Concurrent Hash Tables: Fast and General!(?)

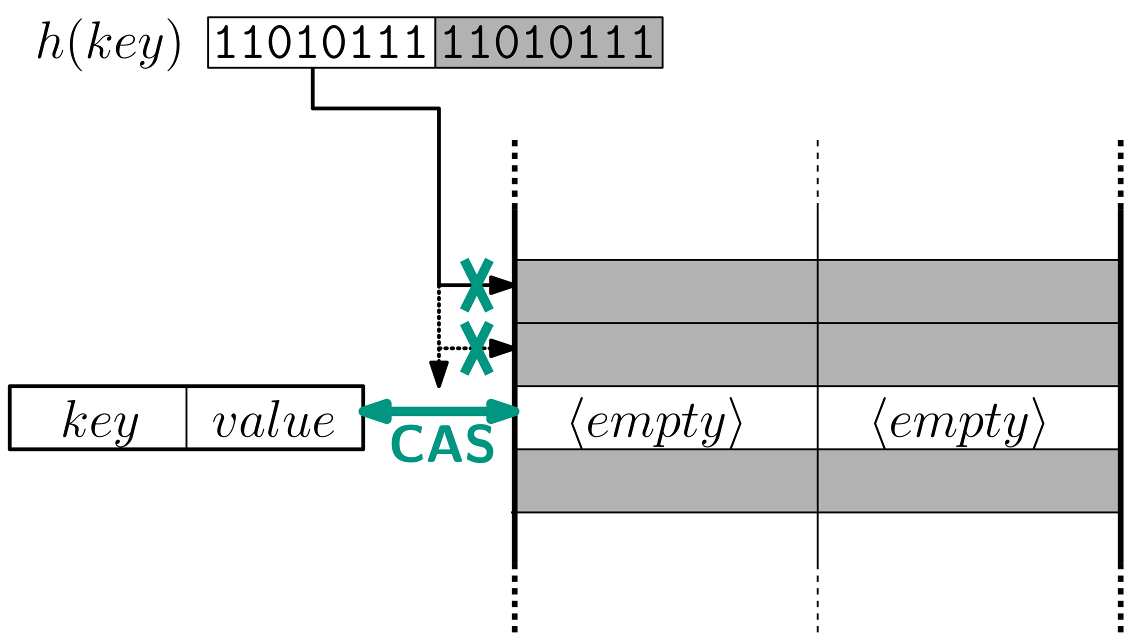
Tobias Maier[1], Peter Sanders[1], and Roman Dementiev[2].

[1]Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2]Intel Deutschland GmbH

Starting Point

- Circular linear probing
- Each entry can be changed atomically (CAS) ($|key| + |value| = 128$ bit)
- Bound capacity 2–4$\times n$ size
- Reserved keys for $\langle empty \rangle$ and $\langle deleted \rangle$
- Adressing using the most significant digits of $h(key)$

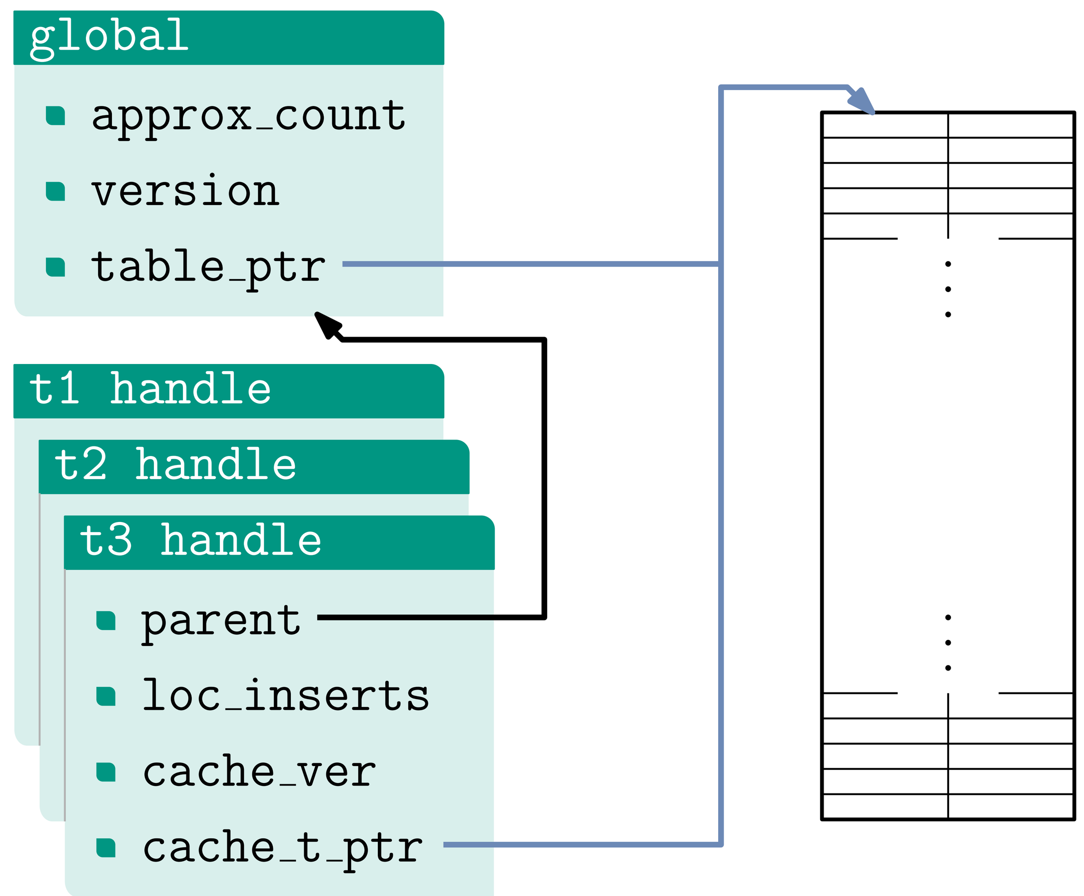
insert(key, $value$)



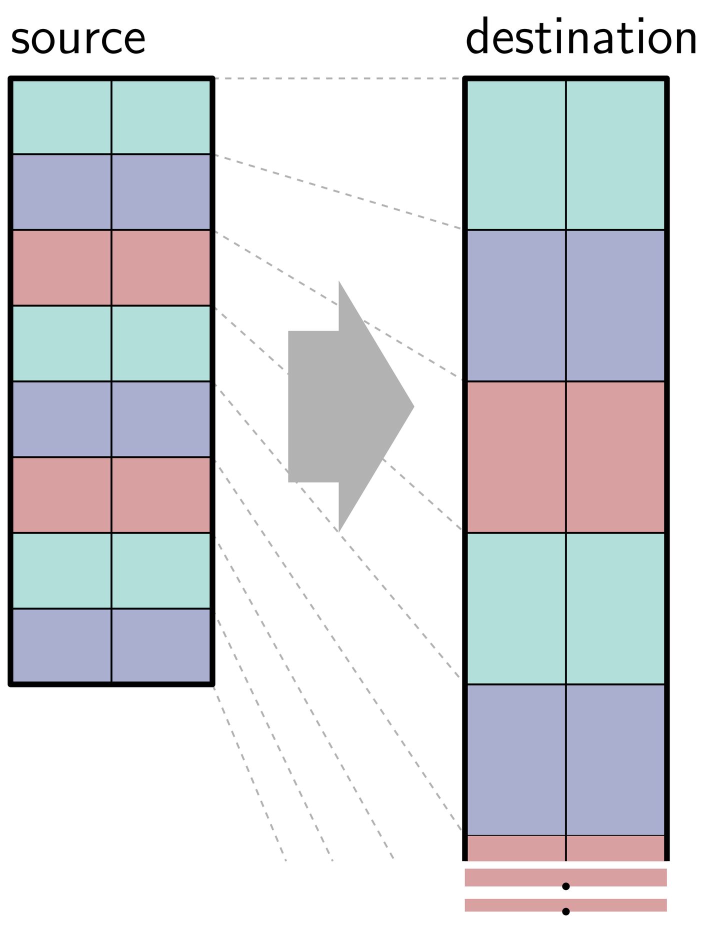
Architecture/Table Management

- **Global Object:** stores the current table and some data; does not offer any functions (except create Handle)
- **Handle Object:** stores threadlocal data and exposes the hash table functionality; cannot be shared between threads

- To approximate element count, count local insertions. Update the global count probabilistically every $\approx p$ insertions. *error* $\in O(p^2)$
- Use `std::shared_ptr` to ensure safe deallocation. Cache the shared pointer to reduce overheads. Compare version numbers before every operation.



Migration with minimal Synchronization

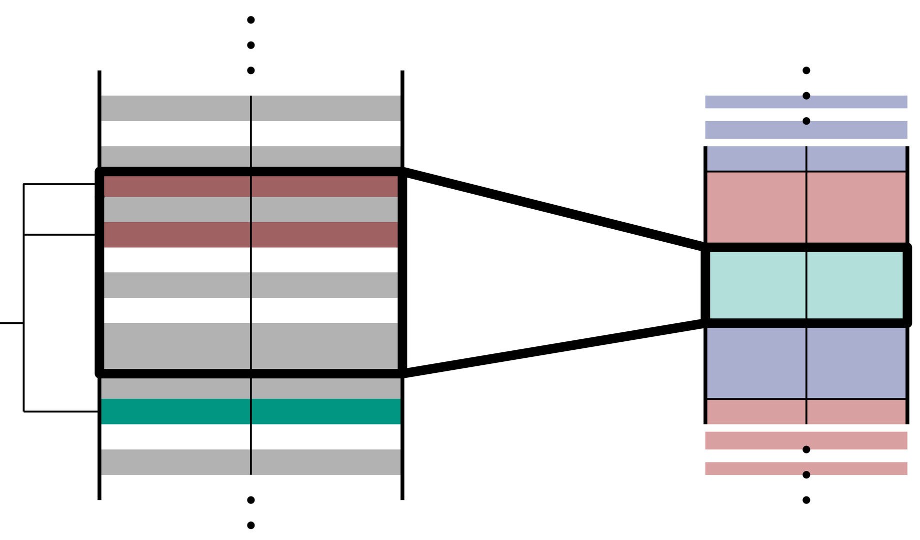


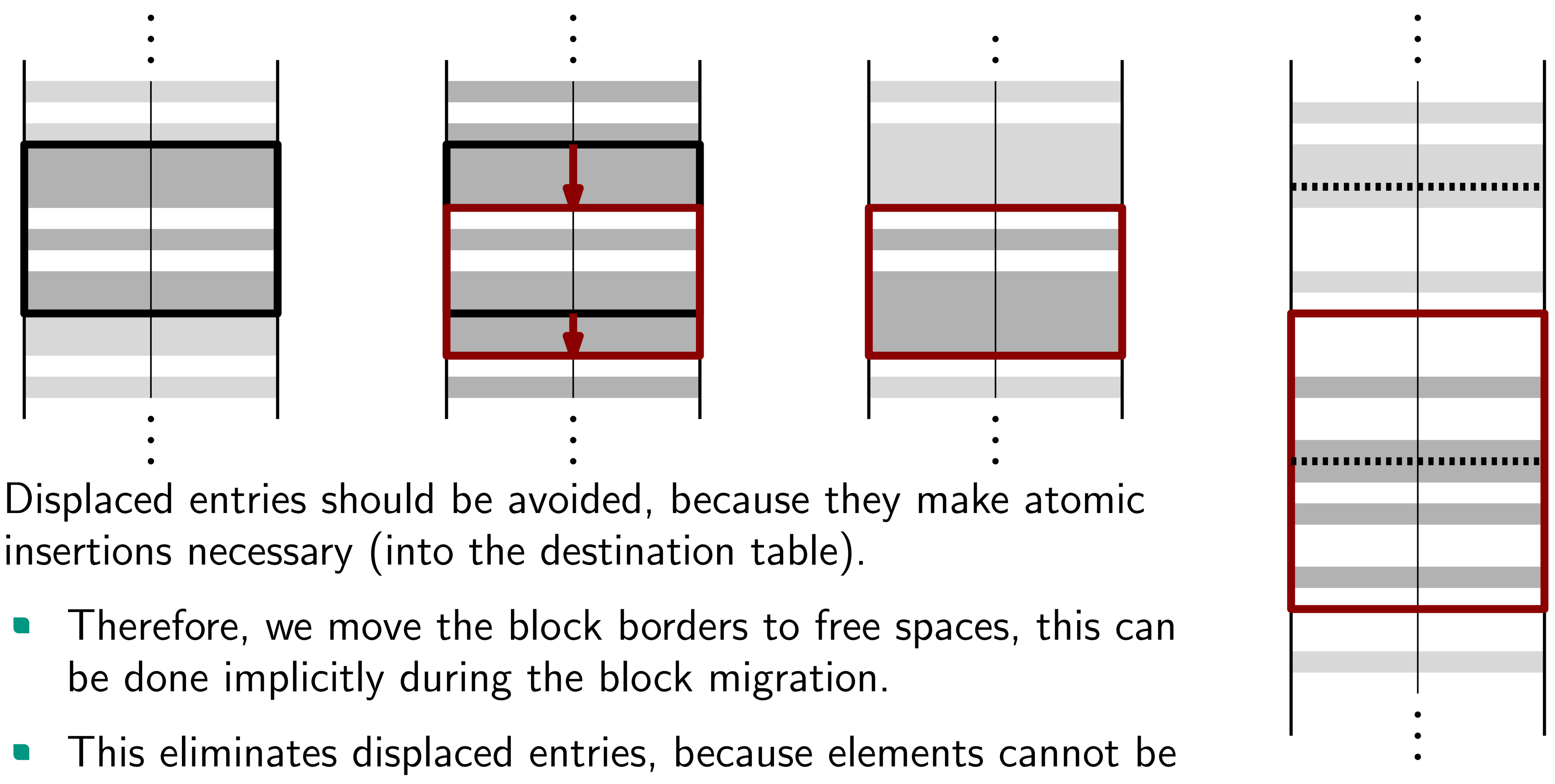
1. Allocate destination table (uninitialized)
2. Whenever a thread accesses the source table while it is growing that thread is forced to help migrate the table (no synchronized start)
3. The source table is separated into constant sized blocks (here 4096)
4. Blocks are dynamically distributed between participating threads
5. Elements within one block are migrated into the corresponding block in the destination table (for details see below).

All entries stored within one block are either

- hashed into that block **OR**
- displaced into that block through linear probing.

Displaced elements can only occur within the first cluster of the block





Displaced entries should be avoided, because they make atomic insertions necessary (into the destination table).

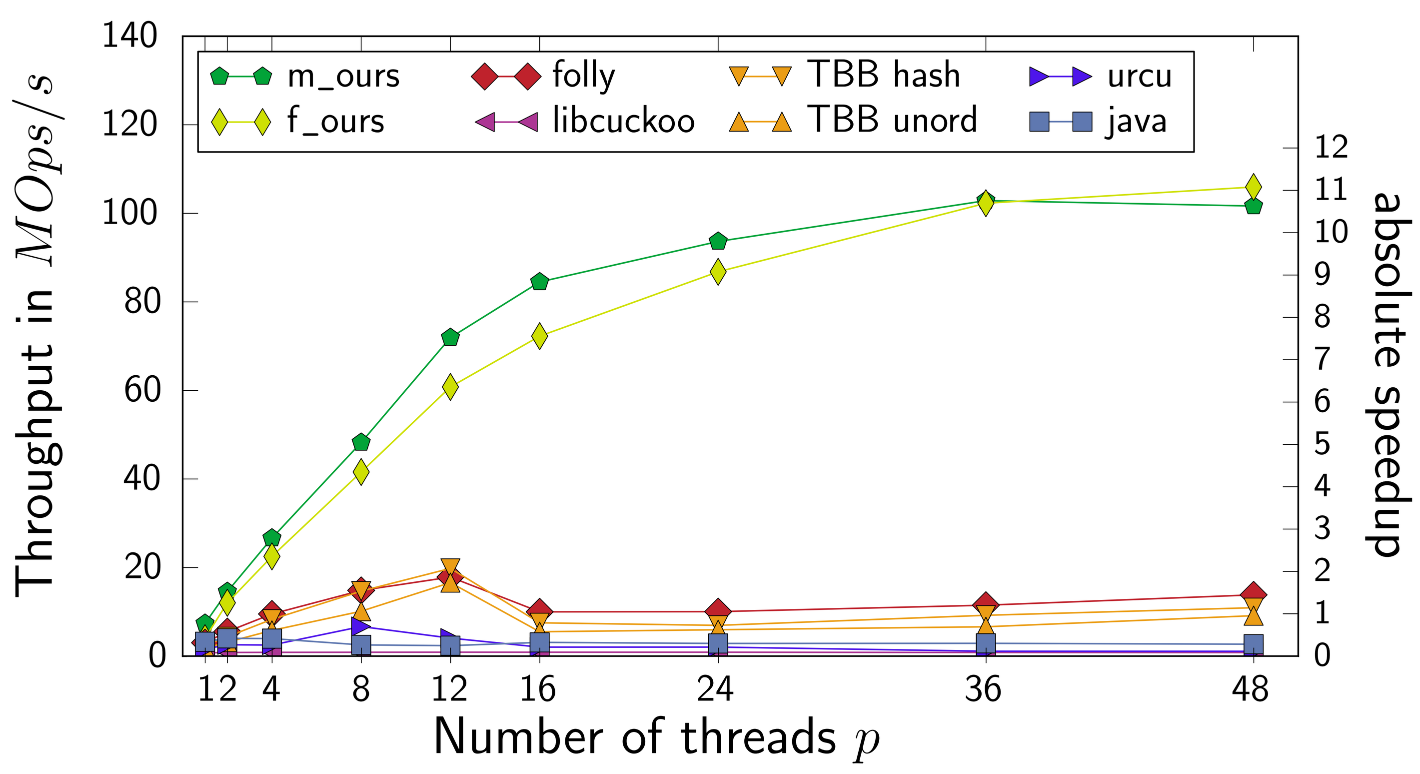
- Therefore, we move the block borders to free spaces, this can be done implicitly during the block migration.
- This eliminates displaced entries, because elements cannot be displaced over empty cells (insertions in the destination table can be done non-atomically).
- The expected size difference between a block and the corresponding implicit block is bound by a small constant

We also implemented two different options to ensure atomicity in the source table

- Marking copied elements (m_ours)
- Using flags to ensure that no update can operate concurrently to the migration (f_ours)

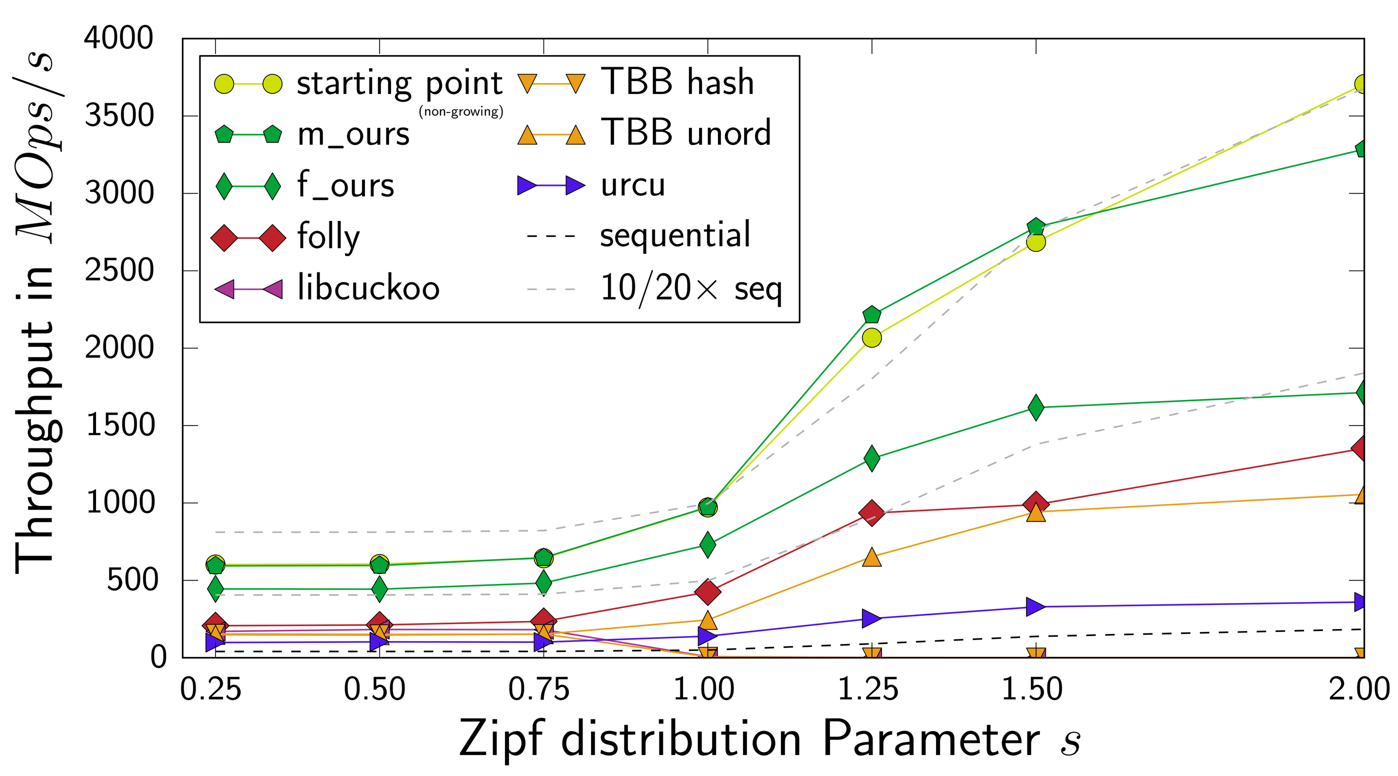
Experiments

Insertions (growing needed)



- Measured by inserting 100 000 000 elements (strong scaling)
- Our tables were initialized with 4096 cells.
- The competitors were initalized with 50 % of the target size.

Successful find (with contention)



- Measured by searching 100 000 000 keys.
- Searched keys are Zipf distributed ($P(key = k) = \frac{1}{k^s \cdot H_{N,s}}$)
- Every searched key was previously inserted. To make sure that the table size doesn't shrink under high contention we inserted 100 000 000 additional elements.
- Using 48 threads (24 cores + hyperthreading)

Test Setup

- dual-socket 2×12 cores with 2.3 GHz each
- Intel Xeon E5-2670 v3 (codenamed Haswell-EP)
- 128 GB RAM
- each measurement is the average of 5 runs