Modern hashing
for alignment-free sequence analysis

Part 4:
Performance Engineering

Jens Zentgraf & Sven Rahmann

GCB 2021

Overview

Saving space

- optimizing the bit-level layout of the hash table
- compact encoding of hash choices and values
- quotienting

Saving time

- optimization of hash choices (store many keys at their first choice)
- shortcuts for unsuccessful lookups
- prefetching
- parallelization

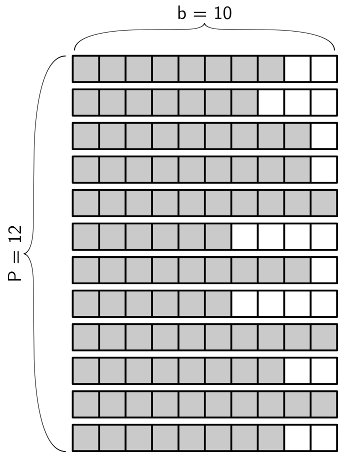
PhD / Postdoc position available

at the "Algorithmic Bioinformatics" group, Saarbrücken

- algorithm engineering applied to bioinformatics:
  e.g., tricks like presented here
- novel methods for new problems
- desired: algorithm & data structures skills
- desired: programming experience, software development,
  or: strong theoretical background
- Application areas: pangenomics, cancer, metagenomics
- full position (100%), also for PhD students, 3 years
- Contact Sven (rahmann@cs.uni-saarland.de)

Saving Space

Bit-level layout of a hash table bucket



Several options to store representation for each (key, value)

- DNA *k*-mer key needs *K*=2*k* bits; value needs *v* bits
- Assume $K \leq 64$, $v \leq 64$, cache line = 512 bits

- [key1 (64) | value1 (64) | key2 (64) | value2 (64) | …]
  (4 pairs will exactly fit in a cache line; may use padding otherwise)
- [key1 (*K*) | value1 (*v*) | key2 (*K*) | value2 (*v*) | …]
  (more pairs fit into a cache line, need bit operations to extract)
- [key1 (*K*) | key2 (*K*) | … | aggregated-values ($\leq bv$)]
  (saves space if number of possible values is not a power of 2;
  for 5 values, *b*=3: 5^3=125 (7 bits) instead of 3 ceil($\log_2$(5)) bits = 9 bits)

Saving Space with Quotienting: Example

Keys: canonical codes of 25-mers (50 bits)
Values: species (5 classes: 3 bits)

4.5 billion k-mers: reference genomes, alternative alleles, cDNA transcripts:
53 bits per entry, load 0.88: **33.88 GB** for hash table 😩

Quotienting to the rescue:

- Do not store full keys (k-mers), but only "quotients" (here 20 bits), plus hash function choice (2 bits) plus values (3 bits) → 25 bits per entry:

15.98 GB for hash table 😃
(could be slightly reduced by higher load, value compression, etc.)

Quotienting: Details

Keys are encoded canonical *k*-mers (half of set $[4^k] := \{0, .., 4^k\text{-}1\}$).

Step 1: Bijective randomizing function $[4^k] \to [4^k]$ with *a* odd

$$g_{a,b}(x) := [a \cdot (\mathrm{rot}_k(x) \text{ xor } b)] \bmod 4^k$$

Step 2: Map to buckets (simply mod *p*: number of buckets). Define

$$f(x) := g_{a,b}(x) \bmod p \quad \text{and} \quad q(x) := g_{a,b}(x) \;//\; p\ .$$

Then *x* can be uniquely reconstructed
from $f(x)$ ("hash value, "bucket number") and $q(x)$ ("fingerprint", "quotient").
Sufficient to store $q(x)$ in bucket $f(x)$ (and which hash function was chosen).

Bit-level layout with quotients and hash choices

- [key1 (K) | value1 (v) | key2 (K) | value2 (v) | …]

⬇️

- [quotient1 (Q) | choice1(2) | value1 (v) | quotient2 (Q) | choice2 (2) | value2 (v) | …]
  = [signature1 (Q+2) | value1 (v) | signature2 (Q+2) | value2 (v) | …]

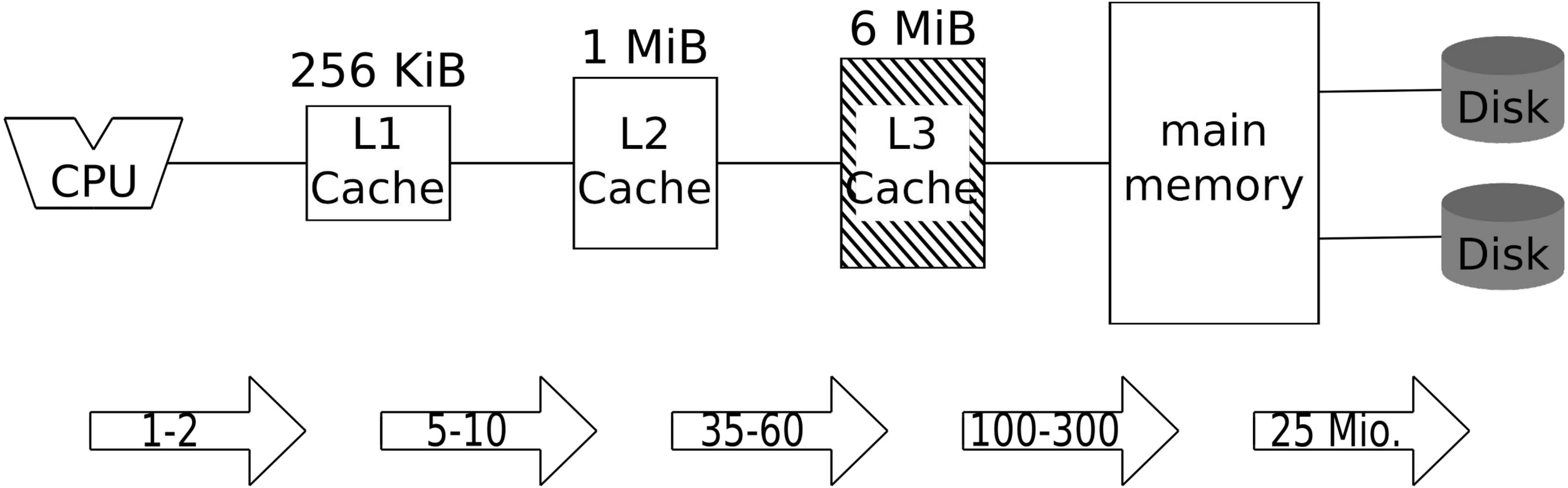
Save more bits by **sorting slots by choice**, and only storing choice counts.
Can be combined with compact value storage:

- [choices ($\leq 2b$) | quotient1 (Q) | quotient2 (Q) | … | values ($\leq bv$)]
  (requires decoding of the "choices" integer into actual numbers)

To pad or not to pad?

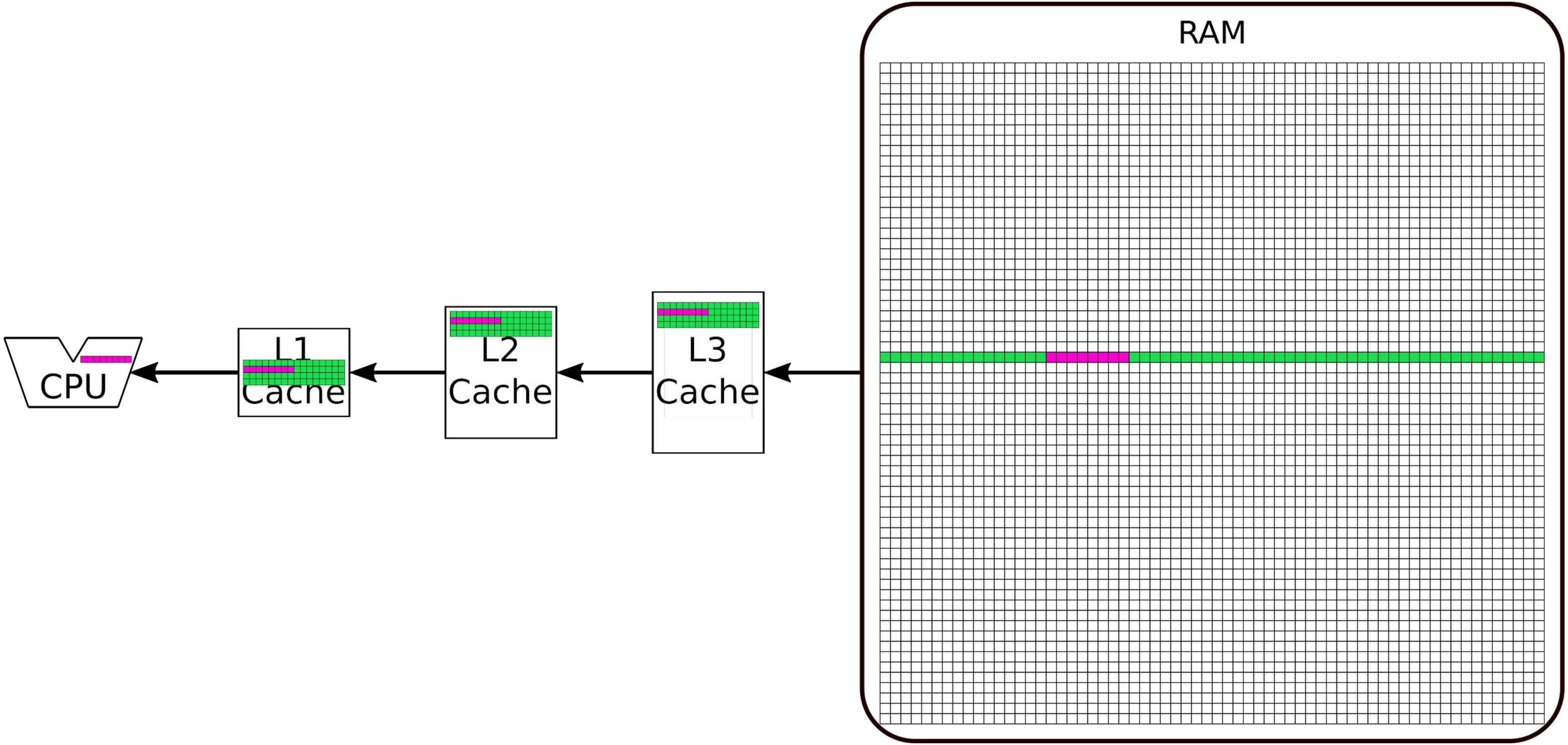
Main decision: pad incomplete 512-bit cache lines or not?

No: some buckets may extend across two cache lines.



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

Cache line



Saving Time

Optimization of the hash function choices

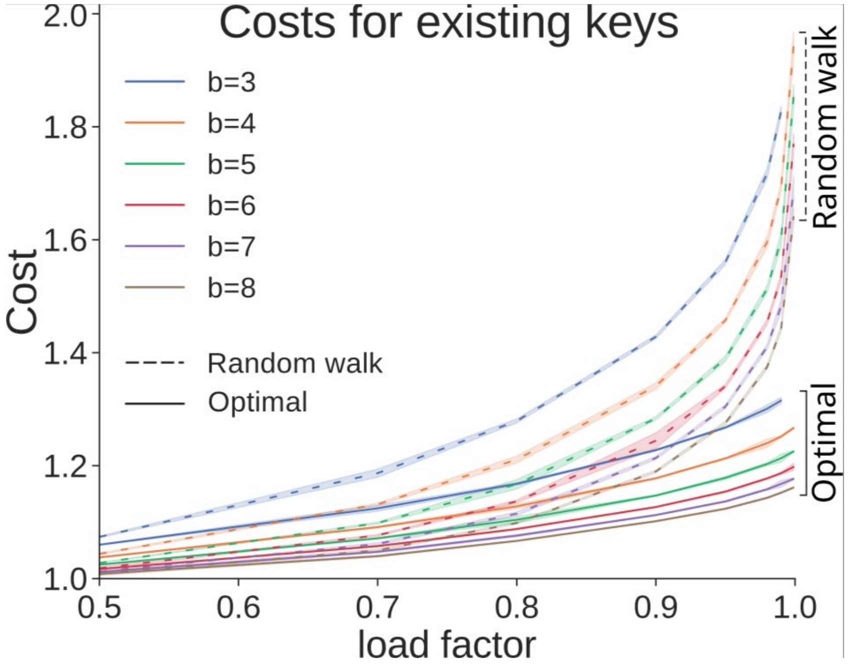
- Idea: Place many *k*-mers into the bucket of their **first** hash function.
- Can be written as a **minimum weighted bipartite matching problem**:

4.5 billions of keys ↔ 100s of millions of buckets

(3 buckets for each key; cost 1, 2, 3)

- Solvable exactly within a few hours up to a few days of CPU time.
- Can save up to 10 - 15% of running time in a real application (xengsort) in comparison to hash tables created by "random walk".

Optimization of the hash function choices



Look-up costs (#cache misses) for different hash table designs:

- bucketed Cuckoo hashing;
- different bucket sizes,
- different load factors,
- two insertion strategies.

Speeding up unsuccessful searches

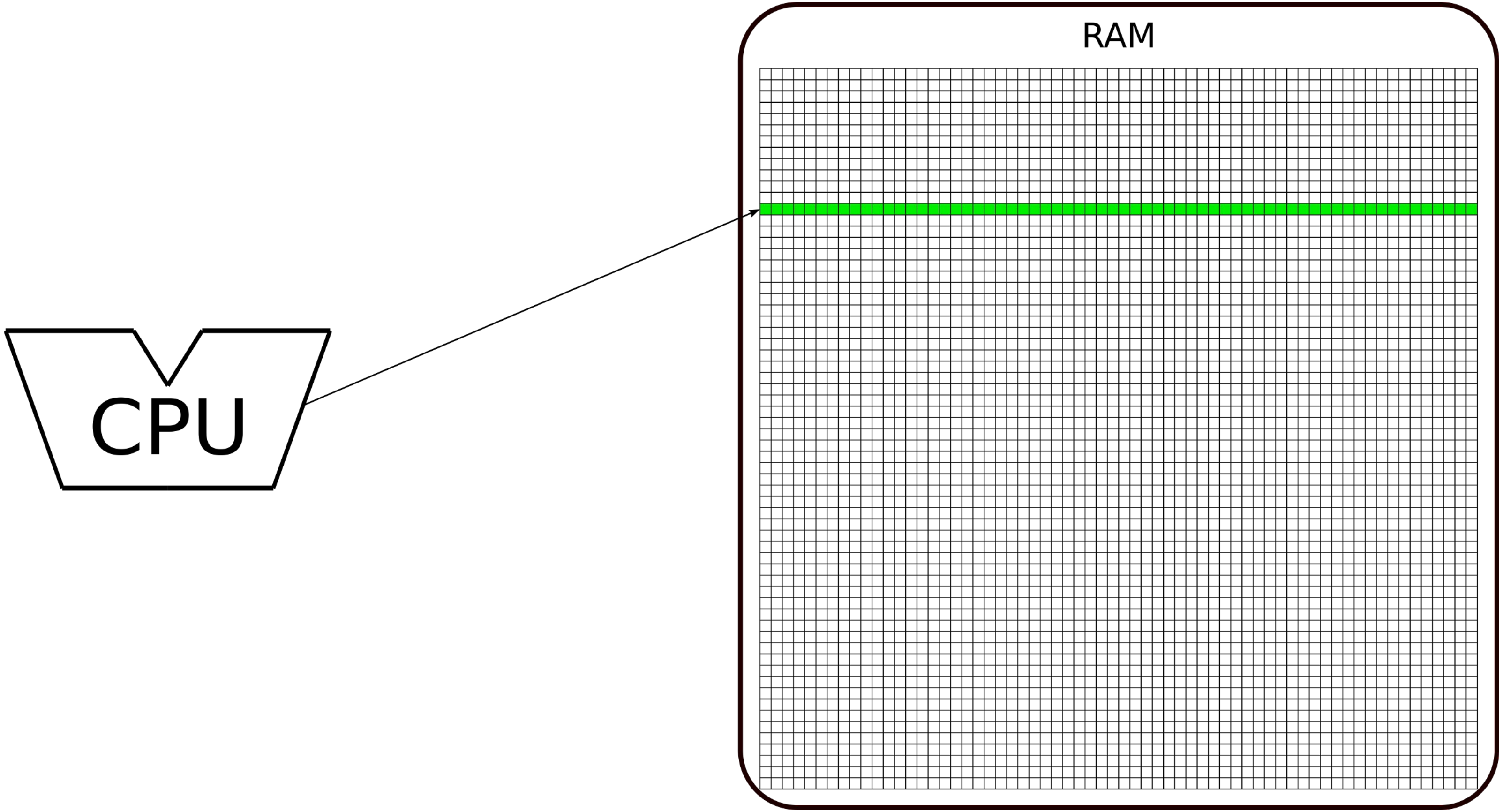
- **Bad:** unsuccessful key searches always incur h=3 cache misses.
- … unless we learn from the first bucket that a search on the second / third bucket will not be successful.
- **Idea:** Reserve bits for each bucket to store information of the following type: "there is at least one key that would be stored here with its 1st (2nd) choice, but is stored at its 2nd (3rd) choice."
- Different combinations or resolutions are possible: 3 bits / 2 bits / 1 bit.

- Good speed-up for unsuccessful searches, little additional space cost.
- Additional set-up time for computing all the bits after inserting all elements.
- Insertions/deletions of keys invalidate the computed bits.

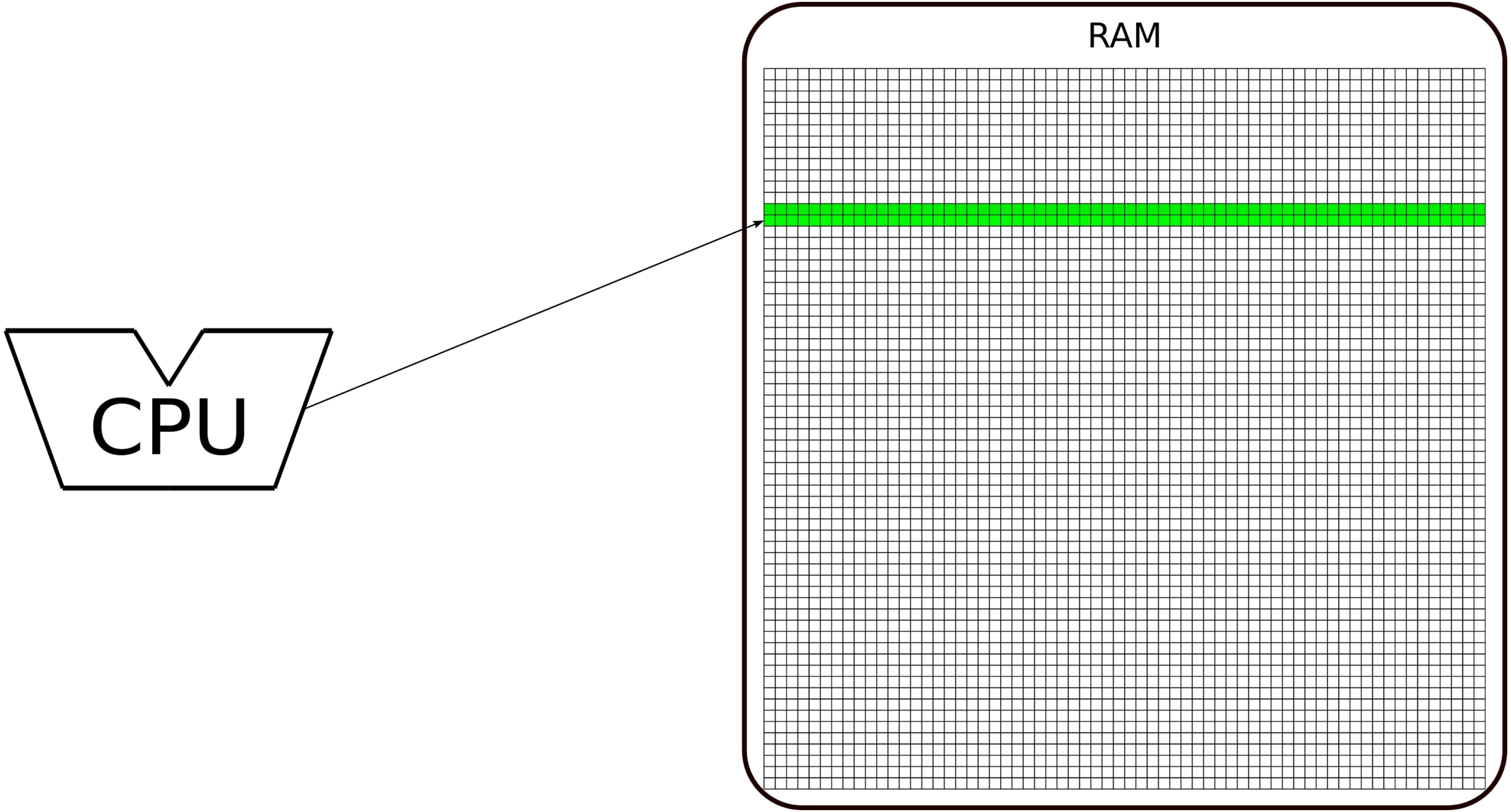
Prefetching

- Reading random access data rom RAM is slow (200 - 300 CPU cycles).
- **Idea:** Reduce the waiting period for data stored somewhere in RAM.
- Easy access patterns are prefetched by the **hardware**
  - Linear consecutive access in both directions ([reverse] streaming)
  - Regular jumps of fixed width
- Complex patterns need manual prefetching (**software** prefetching)

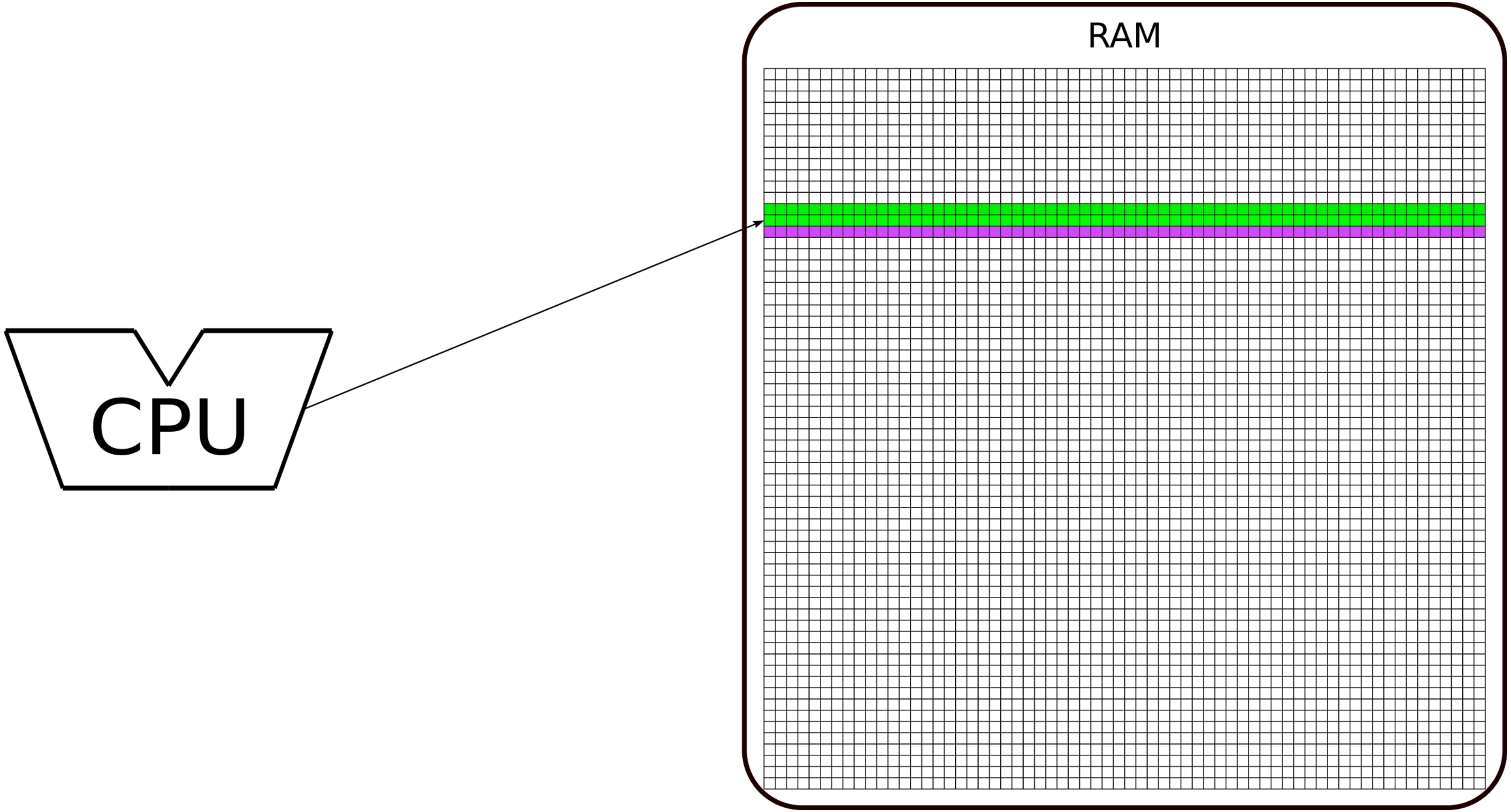
Hardware prefetching



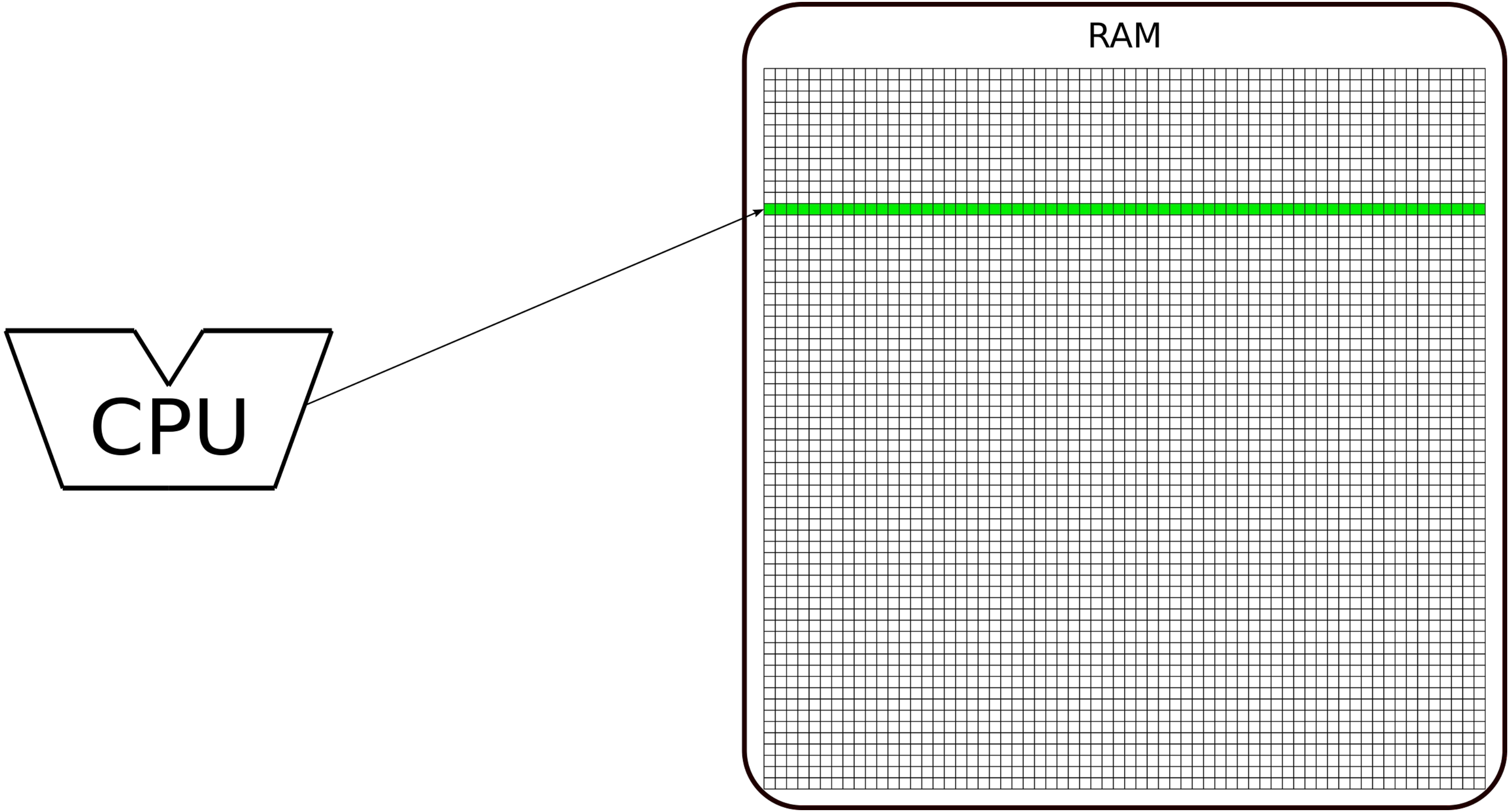
Hardware prefetching



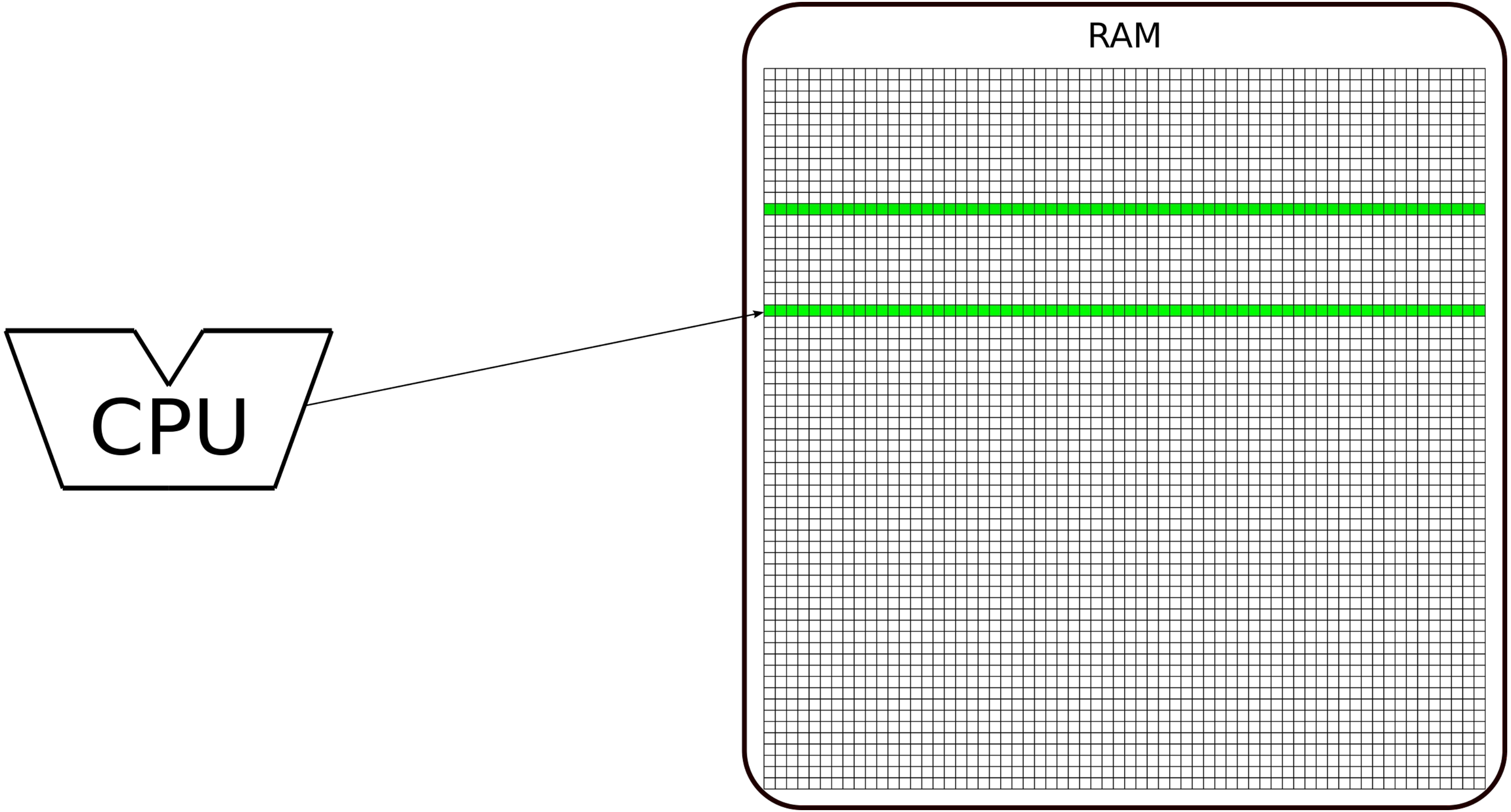
Hardware prefetching



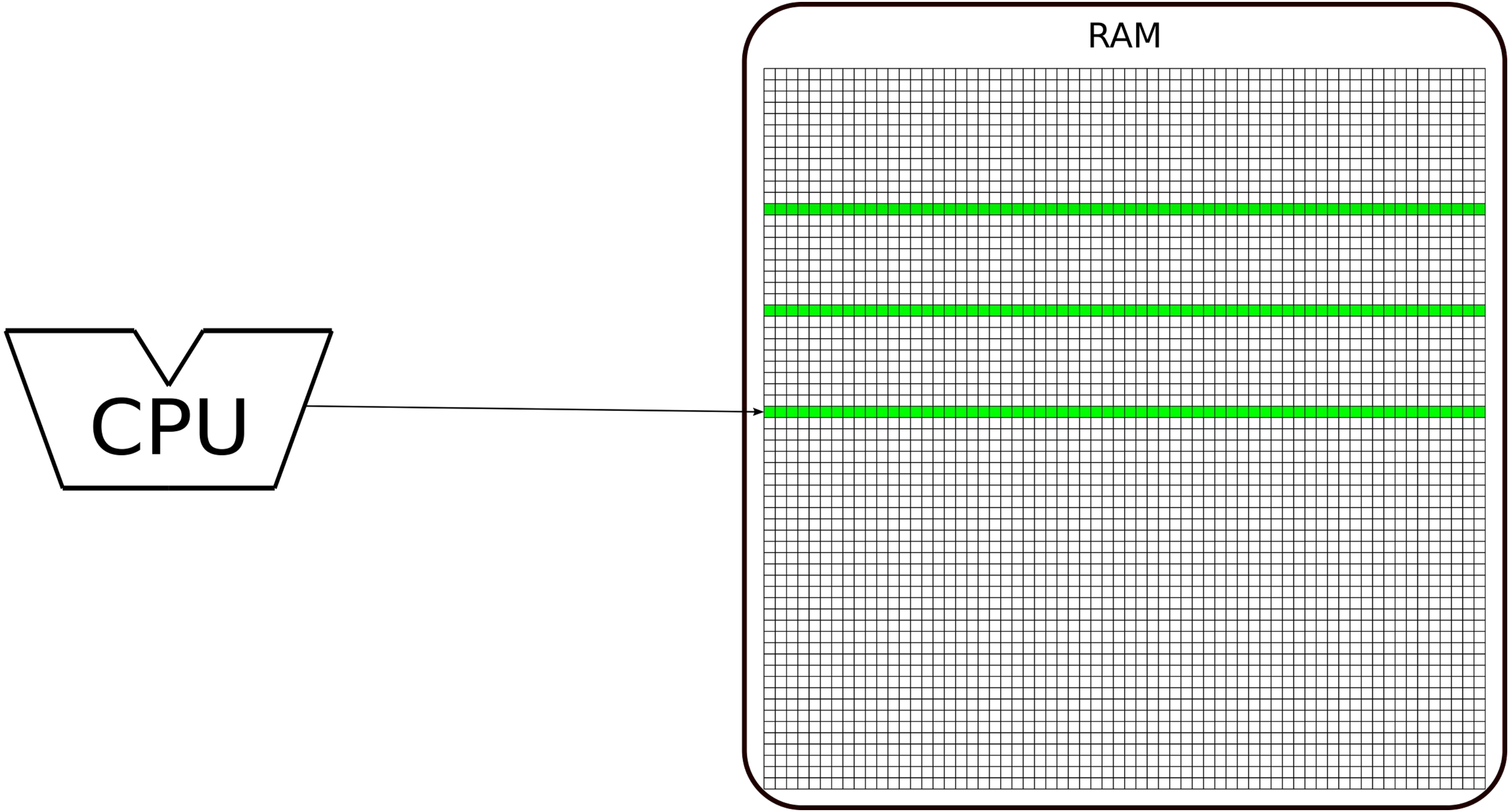
Hardware prefetching



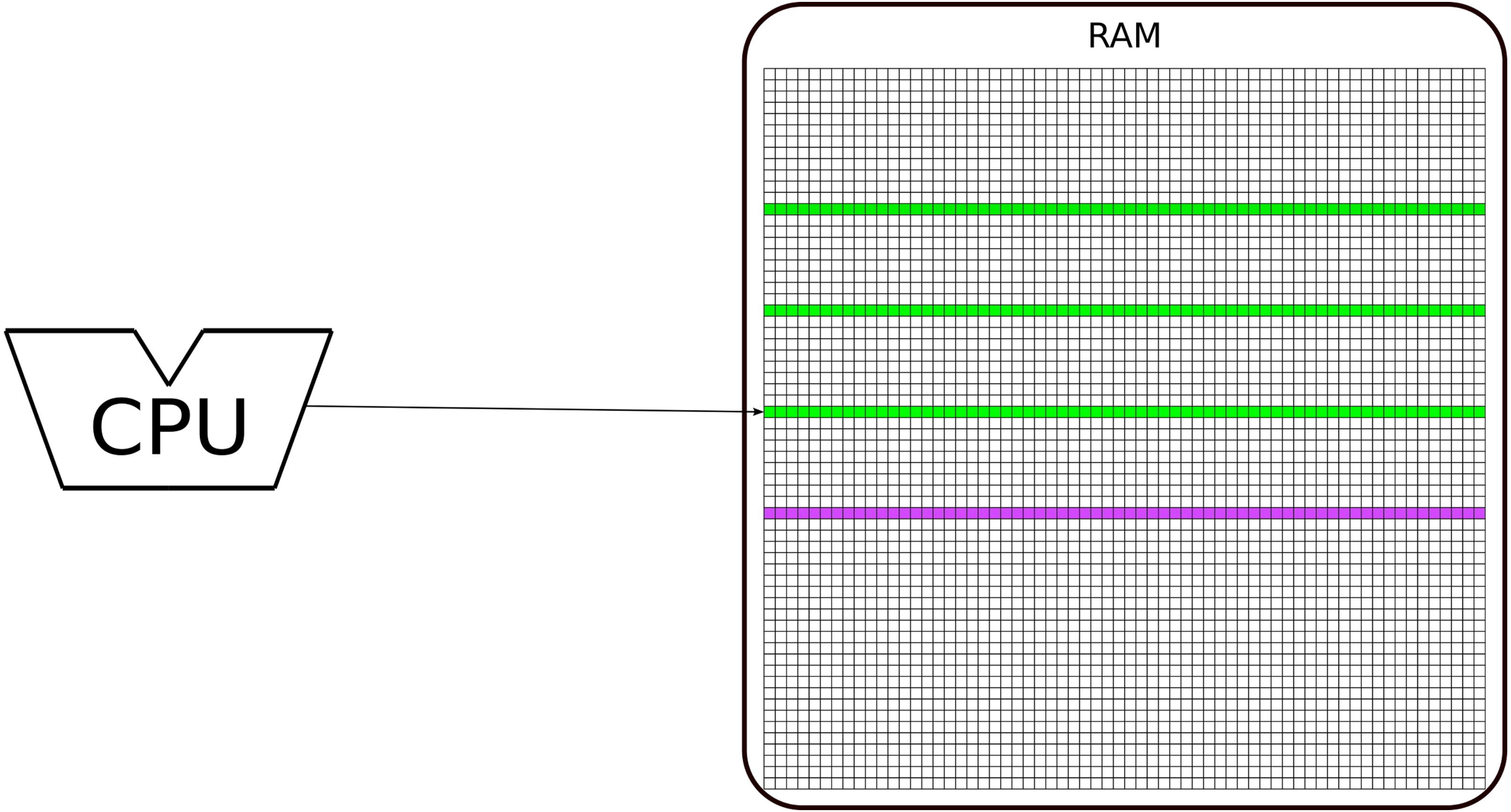
Hardware prefetching



Hardware prefetching



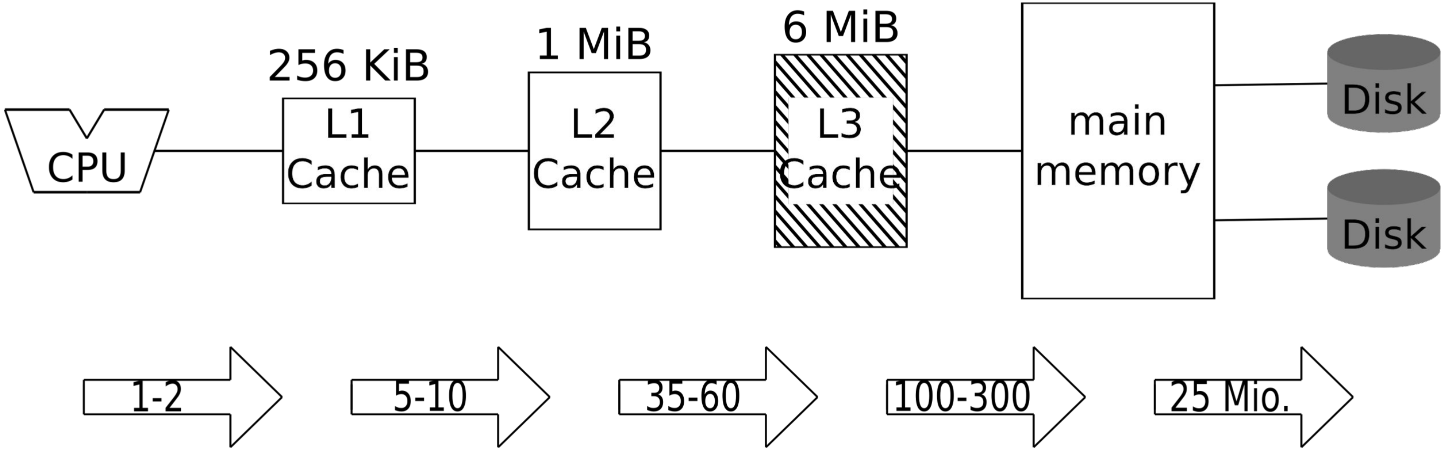
Hardware prefetching



Cache friendliness

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 be helpful if used at the right moments
- Can slow down the program
  - One instruction more to handle by the CPU
  - Still needed data can be removed from the cache

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

Software prefetching in Cuckoo hash tables

Possible strategies

1. Never prefetch
2. Before examining a key's first bucket, prefetch the second bucket. Before examining the second bucket, prefetch the third bucket.
3. Before examining a key's first bucket, prefetch all other buckets.
4. When examining *n* keys in a row, during processing key *i*, prefetch the first bucket of key *i*+*k*, for some offset *k*.

Any of them may be fastest. Needs benchmarking.
Look-ahead (4.) complicates the implementation.
We first recommend comparing 1. with 2.

Parallelism

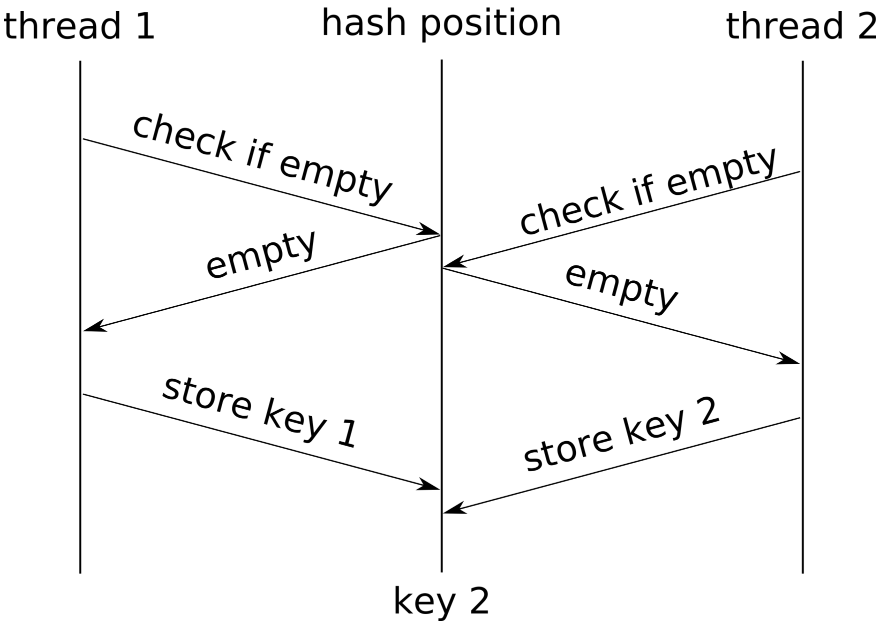
- So far only serial algorithms, but modern hardware is multi-core
- Also SIMD: single instruction multiple data
  (e.g. compute hash functions on multiple *k*-mers in parallel)

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

- Parallel write is harder:
  - Ensure that the data is always consistent
  - Multiple threads write to the same memory location: Synchronisation needed
  - Perhaps avoid the possibility of conflicting writes ?

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.

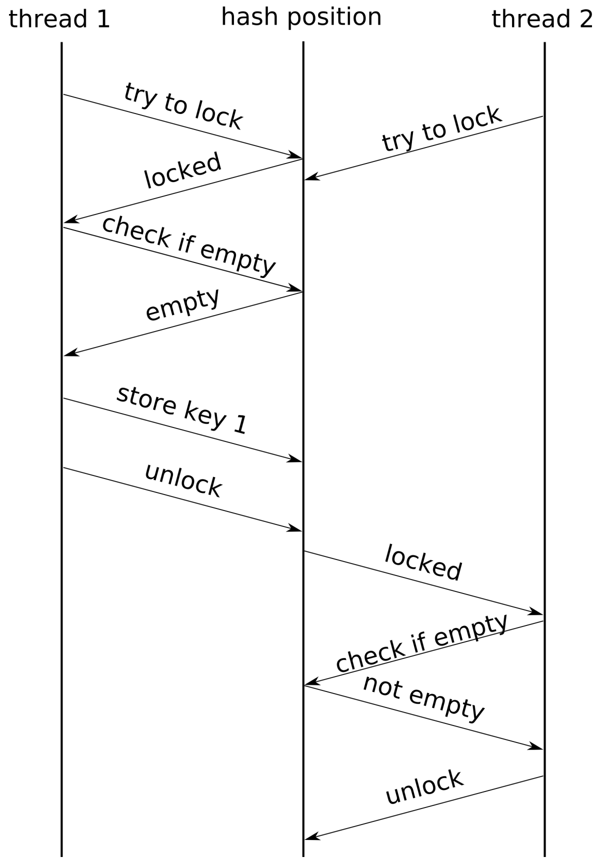


Access with synchronisation

- 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!)



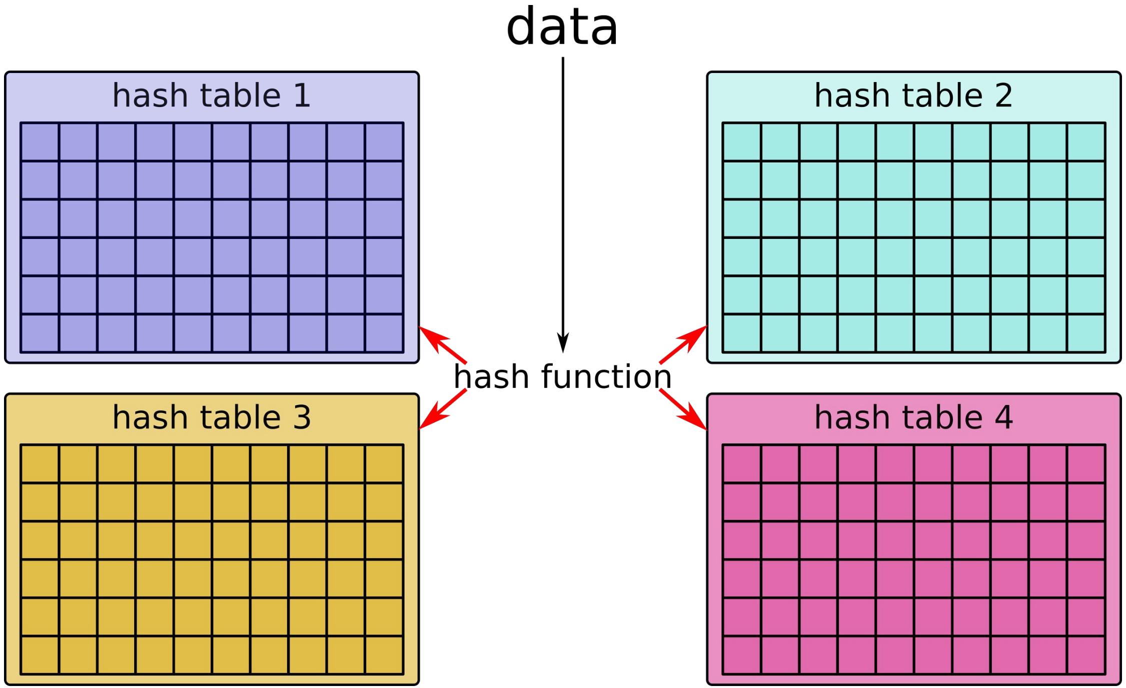
Atomic compare-and-swap (CAS) instruction

- 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
- Otherwise do nothing.
- One atomic CPU instruction, cannot be interrupted by another thread

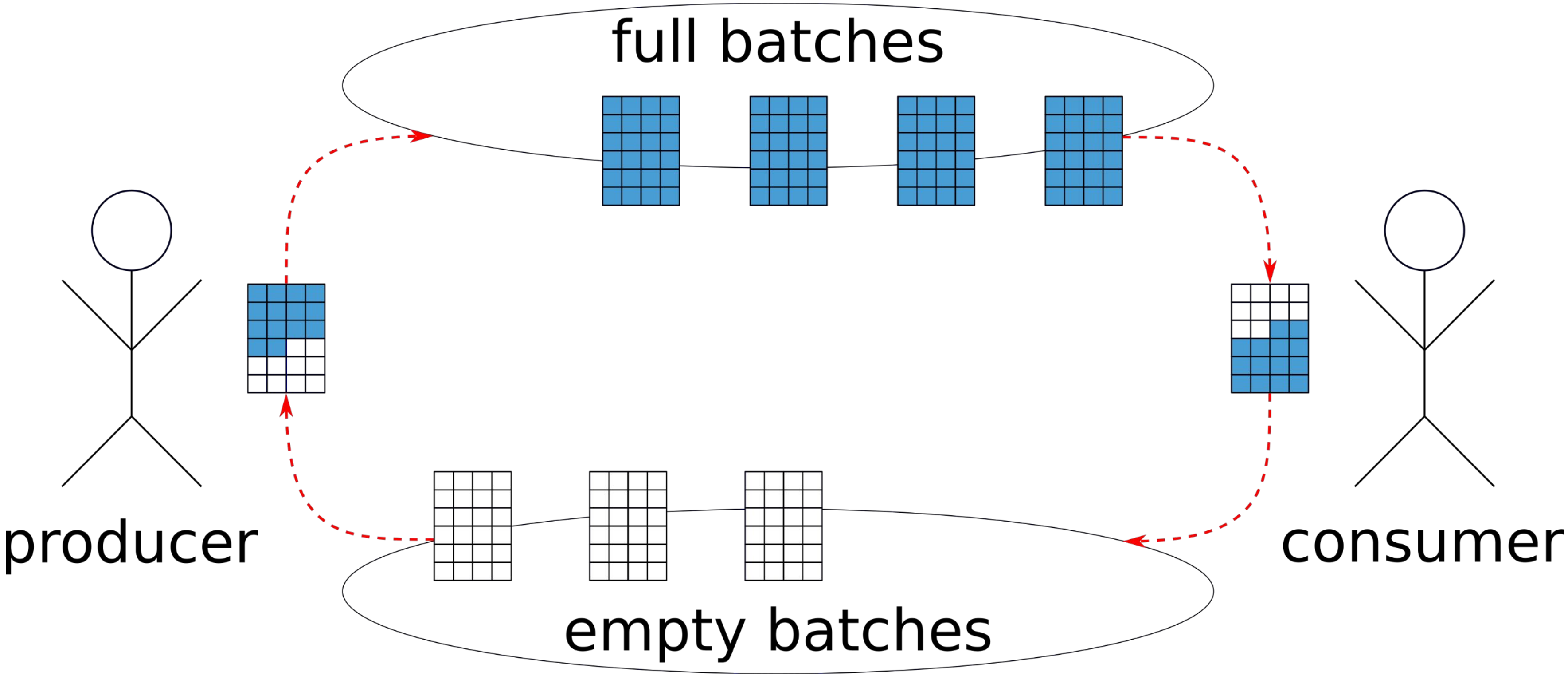
- Positions in a hash table are initialized with 0
- Try to store a new key, expected old value is always 0
- Only store the new key if the slot was empty
- Otherwise find a new location.

Alternative: Partition hash table into sub-tables

- one thread responsible for each sub-table
- design hash functions to be consistent within a table



Producer-consumer model on partitioned table



Summary: Performance Engineering

Saving space

- optimizing the bit-level layout of the hash table
- quotienting
- compact encoding of hash choices and values

Saving time

- optimization of hash choices (store many keys at their first choice)
- shortcuts for unsuccessful lookups
- prefetching
- parallelization