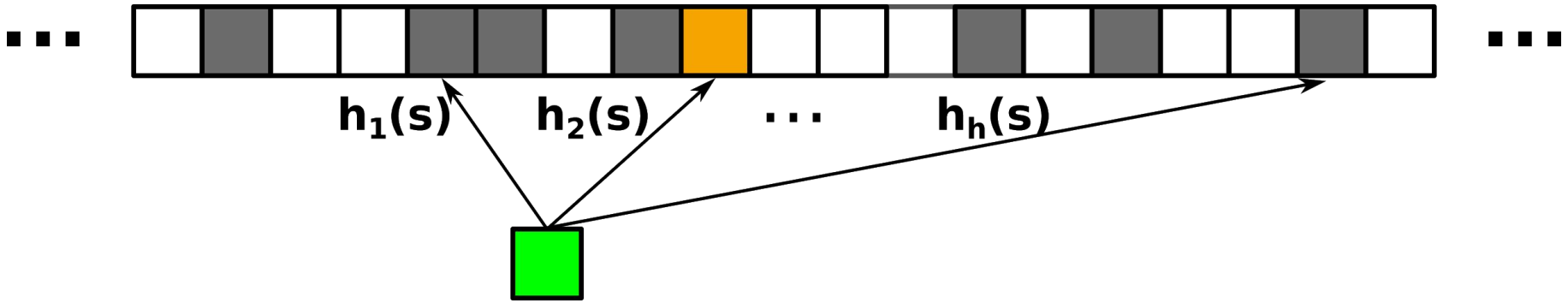


Cuckoo hashing with h hash functions

- Use $h \geq 2$ hash functions
  - Better (more uniform) distribution of the elements
- Different displacement strategies:
  - random walk
  - LSA

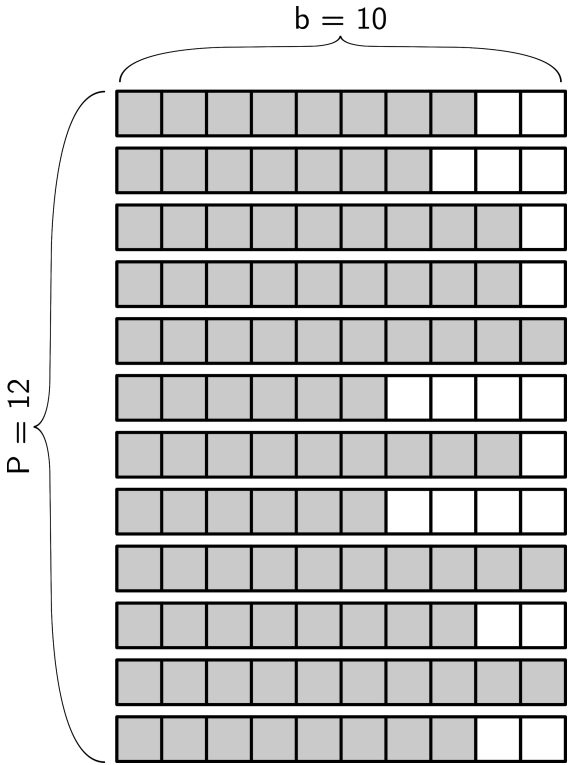
Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. Theory Comput. Syst., 38(2):229–248, 2005.

Cuckoo hashing with h hash functions



Cuckoo hashing with buckets

- Each position can store up to *b* elements ("**buckets**", "pages", "bins" of size *b*)
- Example: *P* = 12 buckets of size *b* = 10: hash table with 120 slots.
- Must search bucket to find element x, until found or empty slot encountered, or entire bucket has been searched



Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. Theor. Comput. Sci., 380(1-2):47–68, 2007.

(h,b) Cuckoo hashing

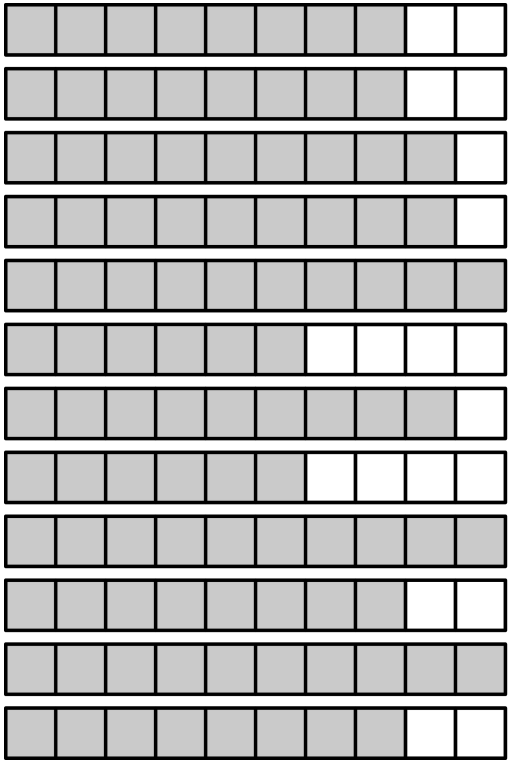
- Combination of
  - buckets with size b and
  - h hash functions
- Different insertion strategies
  - Random walk
  - Breadth first search
  - LSA_{max}
  - Cuckopt
  - …

(h,b) Cuckoo hashing: Random walk

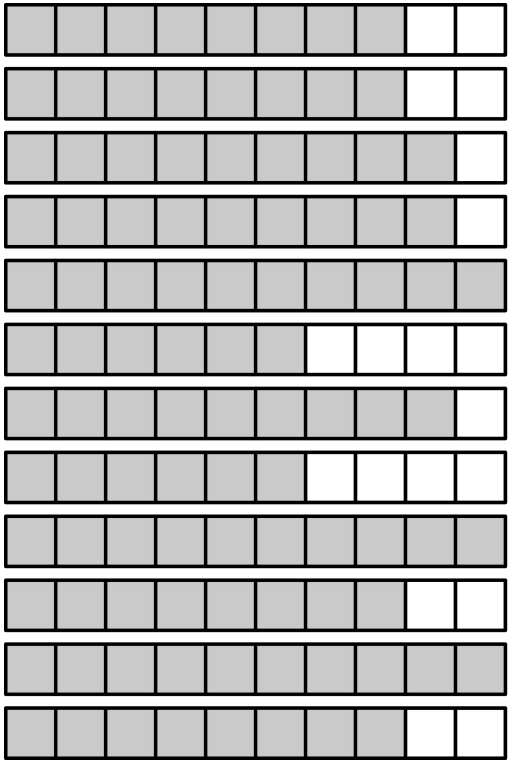
- Check all hash functions for an empty slot
- If no empty slot exits
  - Choose one element and replace it
  - Insert the replaced element
- Maximum of w replacements

- Insert: O(w)
- Lookup: O(h)

(h,b) Cuckoo hashing: Random walk

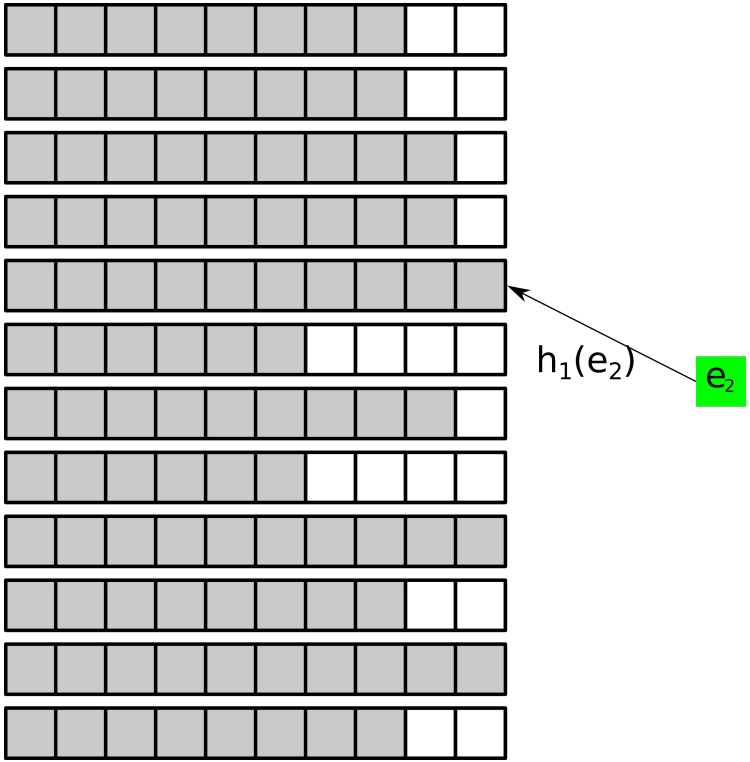


(h,b) Cuckoo hashing: Random walk

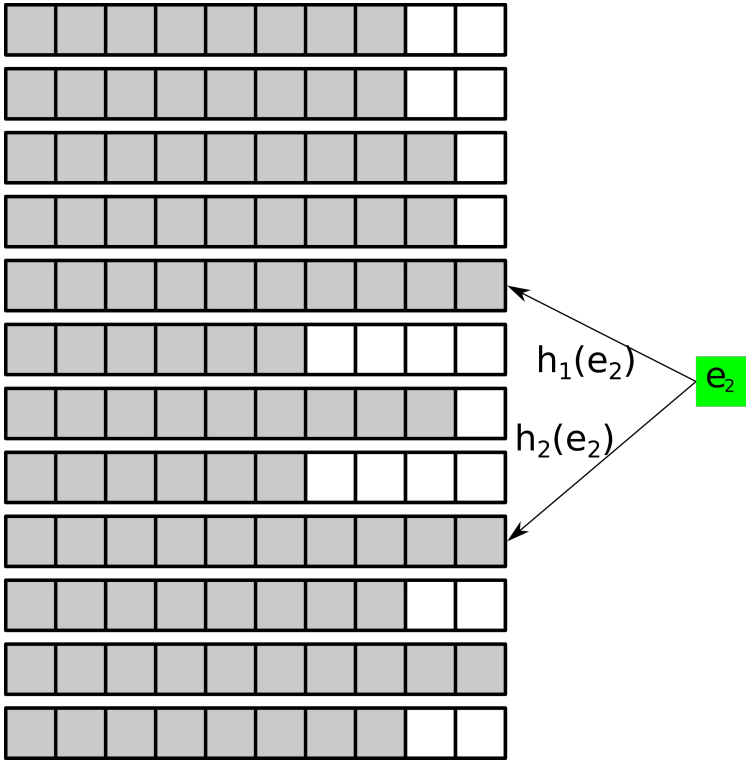


e_2

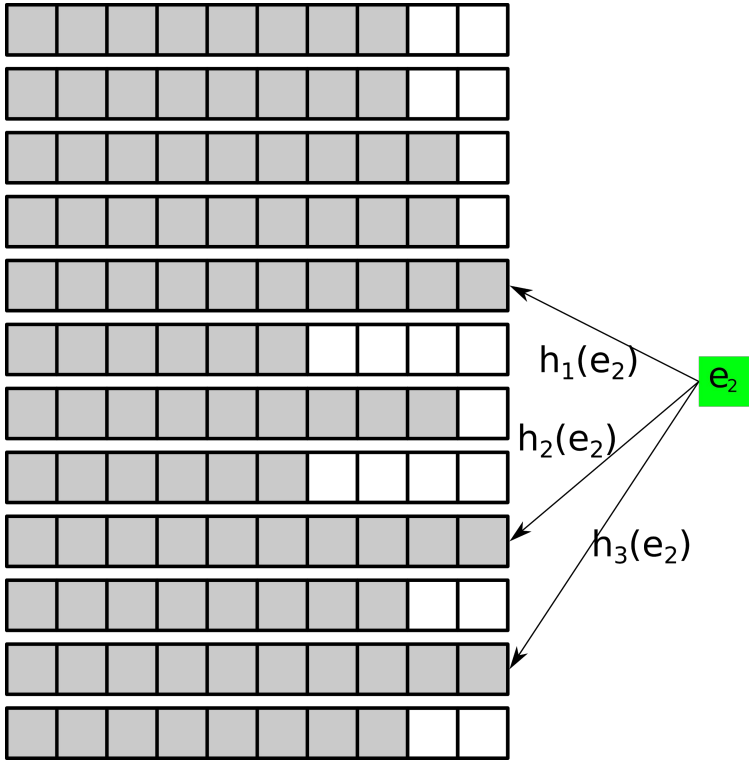
(h,b) Cuckoo hashing: Random walk



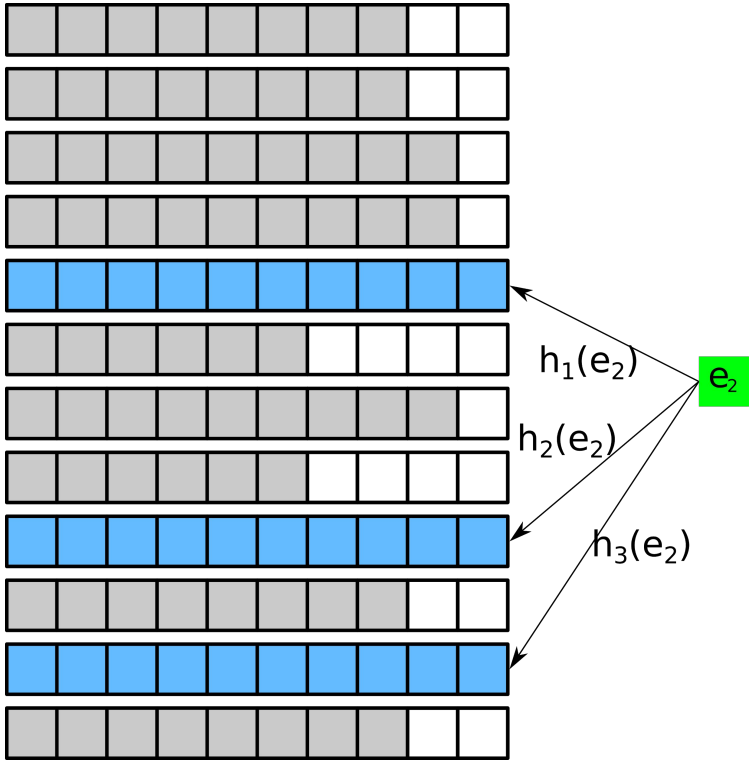
(h,b) Cuckoo hashing: Random walk



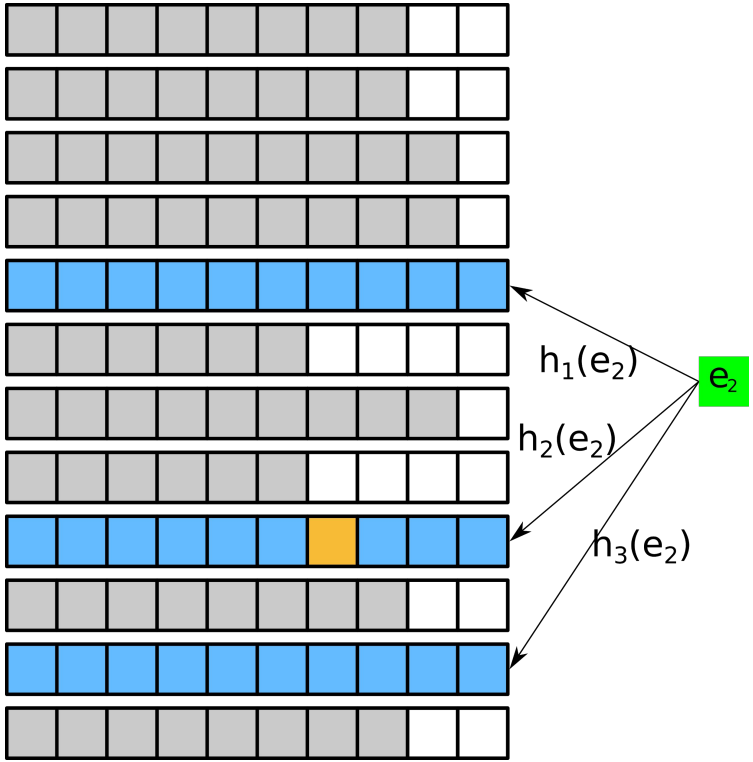
(h,b) Cuckoo hashing: Random walk



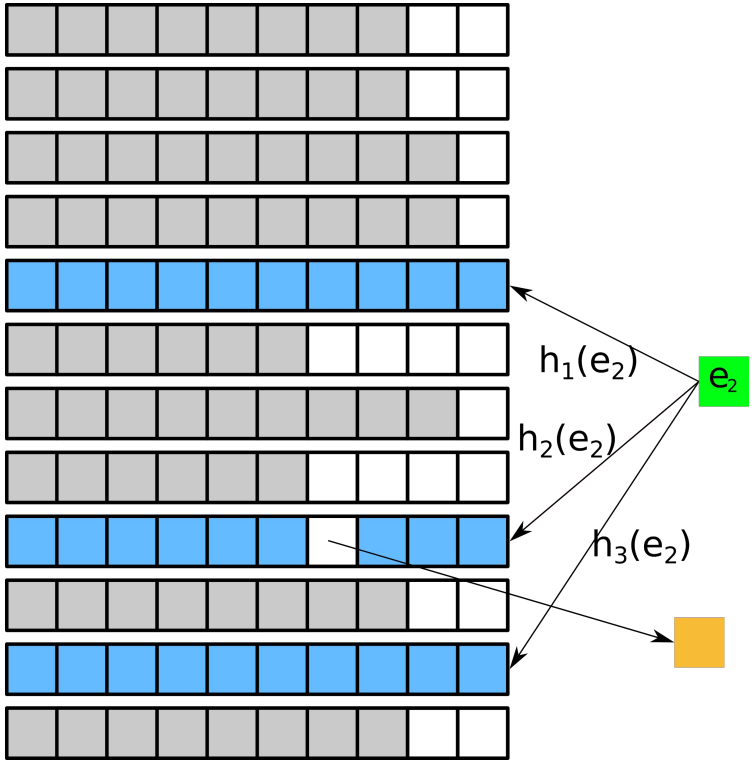
(h,b) Cuckoo hashing: Random walk



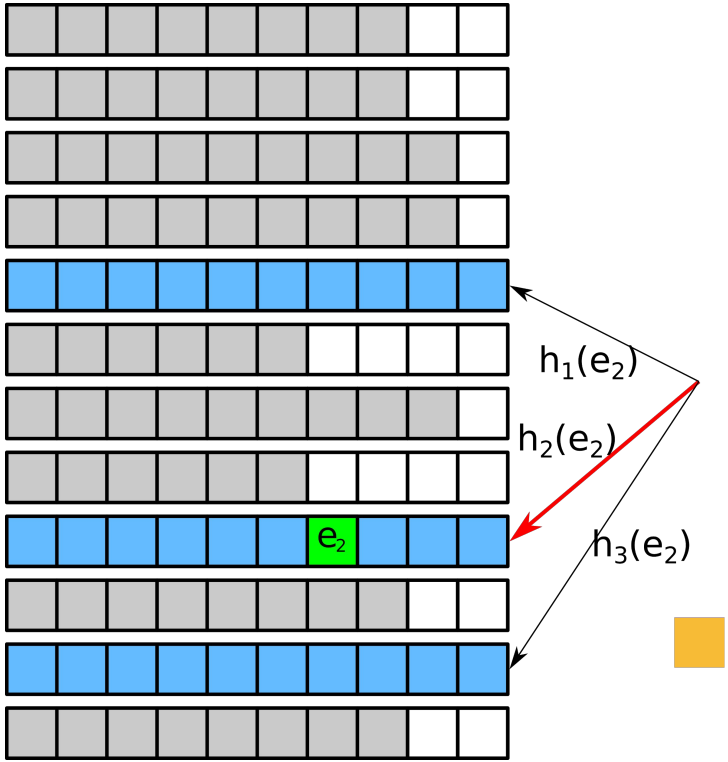
(h,b) Cuckoo hashing: Random walk



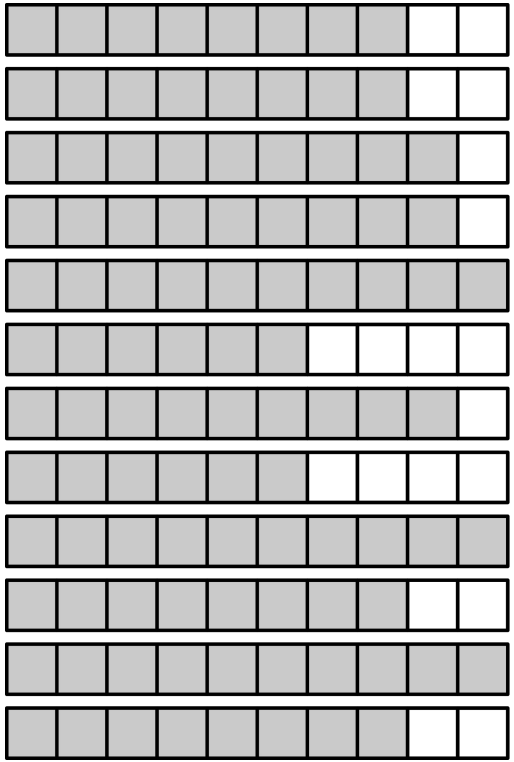
(h,b) Cuckoo hashing: Random walk



(h,b) Cuckoo hashing: Random walk



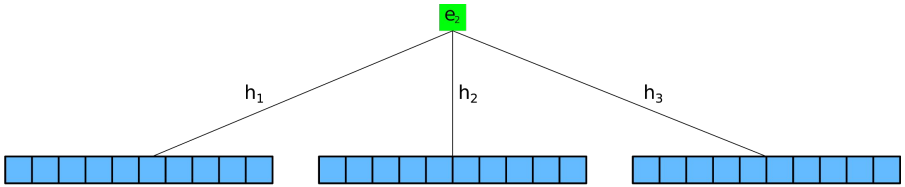
(h,b) Cuckoo hashing: Random walk

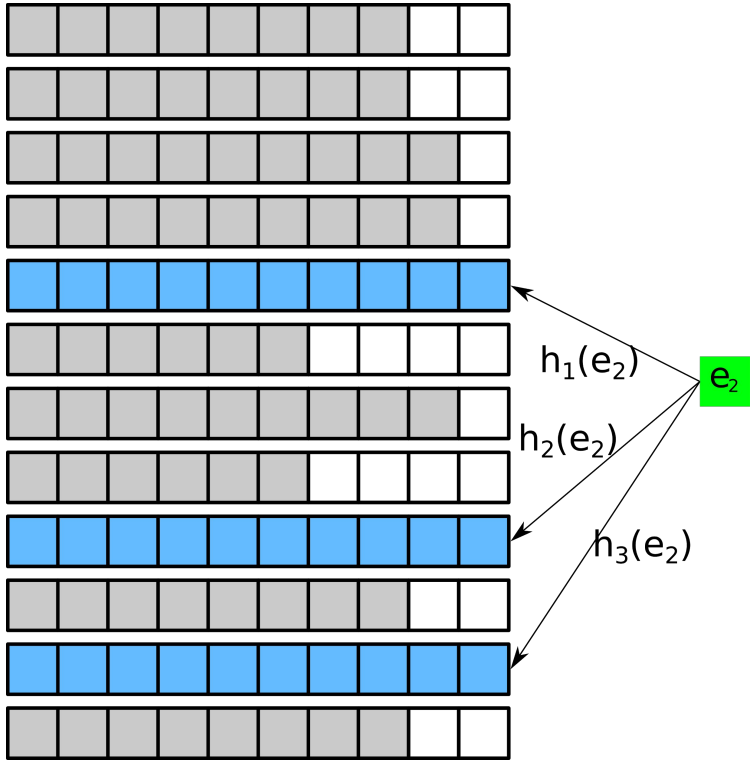


(h,b) Cuckoo hashing: BFS

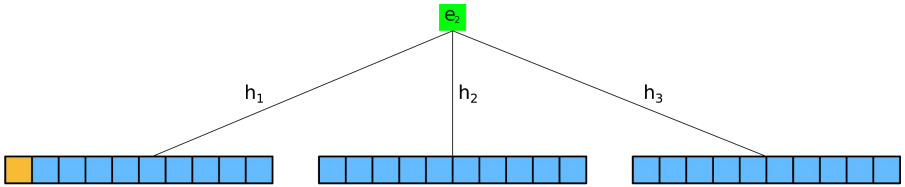
- BFS always finds a path to insert a new element (if one exists)
- BFS starts at the new elements
  - Check all reachable buckets
  - if there exists no empty slot:
    - Follow all elements in these buckets to their alternative locations

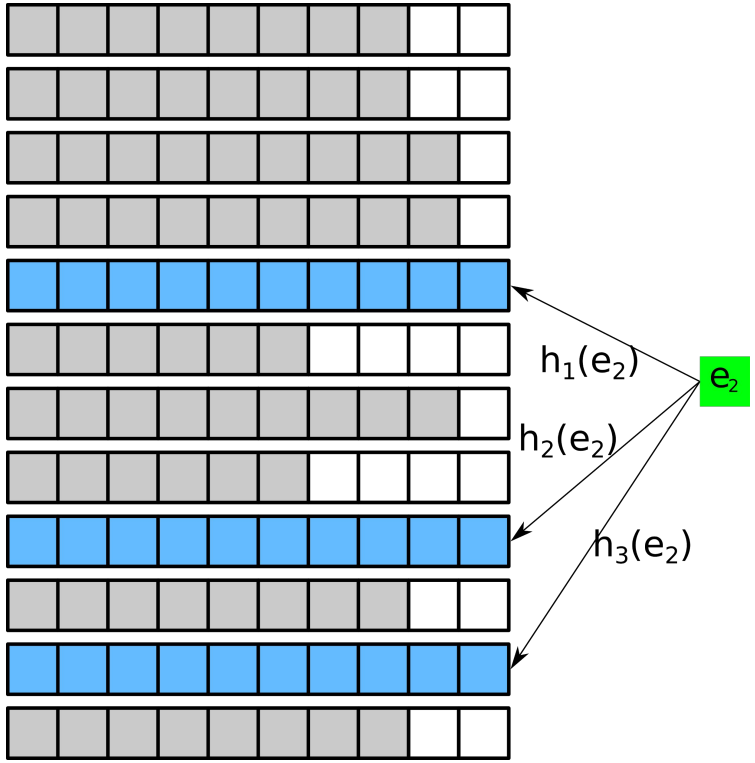
(h,b) Cuckoo hashing: BFS



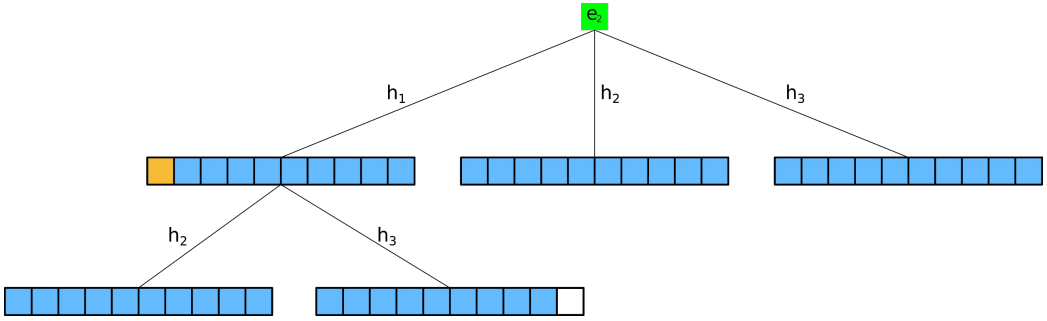


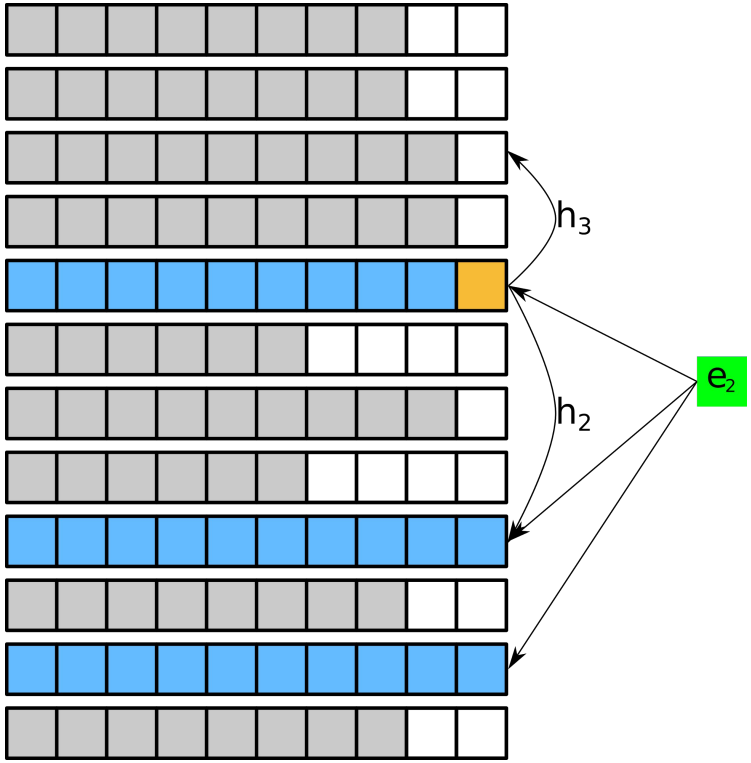
(h,b) Cuckoo hashing: BFS



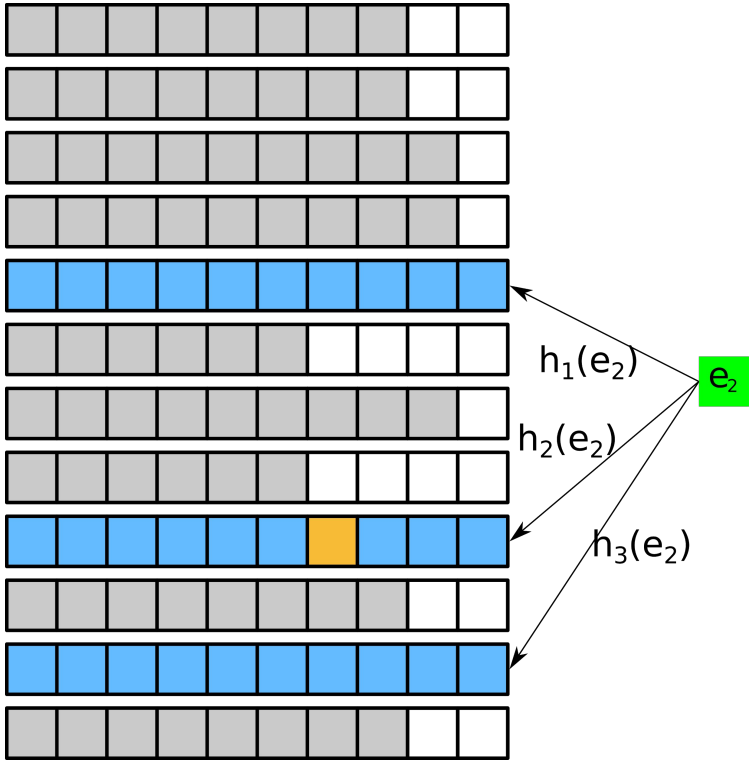


(h,b) Cuckoo hashing: BFS





(h,b) Cuckoo hashing: Query



Achievable loads for (h,b) Cuckoo hashing

Classical (2,1) Cuckoo hashing: 0.5

- Table from paper with load thresholds

- Expected lookups for random walk (Graphs) for (3, b)

- Werbung für Poster

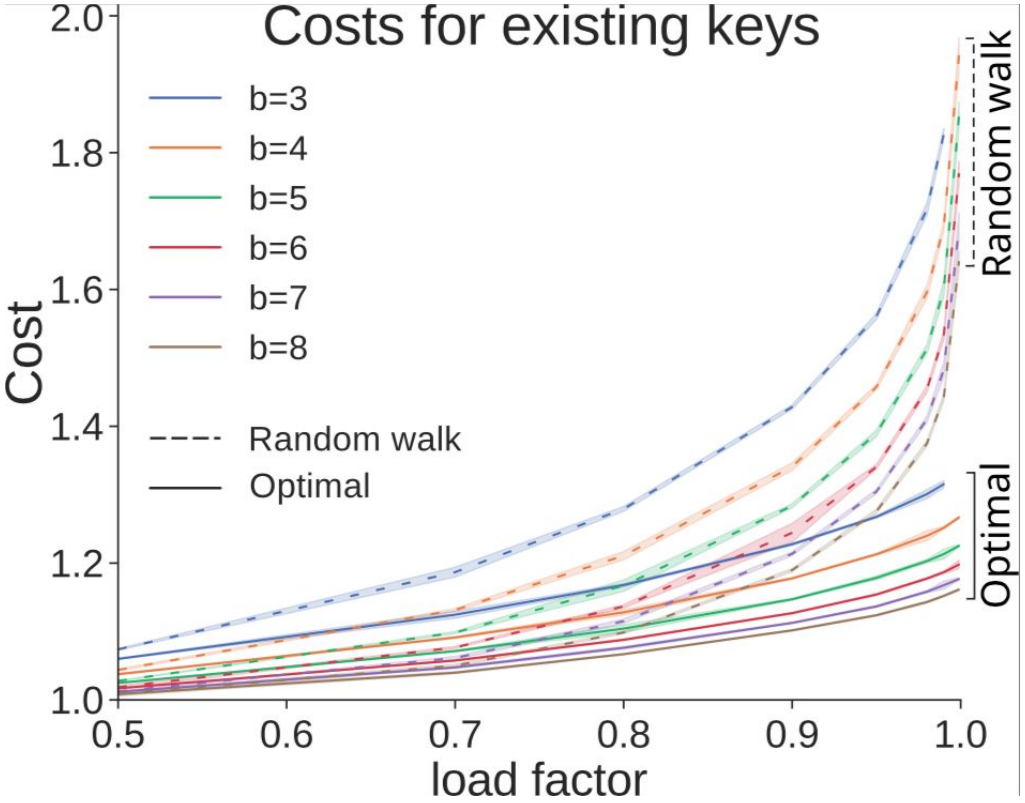
Achievable loads for (h,b) Cuckoo hashing

b ***h***	**2**	**3**	**4**	**5**	**6**	**7**
1	0.5	0.9179352767	0.9767701649	0.9924383913	0.9973795528	0.9990637588
2	0.8970	0.9882014140	0.9982414840	0.9997243601	0.9999568737	0.9999933439
3	0.95915	0.9972857393	0.9997951434	0.9999851453	0.9999989795	0.9999999329
4	0.98037	0.9992531564	0.9999720661	0.9999990737	0.9999999721	0.9999999992

S. Walzer. Load thresholds for cuckoo hashing with overlapping blocks. ICALP 2018.

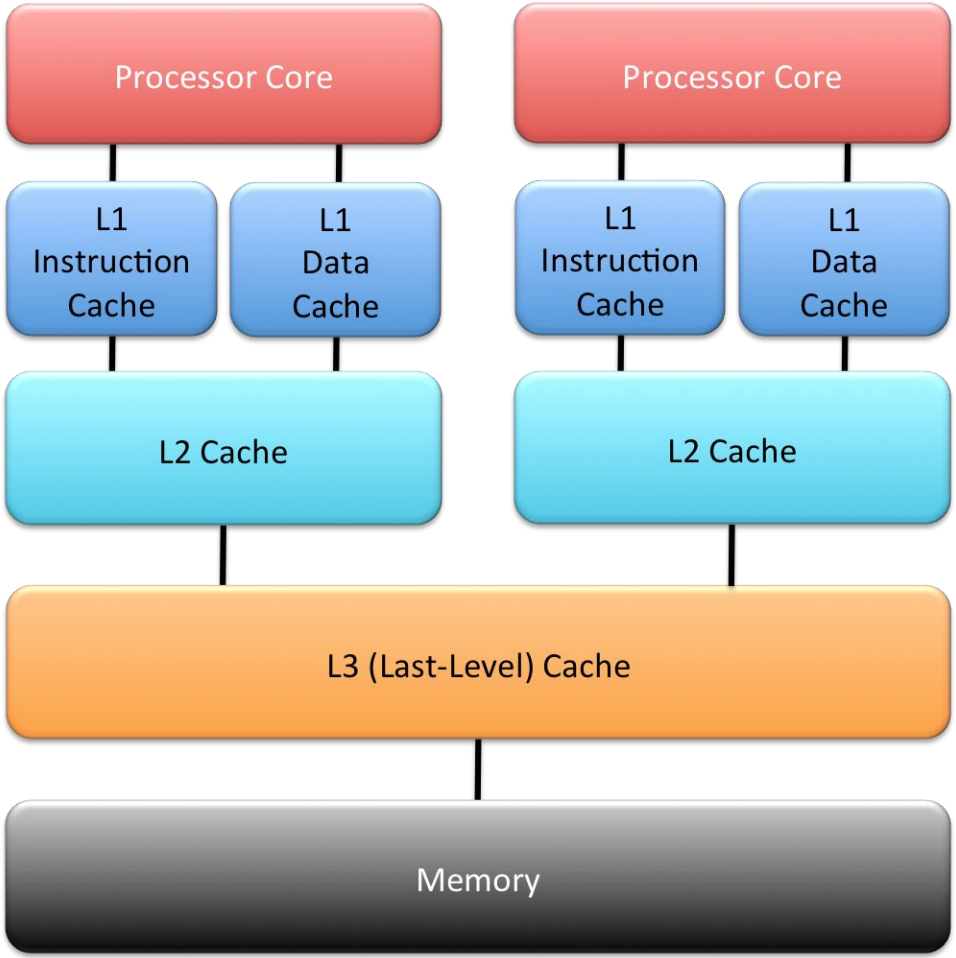
Costs for (h=3, b) Cuckoo hashing with random walk

- Bucket size $b \in \{3, \ldots, 8\}$
- Load factor
  $\alpha \in \{0.5, 0.7, 0.8, 0.9, 0.95, 0.99, 0.999\}$
- Cost: expected number of memory lookups per lookup of a present key
- Even with $h = 3$, close to 1 lookup suffices on average (loads < 0.95). Worst case of 3 happens rarely.

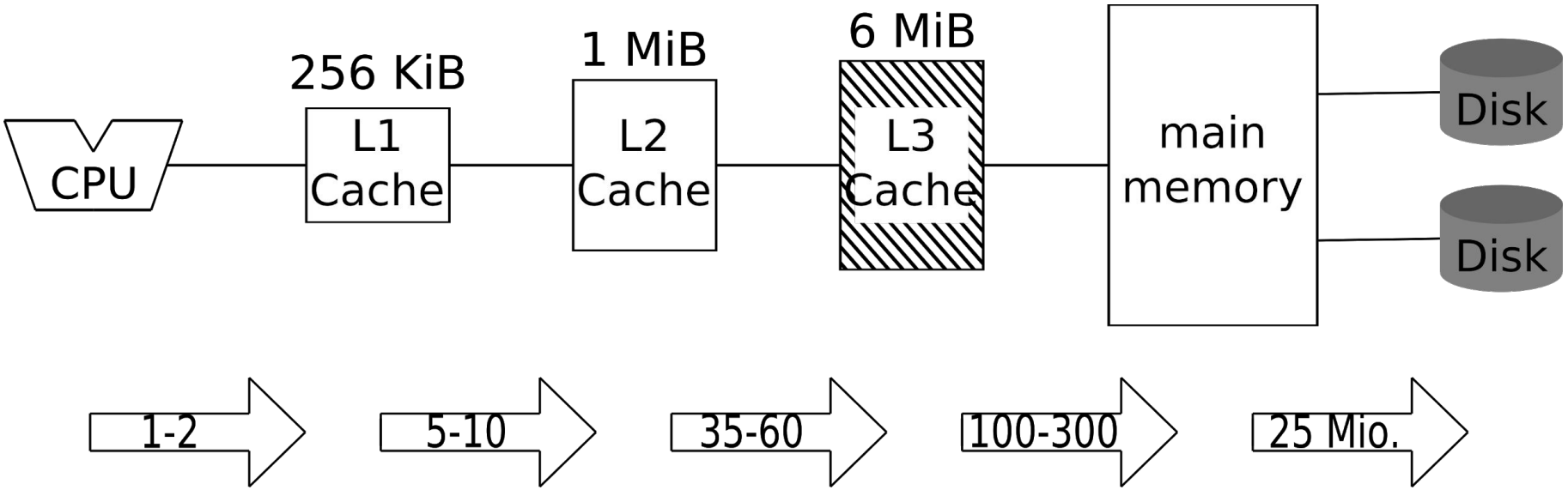


The role of modern hardware

CPU caches

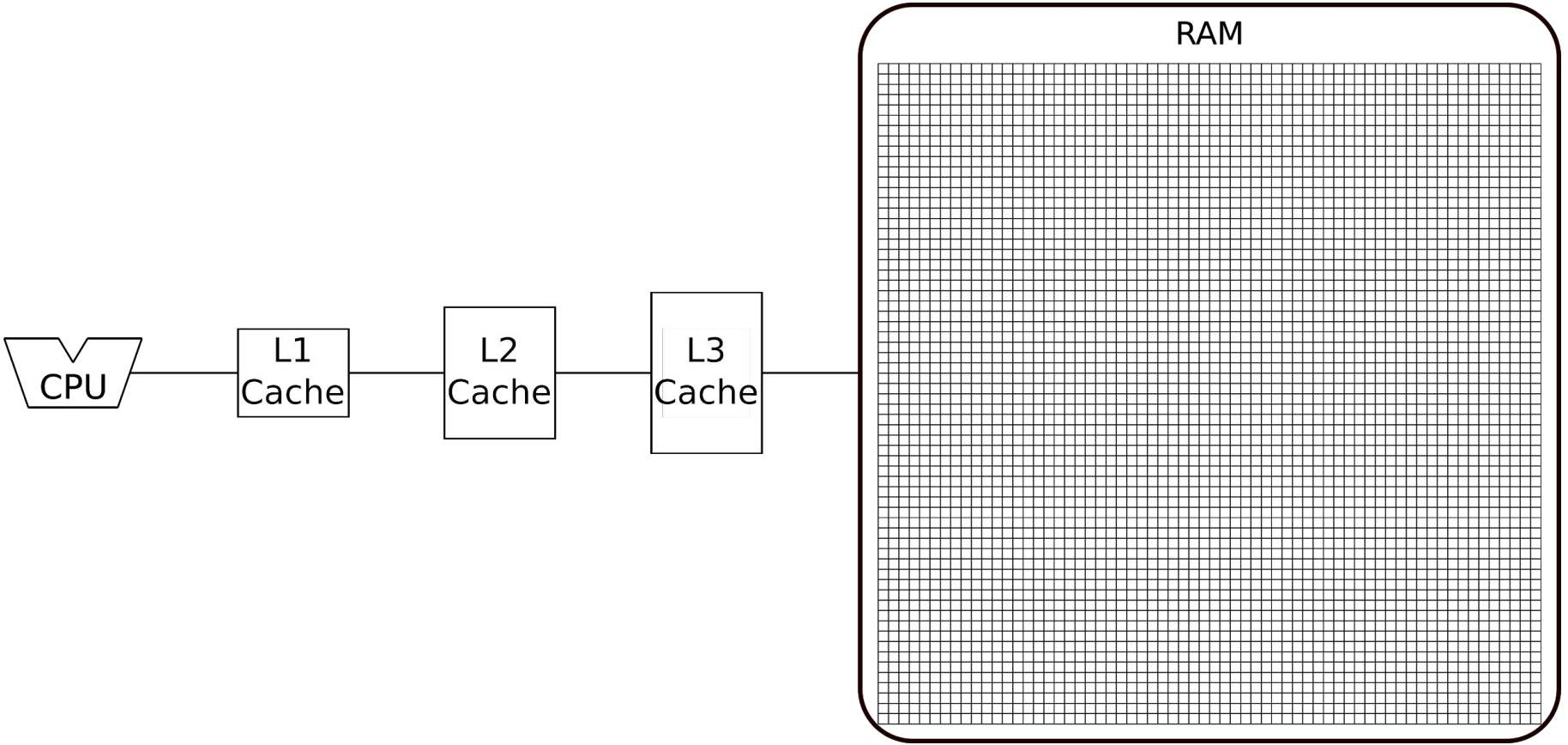

Processor Core
Processor Core
L1 Instruction Cache
L1 Data Cache
L1 Instruction Cache
L1 Data Cache
L2 Cache
L2 Cache
L3 (Last-Level) Cache
Memory

CPU caches

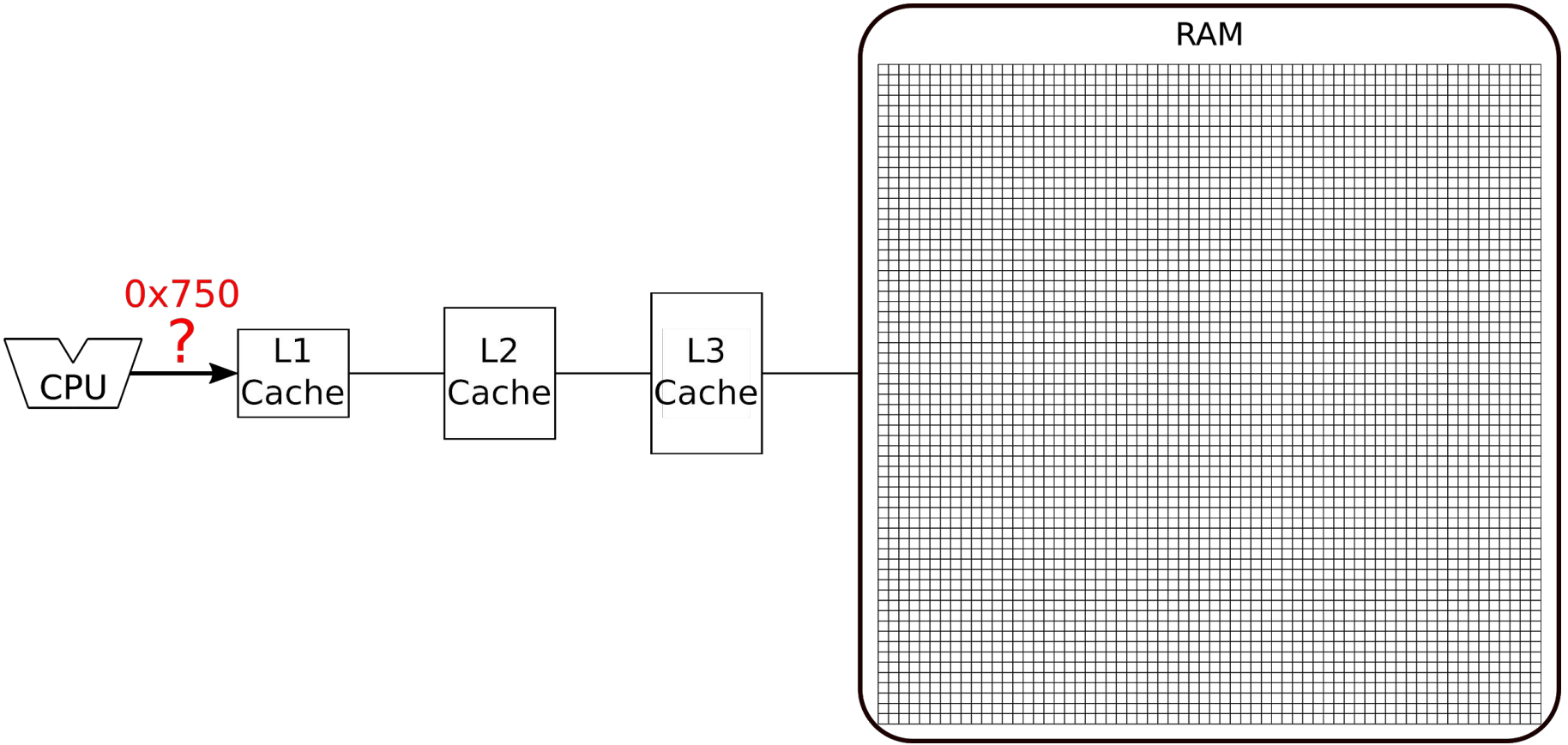


This and following illustrations by Uriel Elias Wiebelitz (TU Dortmund)

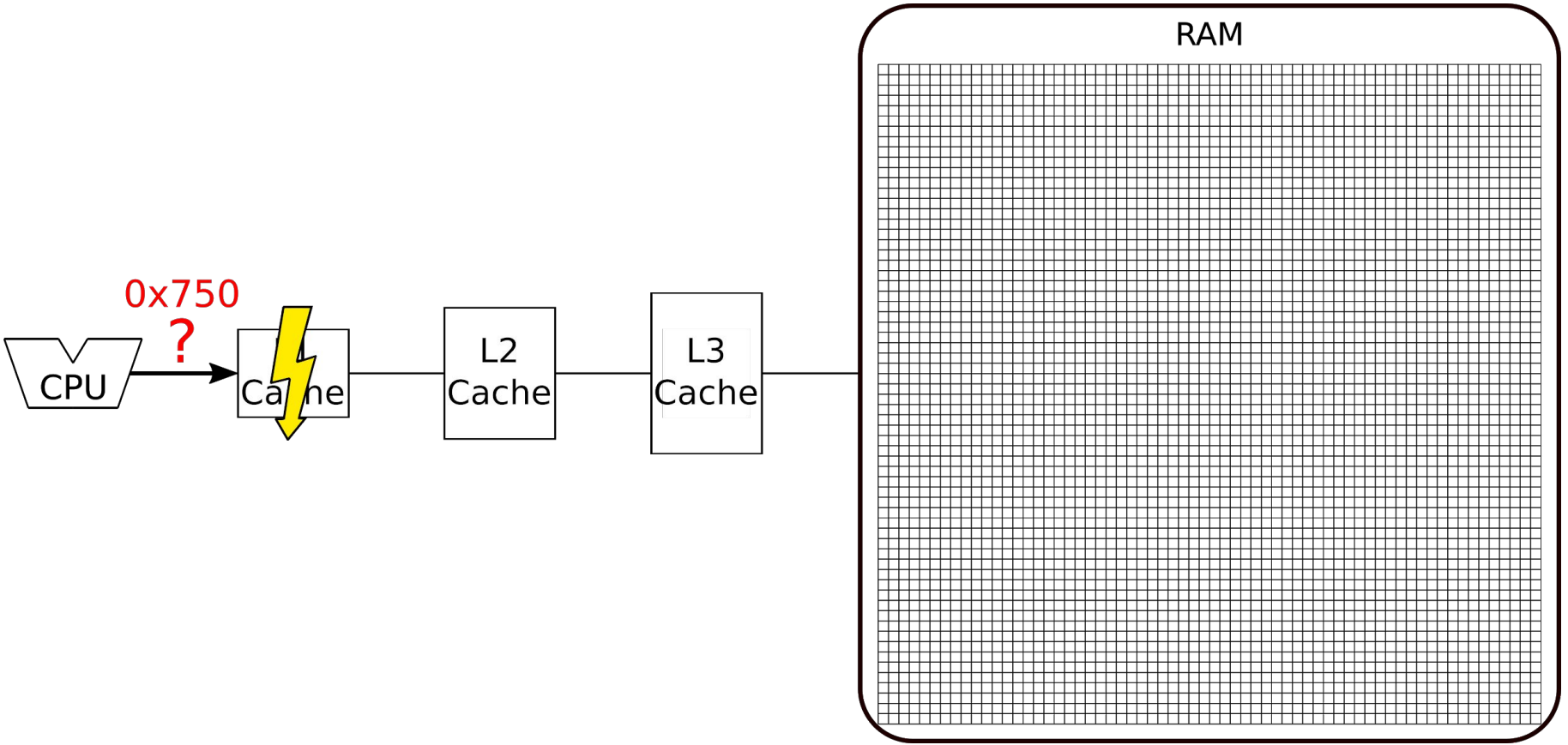
Cache line



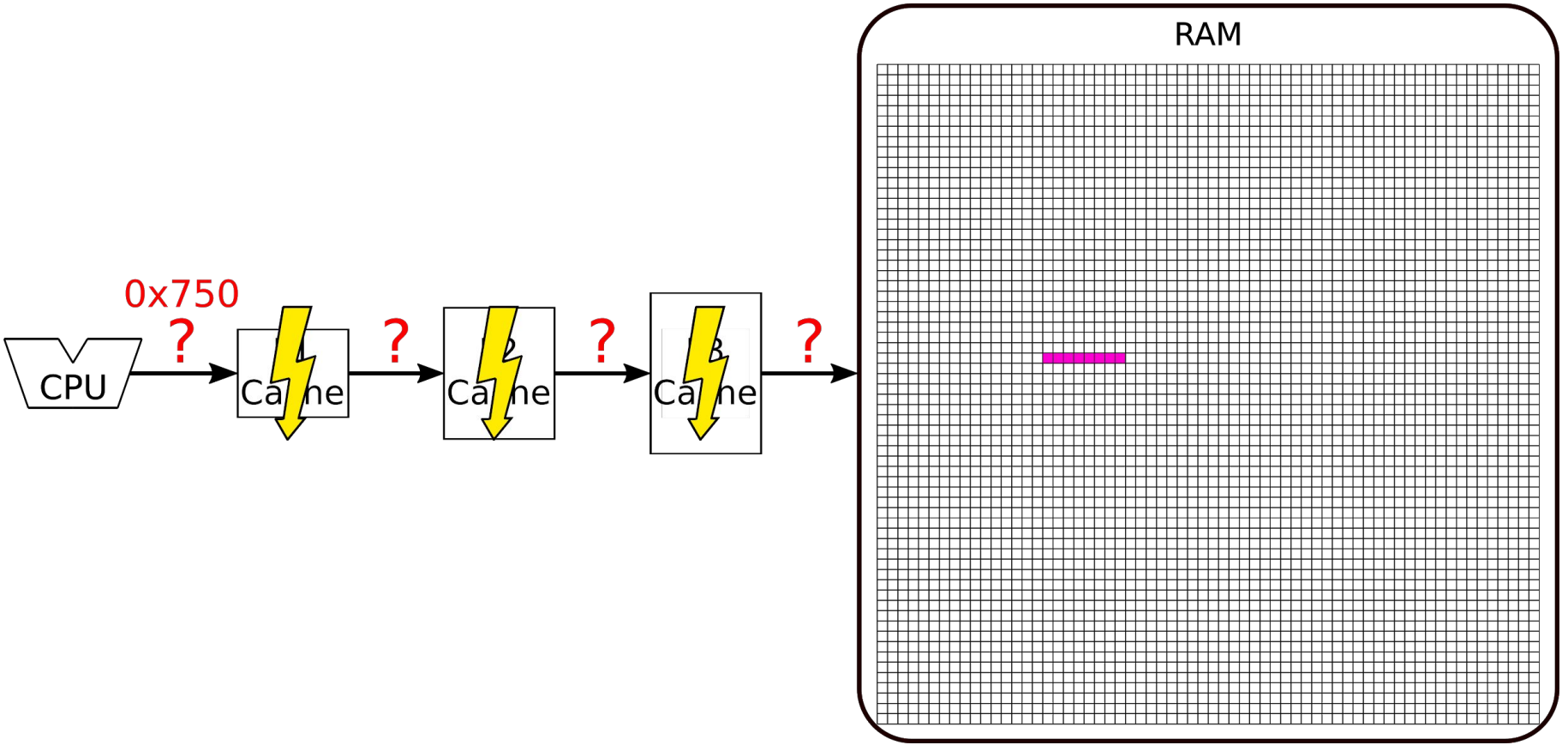
Cache line



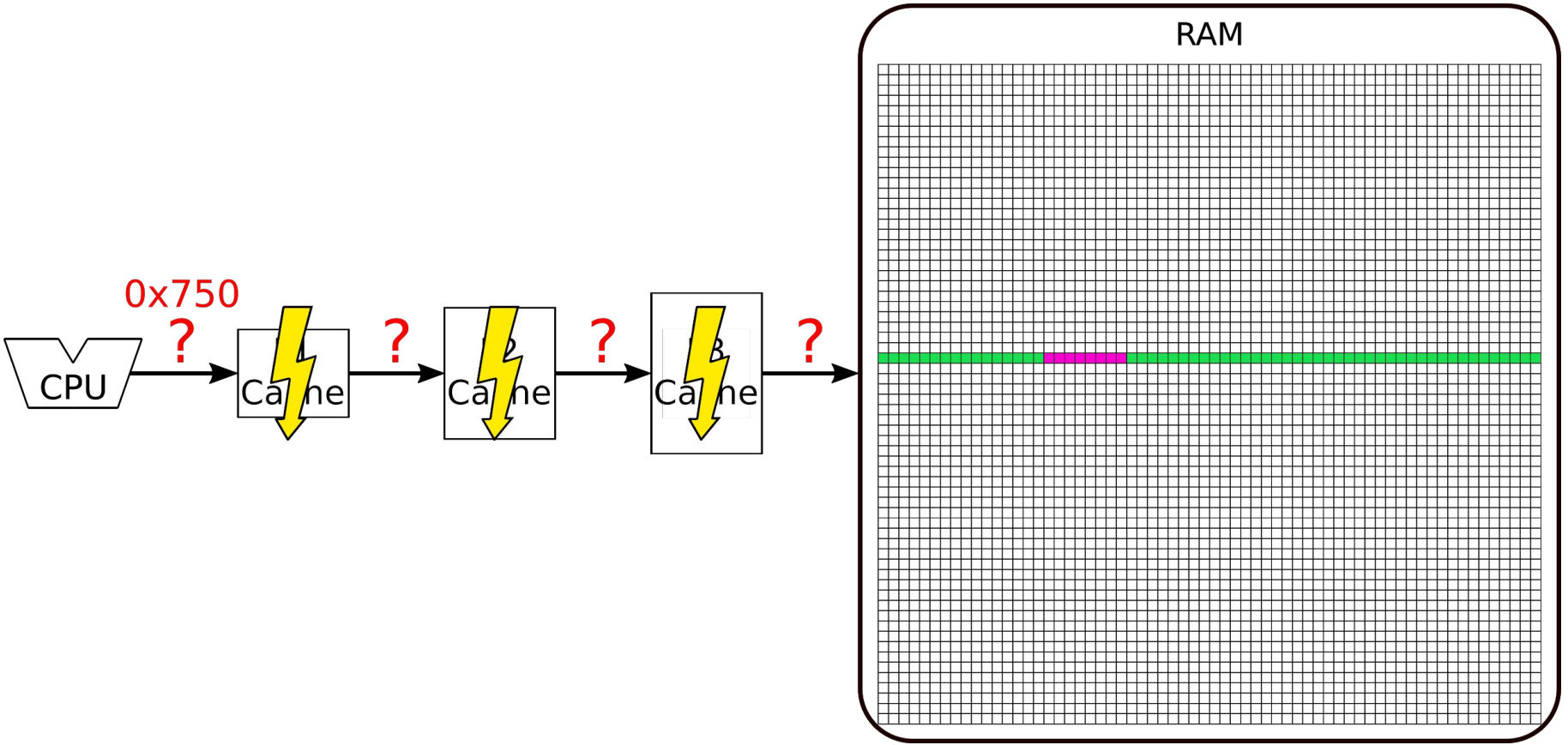
Cache line



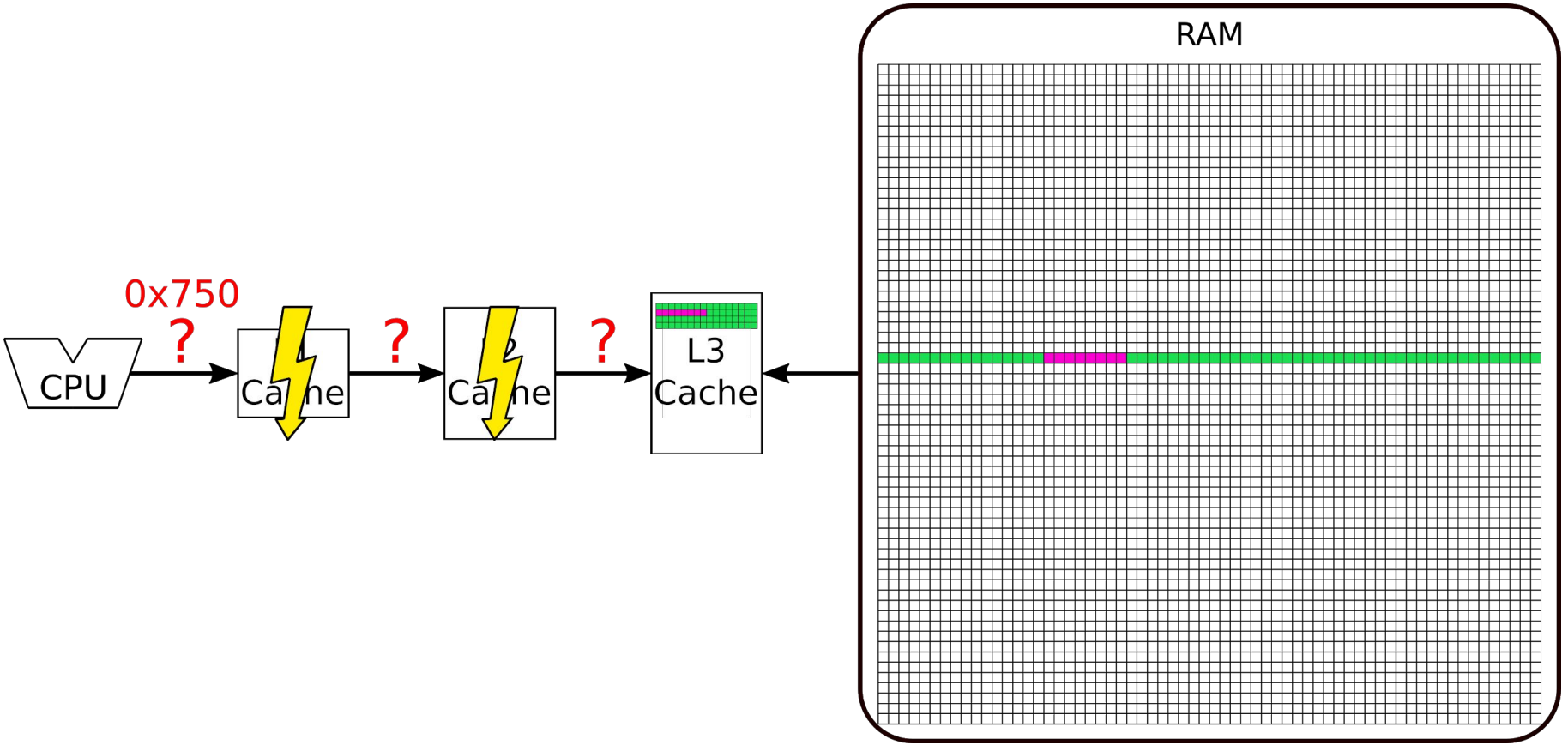
Cache line



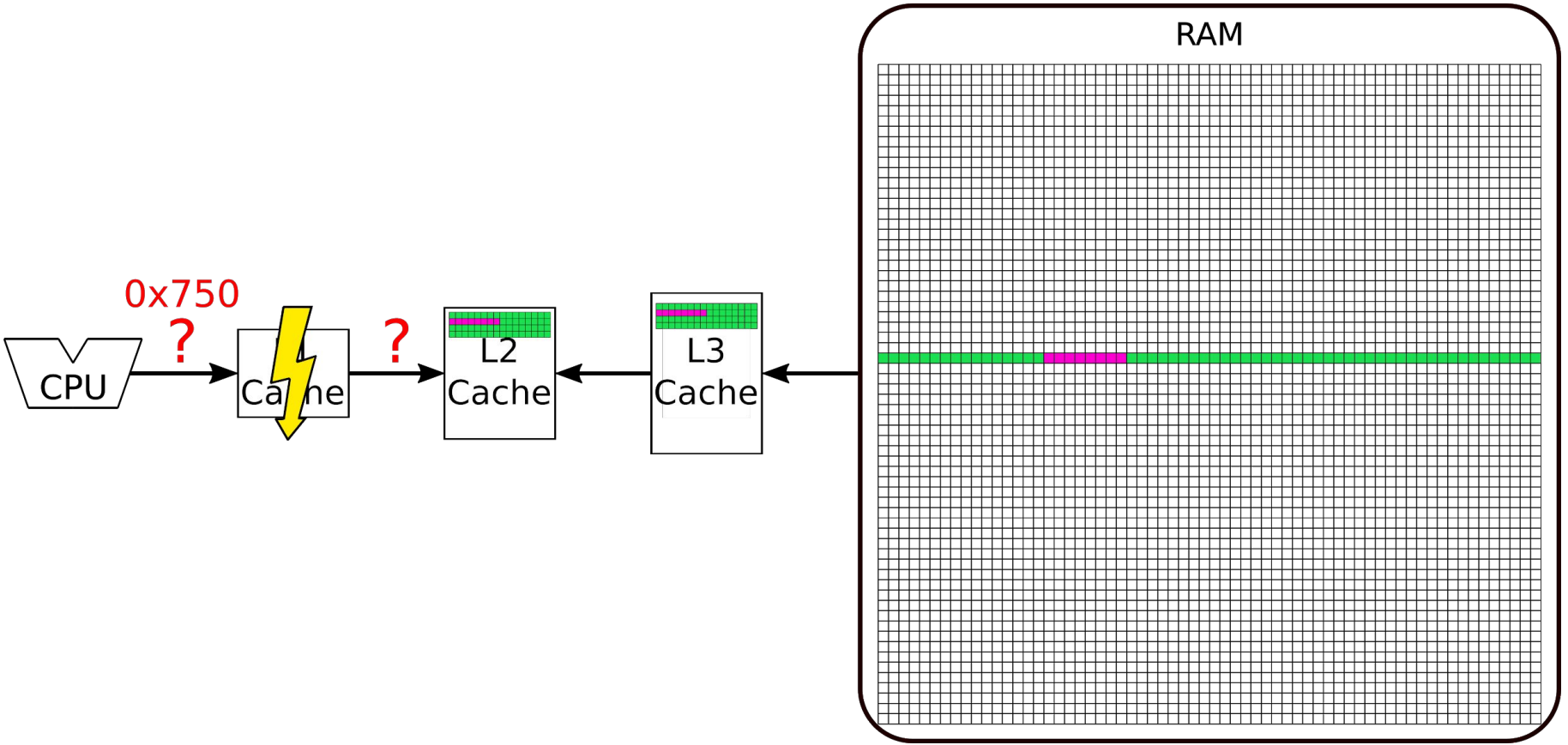
Cache line



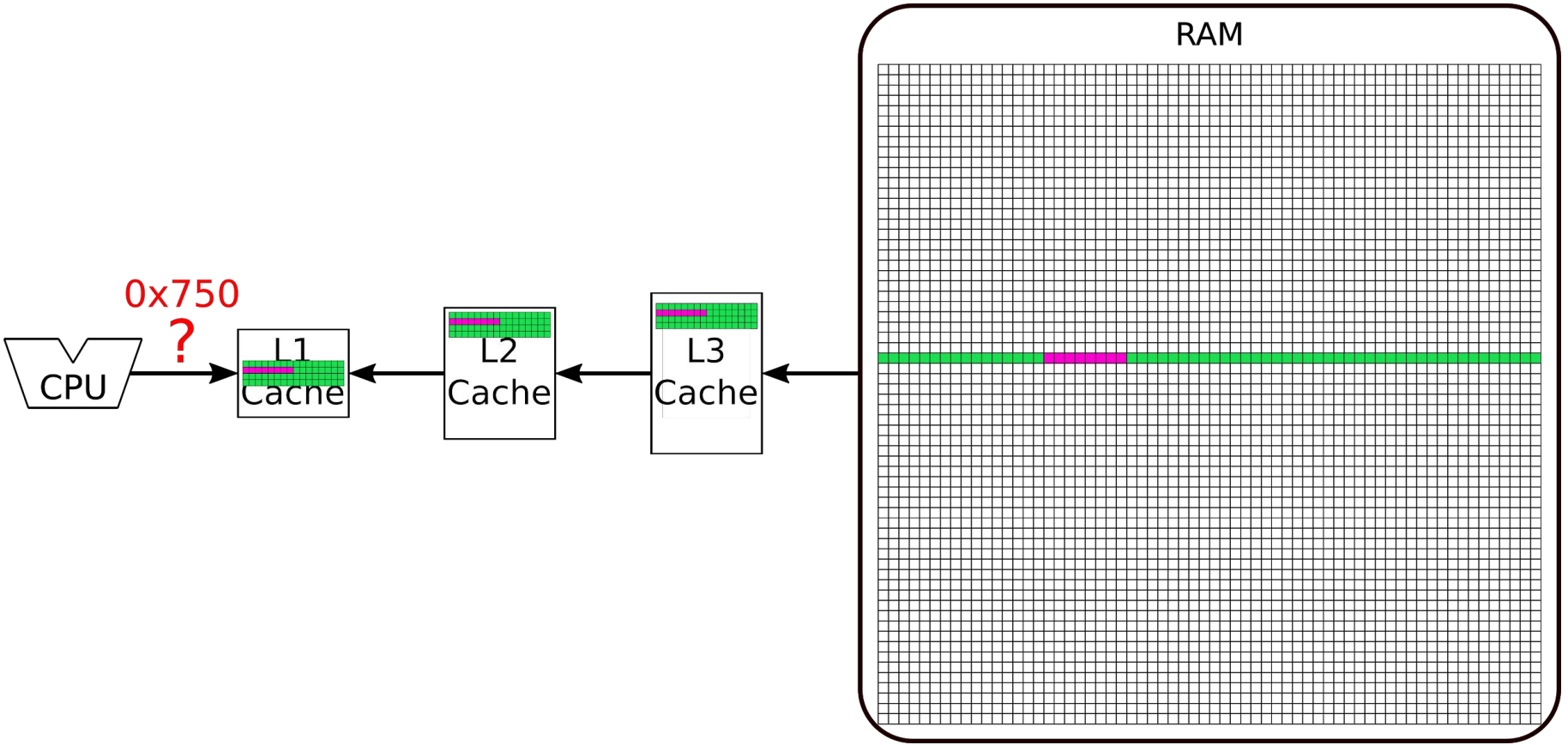
Cache line



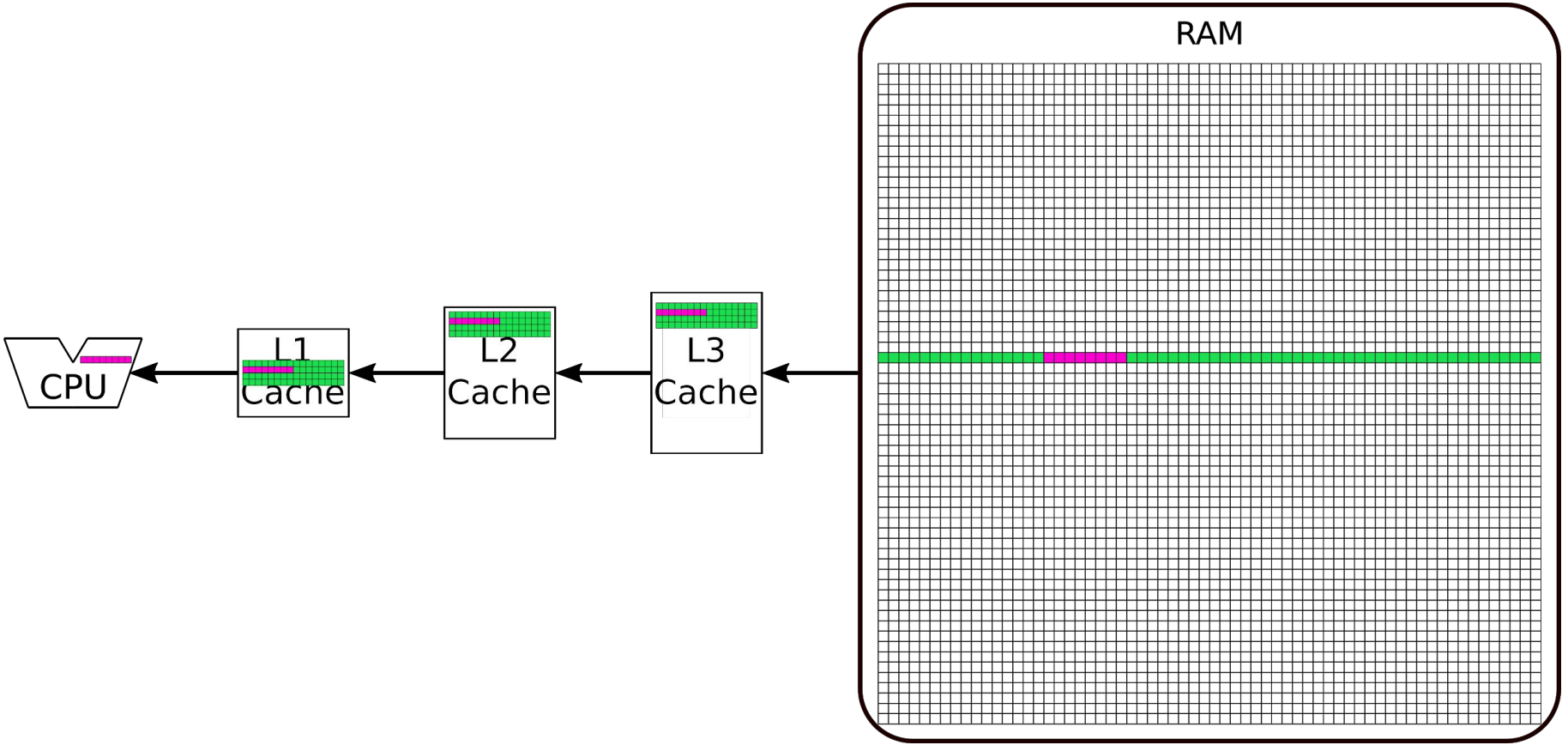
Cache line



Cache line



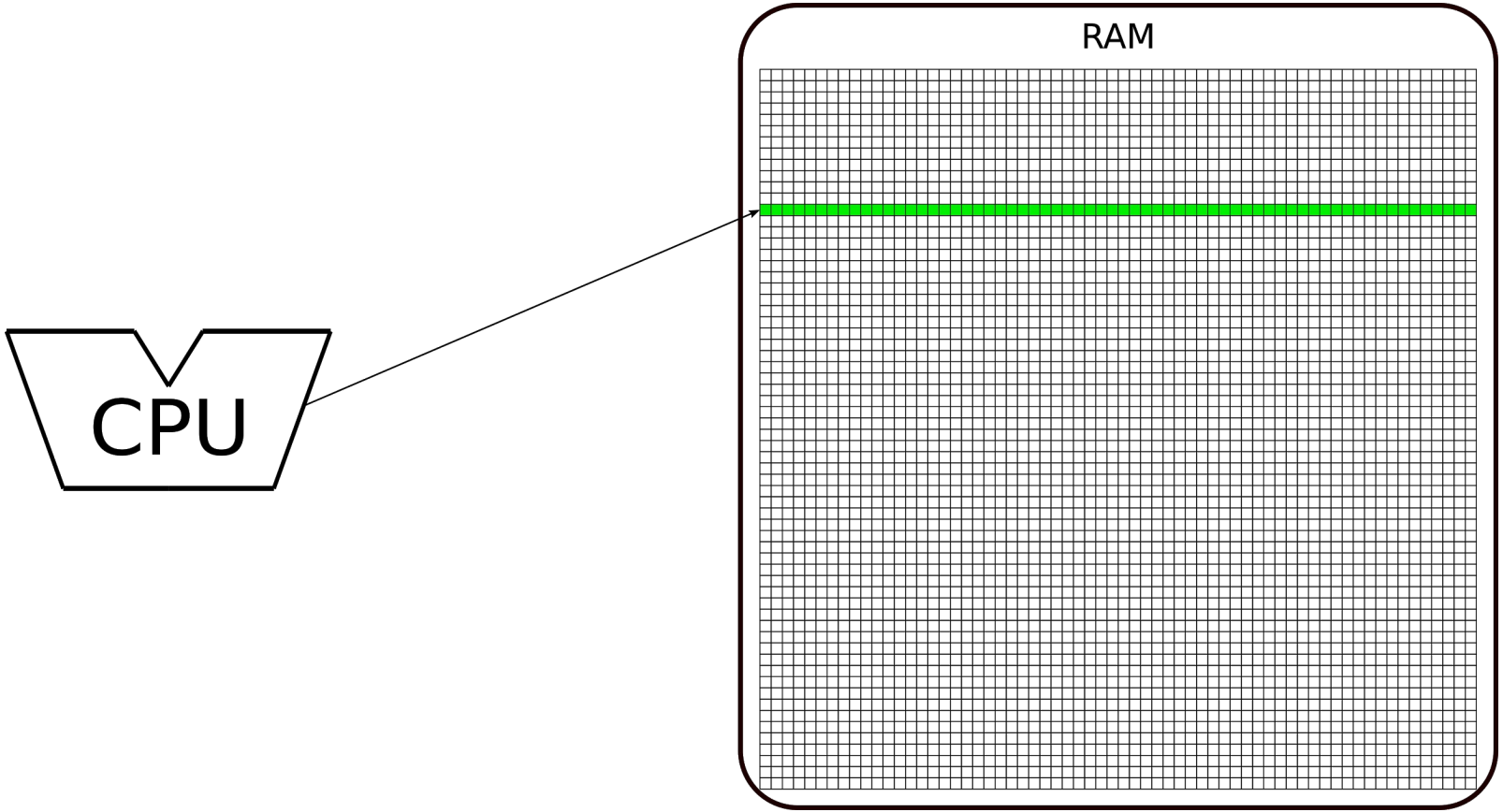
Cache line



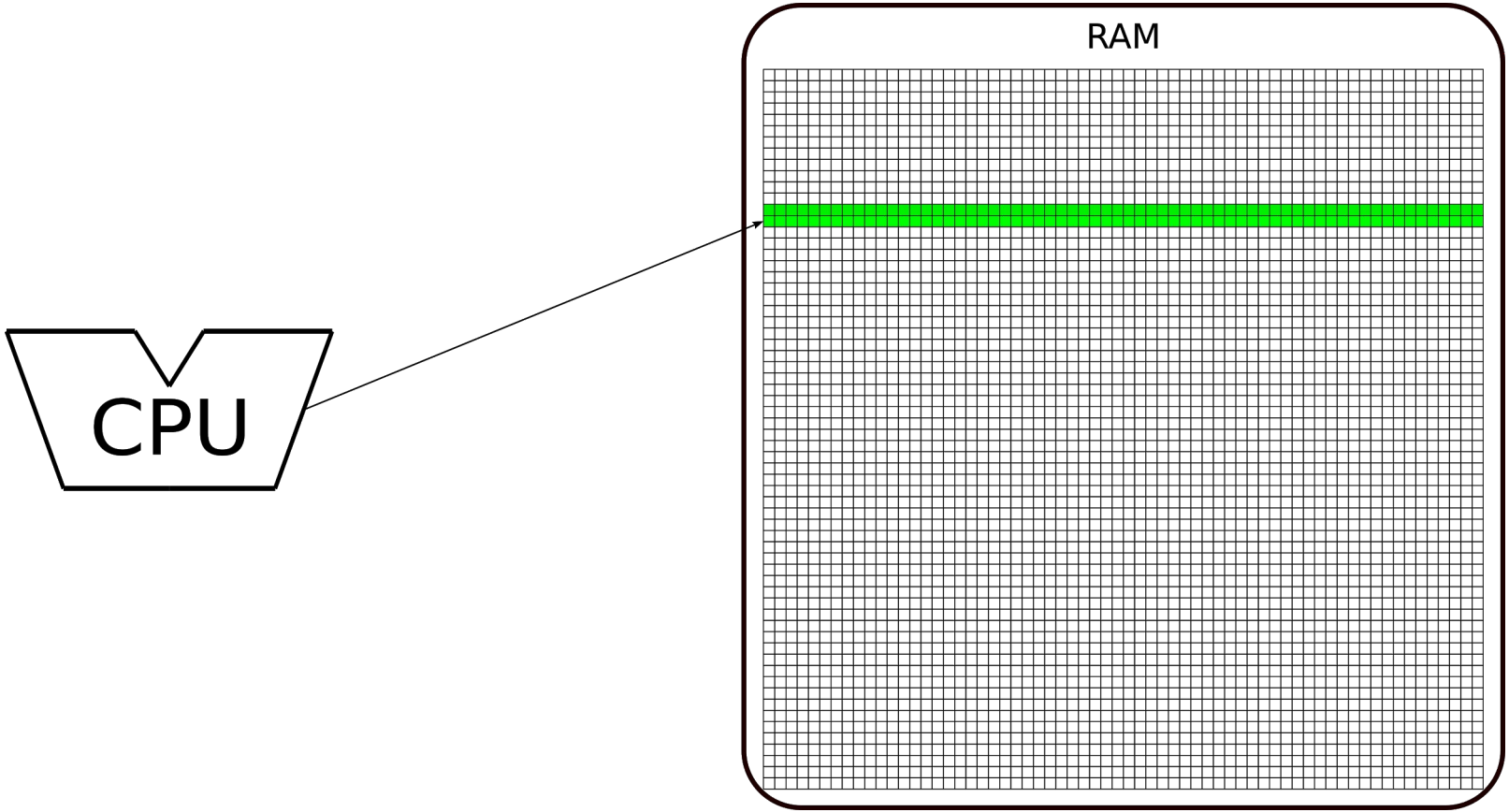
Prefetching

- Reduce the waiting period for missing data
- Easy access patterns are prefetched by the hardware
  - Linear access in both directions
  - Jumps of fixed width
- Complex patterns need manual prefetching (software prefetching)

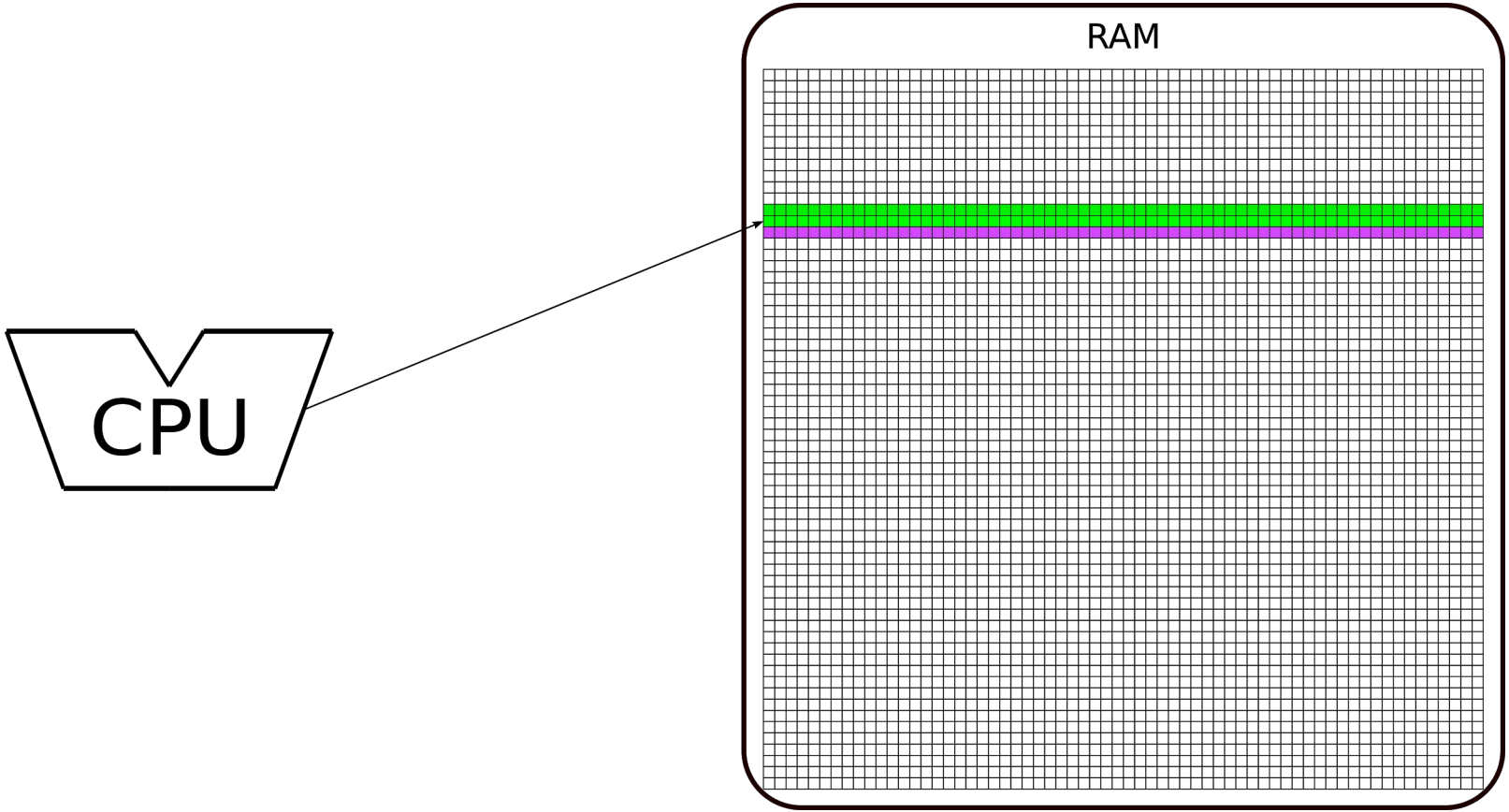
Hardware prefetching



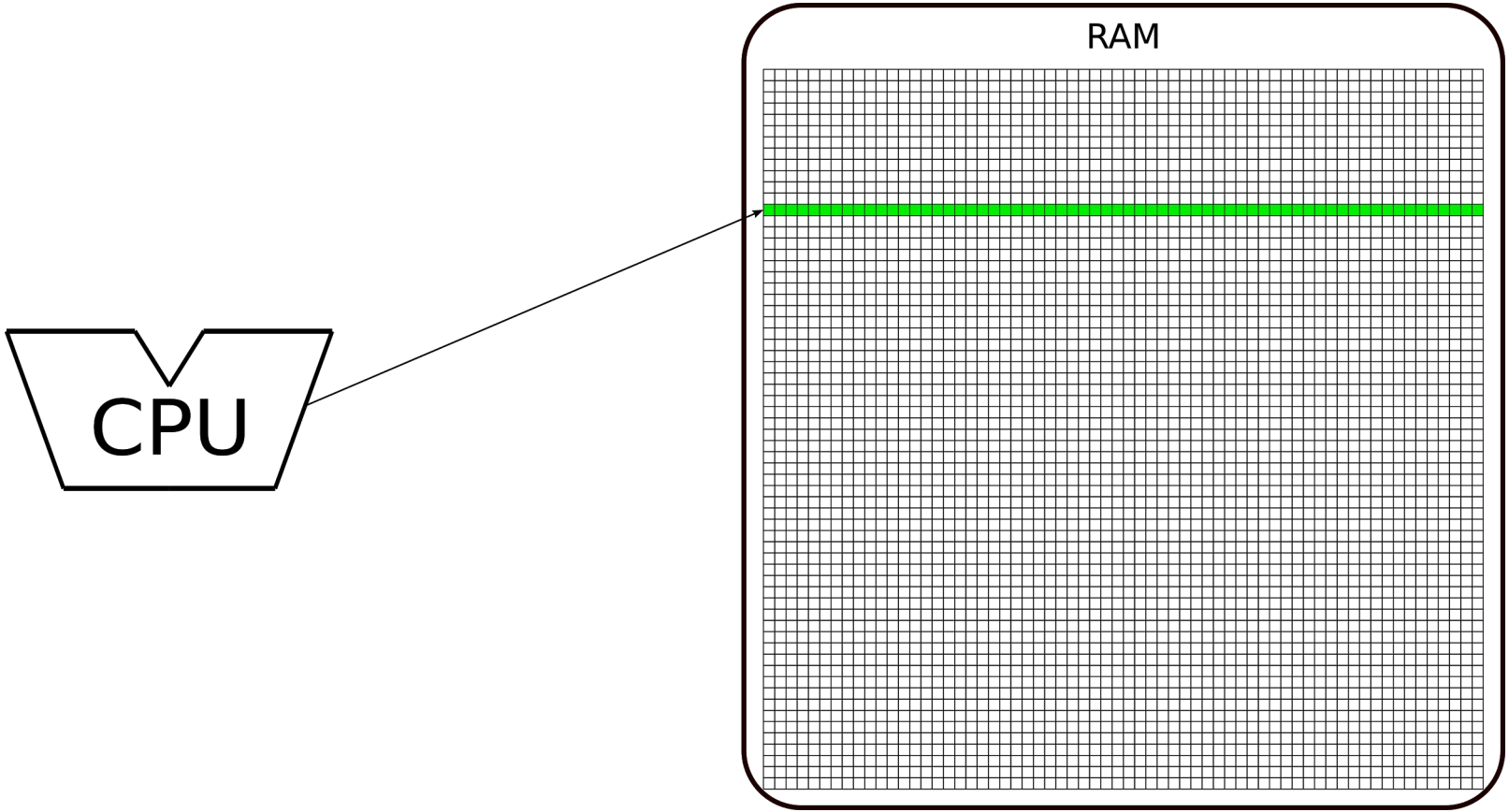
Hardware prefetching



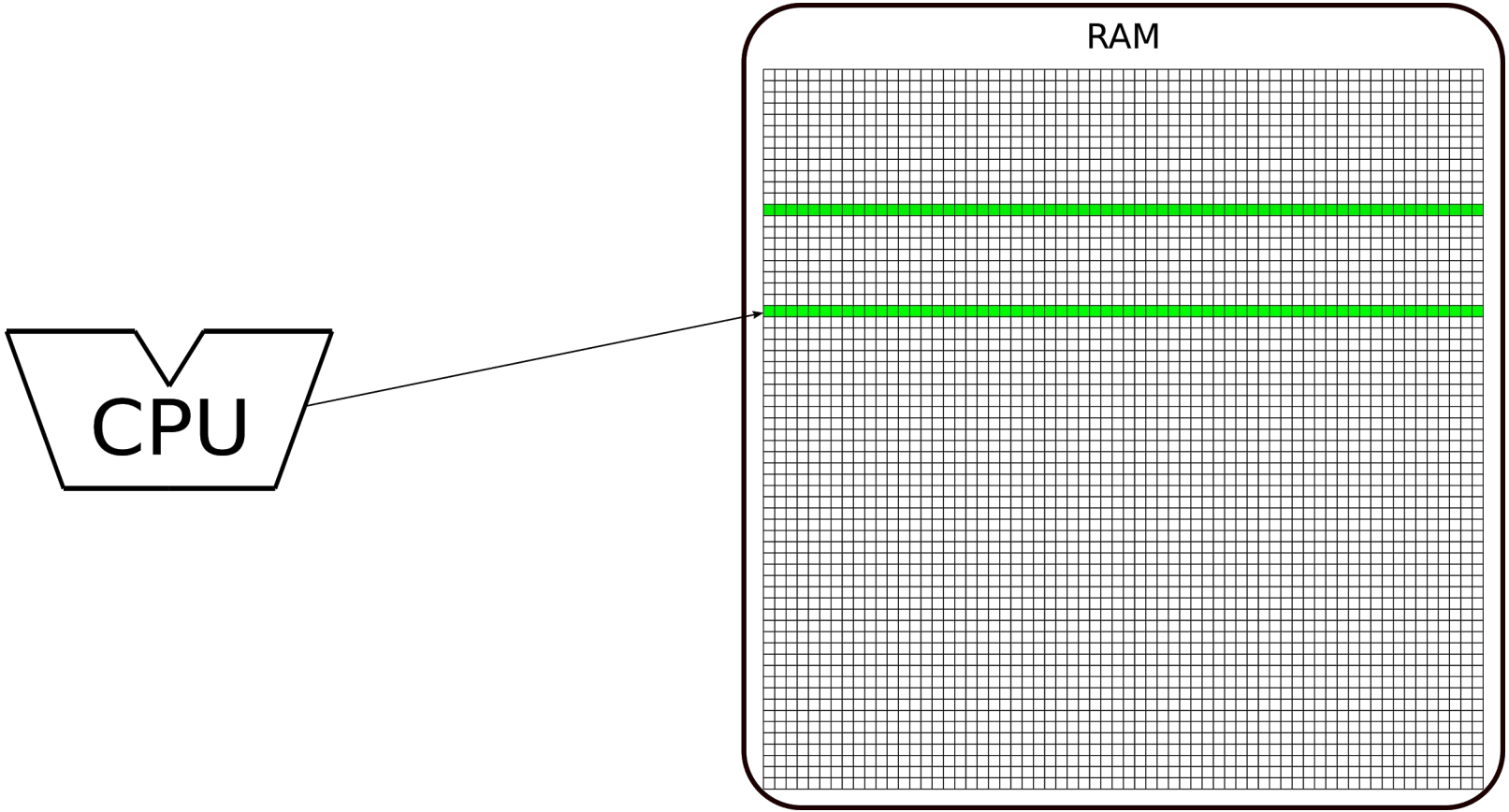
Hardware prefetching



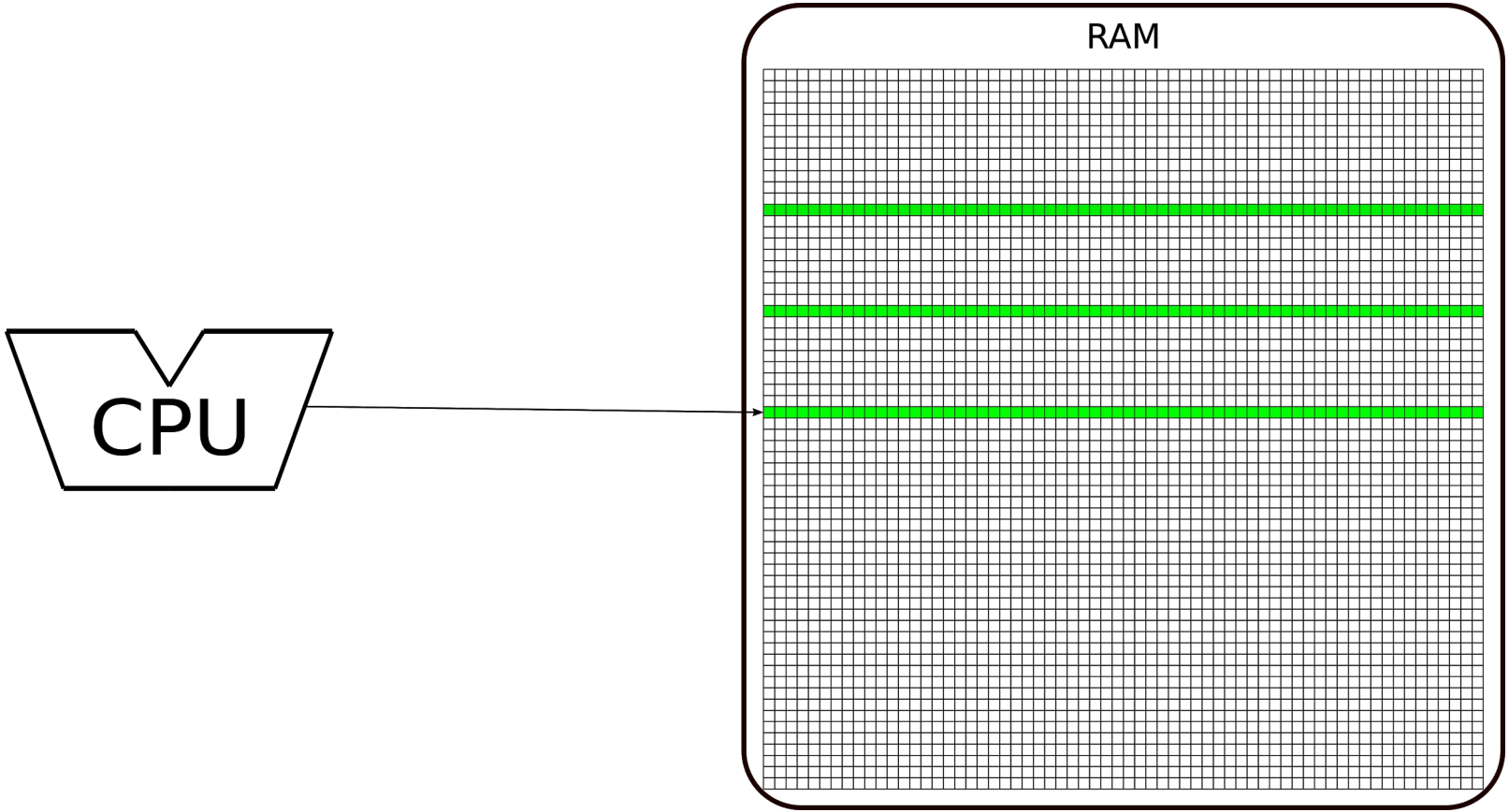
Hardware prefetching



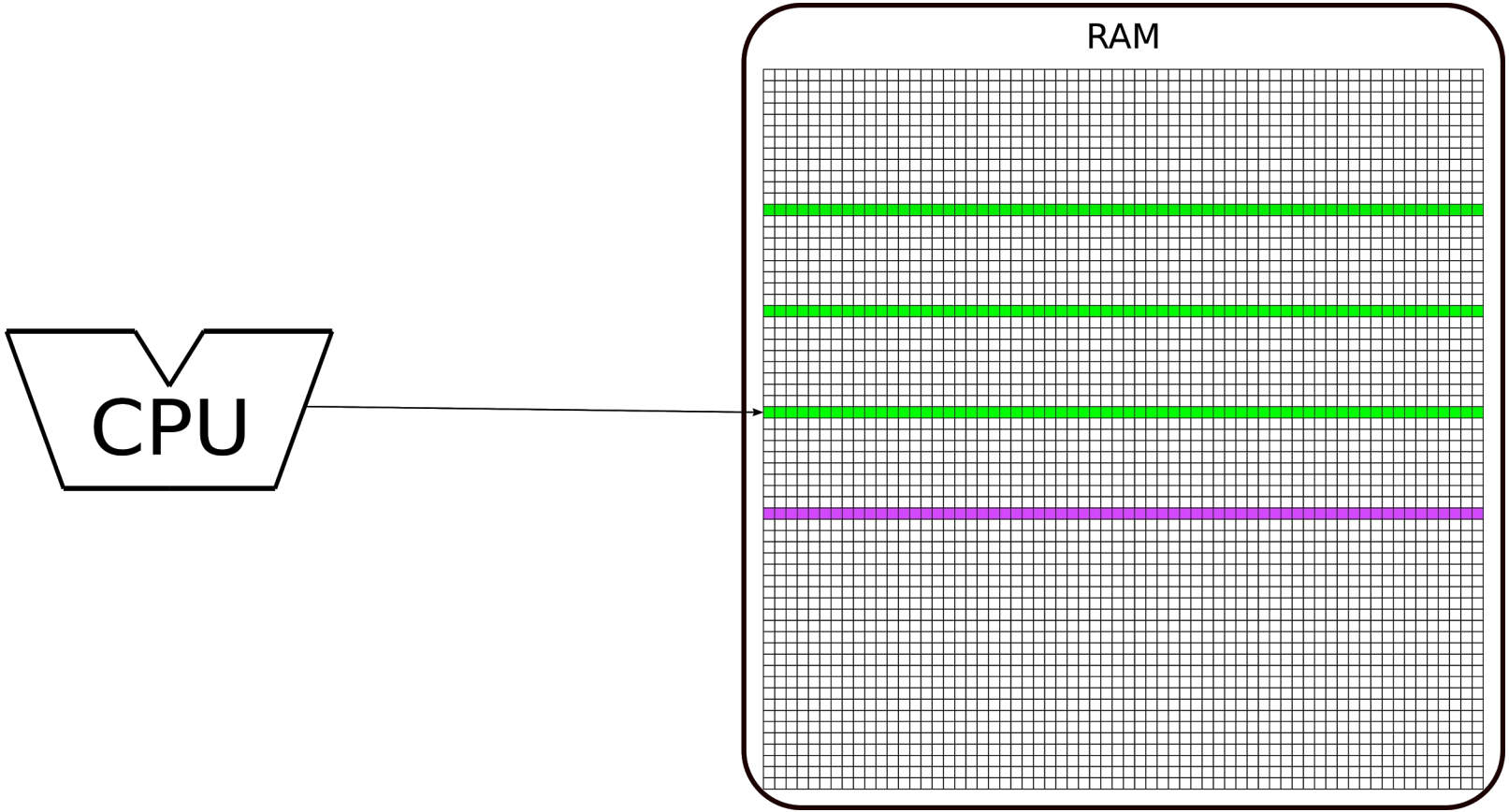
Hardware prefetching



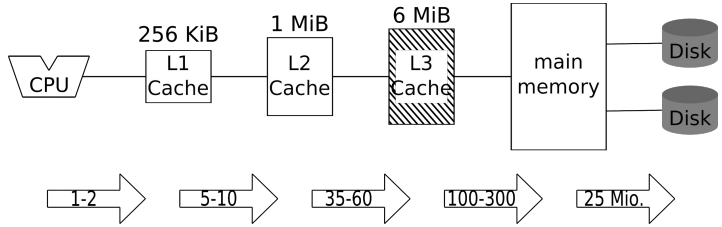
Hardware prefetching



Hardware prefetching



Efficiency of linear probing



- Cache hierarchy + hardware prefetching for regular access patterns means:
- Hopscotch:
  - check consecutive positions → cache-efficient
- Cuckoo hashing:
  - Searching within a bucket is cache-efficient
  - Looking up a bucket is not, but limited to h=3 buckets.
  - Also, there is software prefetching !

Software prefetching

- CPU instruction
- Can slow down the program
  - One instruction more to handle by the CPU
  - Still needed data can be removed from the cache
    - Needs to be loaded again

```
for(int i=0; i<1000; ++i) {
  __builtin_prefetch(&arr[i + k]);
  ++arr[i];
}
```

Prefetching in hash tables

- Strategies are based on multiple facts:
  - cache size
  - distribution of elements
  - size of needed data
- Hard to predict effects theoretically, needs benchmarking !

Parallelism

So far only serial algorithms, modern hardware is multi-core, so can we parallelize a) insertion b) lookup? b) is easy (read-only) but a) is difficult (data races)

Synchronization needed (locks, spin locks ?)

CAS instruction (Jellyfish)

SIMD instructions (single instructions multiple data)

Parallelism

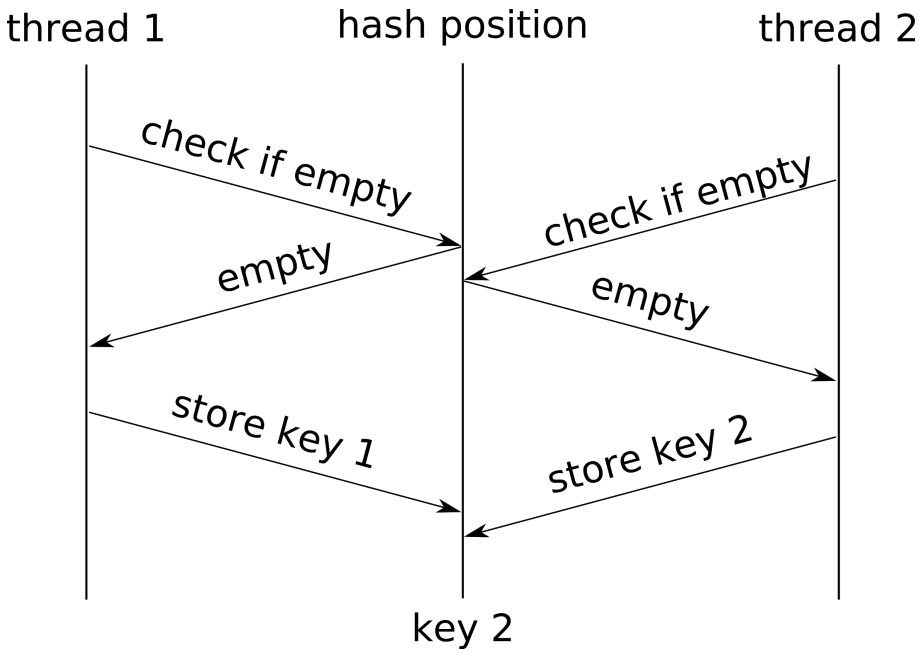
- So far only serial algorithms
- Modern hardware is multi-core
- Also SIMD: single instruction multiple data
  (e.g. compute hash functions on multiple *k*-mers in parallel)

- Parallel lookup is easy:
  - data does not change
  - only read access

- Parallel write is harder:
  - Ensure that the data is always consistent
  - Multiple threads write to the same memory location:
    Synchronisation needed

Access without synchronisation

- Both threads check whether the hash position is empty or not.
- Both see that the location is empty.
- Thread one stores key 1.
- Thread 2 stores key 2 and overwrites key 1.
- Key 1 is lost.

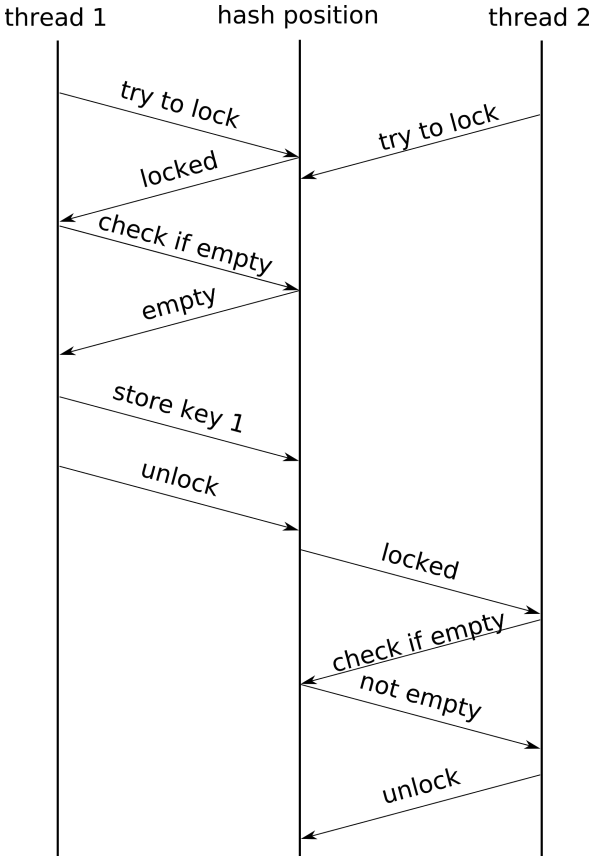


Access with synchronisation

- Try to lock table slot
- As soon as the lock is confirmed:
  - change value in slot
- If slot is locked:
  - Wait until the lock can be obtained

Large memory overhead if explicit locks are used for every single slot.

(Don't do this!)



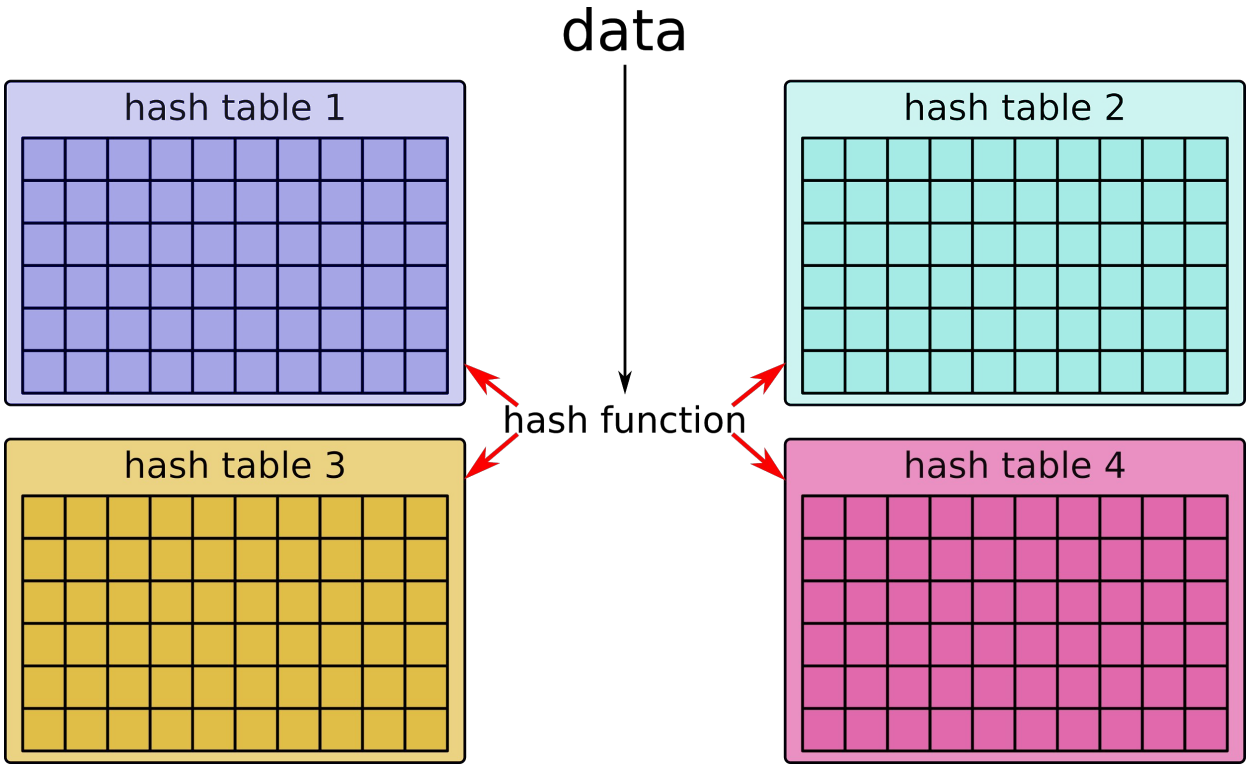
Atomic compare-and-swap (CAS) instruction

- Can be used to implement **lock-free** algorithms

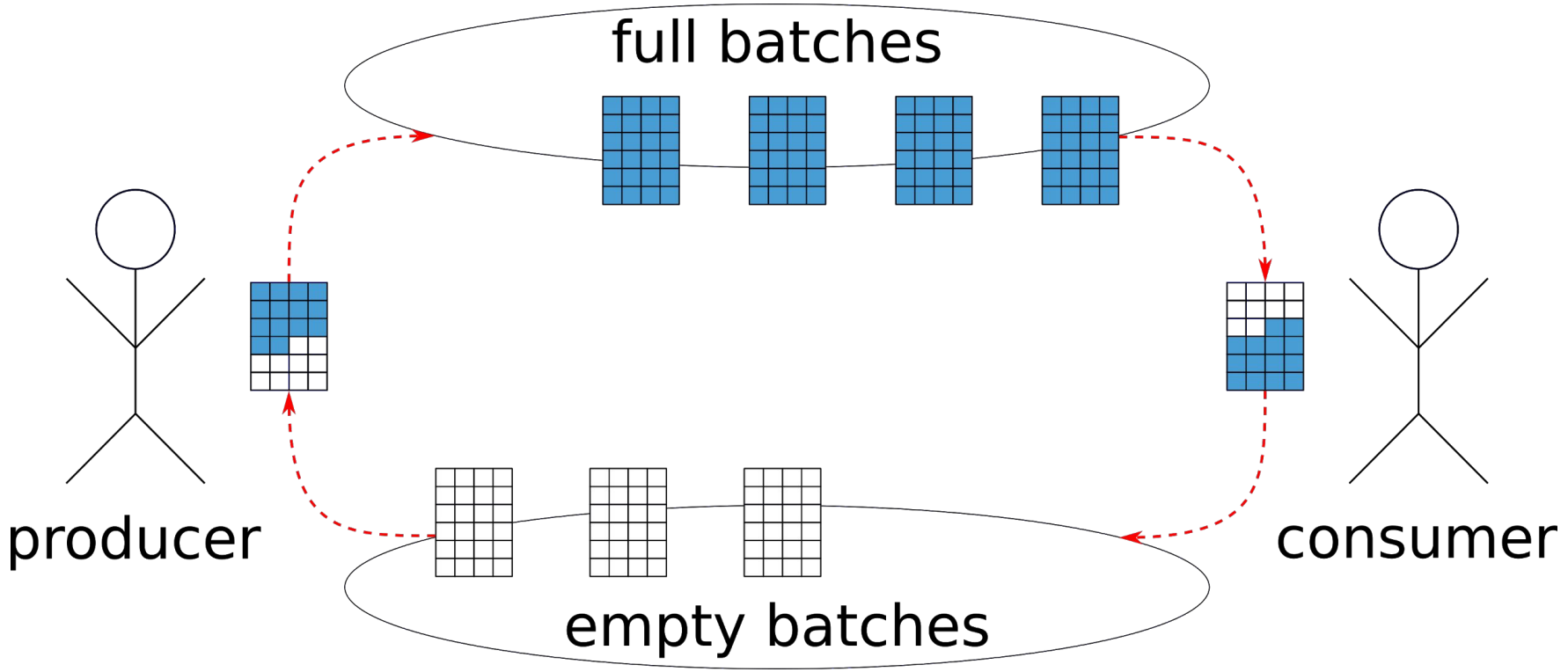
- **Compare** content of memory location with an expected value
- If the content equals the expected value:
  - Store the new value
  - Return the old value
- One atomic CPU instruction
  - Cannot be interrupted by another thread

- Positions in a hash table are initialized with 0
- Try to store a new key
- Expected value is always 0
- CAS only stores the new key if the slot was empty

Alternative: Partition hash table into sub-tables



Producer-consumer model on partitioned table



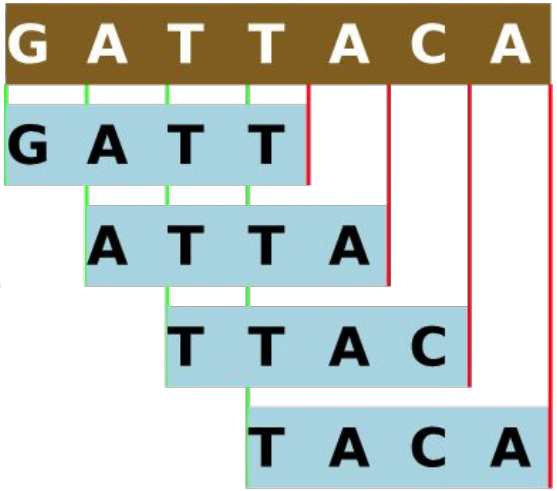
Summary

Summary and Recommendations

- Use (2, *b*) Cuckoo hashing with *b* = 4 to 8
- Use (3, *b*) Cuckoo hashing with *b* = 2 to 8
- Ensure each bucket is contained in a cache line (no add'l cache misses!)
- Perhaps use (*h,b*)-Cuckoo hashing with overlapping buckets
  (Espinoza: "Cuckoo breeding ground", unpublished)
- All of the above offer high loads and fast access
  (close to 1 lookup per element is possible).

Xengsort: Alignment-free xenograft sorting

- Partition each read into its *k*-mers ($k \geq 23$)
- Look up information on each present *k*-mer
  k-mer ↦ human | mouse | both
  k-mer ↦ human | human? | mouse | mouse? | both
  "weak" *k*-mers: Hamming-1 neighbor in other genome
- Aggregate *k*-mer information into a statement about the entire read, e.g., by majority vote



Xengsort

Hashing strategy:

- *(h,b)*-Cuckoo hashing with *h=3* and *b=4*
  - Insertion strategy: Random walk
- Value: 3 bit for host, graft and weak
- achieves up to 99% load

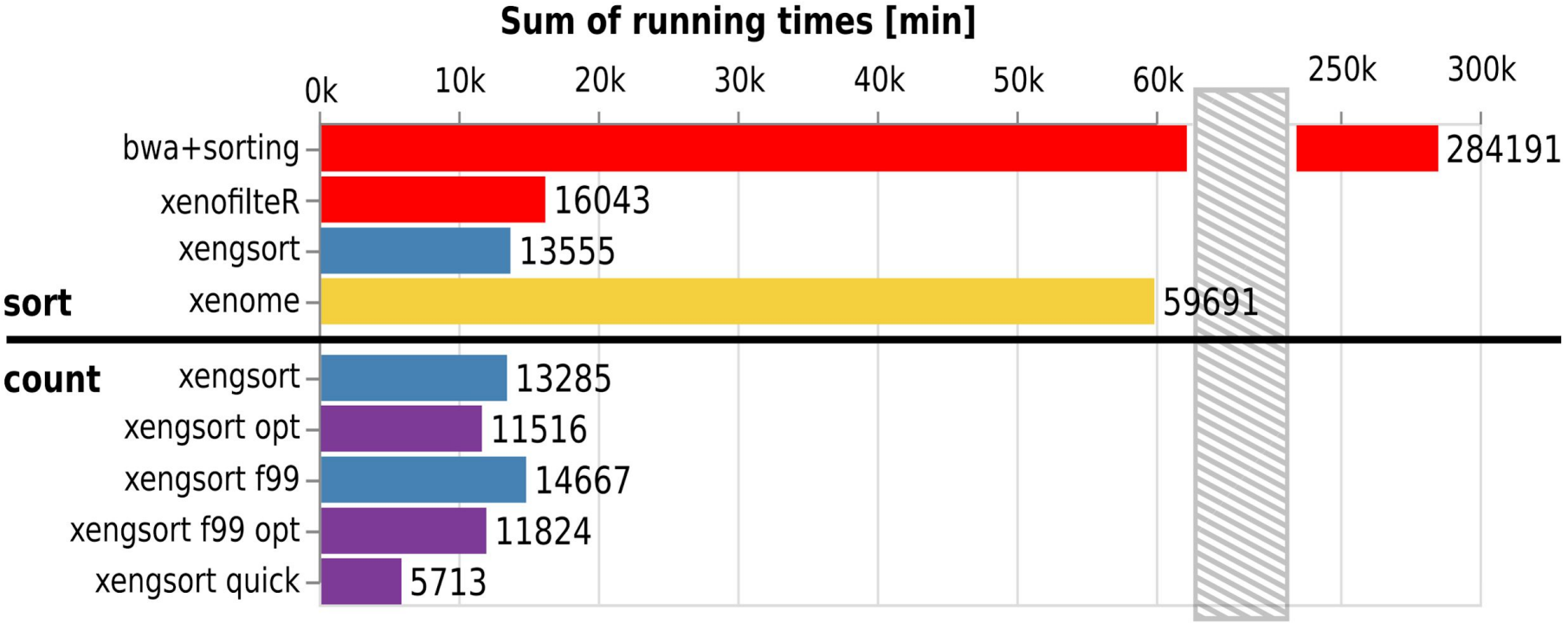
Page layout:

- page = [$slot_1$, …, $slot_b$]
- slot = [fingerprint, choice, value]

Classification:

- Use multiple threads (only read access)
- divide a chunk of reads to each thread
- query *k*-mers of each read
- use decision tree to classify reads

Run time


Sum of running times [min]
0k
10k
20k
30k
40k
50k
60k
250k
300k
bwa+sorting
284191
xenofilteR
16043
xengsort
13555
sort
xenome
59691
count
xengsort
13285
xengsort opt
11516
xengsort f99
14667
xengsort f99 opt
11824
xengsort quick
5713

Trade-offs

- Larger page sizes (up to one Cacheline)
  - Less pages → More bits for the fingerprints
- More hash functions:
  - more even distribution of the elements on the pages
  - more positions to check → more cache misses