UNIVERSITY OF TWENTE.

A Parallel Compact Hash Table

Alfons Laarman & Steven van der Vegt

Overview

Research Motivation

Background

Contribution

Introduction

- Hash tables are fundamental data structures

Introduction

- Hash tables are fundamental data structures
- Compact hash tables: memory efficient hash tables

Introduction

- Hash tables are fundamental data structures
- Compact hash tables: memory efficient hash tables
- Useful in i.e. Model checking, planning, BDDs, Tree tables

Introduction

- Hash tables are fundamental data structures
- Compact hash tables: memory efficient hash tables
- Useful in i.e. Model checking, planning, BDDs, Tree tables
- Problem: No concurrent implementation of concurrent hash tables

Introduction

- Hash tables are fundamental data structures
- Compact hash tables: memory efficient hash tables
- Useful in i.e. Model checking, planning, BDDs, Tree tables
- Problem: No concurrent implementation of concurrent hash tables
- Our contribution: A scalable lockless algorithm for compact hashing

Goals

- Parallel compact hash table
- Scalable
  - Fast: lockless
  - Memory efficient: no pointers (otherwise we lose the benefits from compact hashing)
- Focus on findOrPut
  - Already sufficient Model checking (monotonic growing dataset)
  - subsumes individual find and put operations

Overview

Research Motivation

Background

Contribution

Hashing Revisited

- A hash table stores a subset of a key universe U into an table T of buckets
  typically $|U| \gg |T|$
- Multiple keys can be mapped upon 1 bucket
- The full key is stored in T to resolve collisions
- Several possible collision resolution algorithms, i.e. linear probing

Hashing Revisited - Example

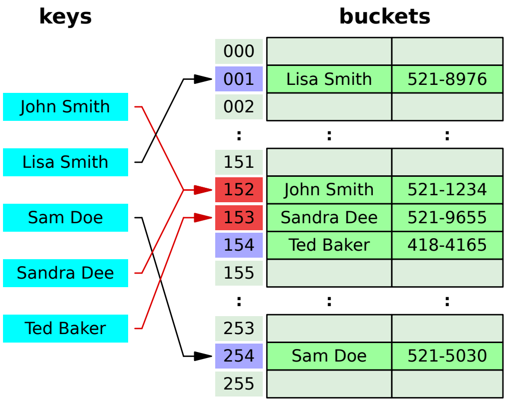

Figure: Example of an open addressing hash table.

Introduction Into Compact Hash Tables

- If however $|U| \leq |T|$, we only need a bit array! (and a perfect hash function)
- What if $|U|$ just slightly bigger than $|T|$? Cleary Tables:
  1. Maintain order in T
  2. Add three bits to buckets in T

Introduction Into BLP

Let K be the set of possible keys and h the hash function which computes the indexes. $h : K \to \{0..M-1\}$ with the property $K_1, K_2 \in K | K_1 \leq L_2 \textbf{iff } h(K_1) \leq h(K_2)$

- *All keys are stored in ascending order.*
- *There can not be empty locations between a keys original hash location and its actual storage position.*
- All keys sharing the same initial hash location form one continuous *group*.
- Groups can grow together forming *clusters* of groups.
- Bidirectional linear probing algorithm (probing possible in both directions)

Introduction Into BLP - Insert Example

Inserting k into table T in 5 steps:

1. Determine index: $i \leftarrow h(k)$
2. Determine probing direction $T[h(k)] > k ? right : left$
3. Search empty bucket
4. Insert K into empty bucket
5. Swap bucket into correct place

Cleary Table

Cleary administration bits:

- **Virgin** Set upon a bucket if its location is the initial hash location for some key in the tables
- **Change** Set at the beginning of a group with the same initial hash location
- **Occupied** Set if the bucket contains a key

Cleary Table - Example

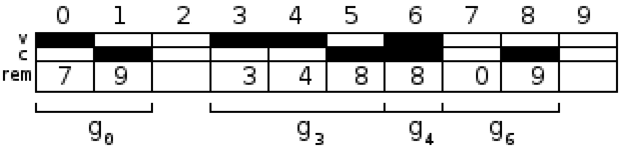

Figure: Example of a partially filled Cleary table with 4 groups.

Overview

Research Motivation

Background

Contribution

Requirements for Parallelizing

We need a write-exclusive locking mechanism that

- Scales well
- Is memory efficient

Locking Mechanism

Properties:

- 1 bit per bucket

Locking Mechanism

Properties:

- 1 bit per bucket
- CAS(a,b,c) - Compare-and-Swap (**if** $a == b$ **then** $a \leftarrow c$)

Locking Mechanism

Properties:

- 1 bit per bucket
- CAS(a,b,c) - Compare-and-Swap (**if** $a == b$ **then** $a \leftarrow c$)

Locking steps:

1. Search for both left and right bucket of cluster

Locking Mechanism

Properties:

- 1 bit per bucket
- CAS(a,b,c) - Compare-and-Swap (**if** $a == b$ **then** $a \leftarrow c$)

Locking steps:

1. Search for both left and right bucket of cluster
2. Lock these buckets

Locking Mechanism

Properties:

- 1 bit per bucket
- CAS(a,b,c) - Compare-and-Swap (**if** $a == b$ **then** $a \leftarrow c$)

Locking steps:

1. Search for both left and right bucket of cluster
2. Lock these buckets
3. If one of these locks fails $\rightarrow$ unlock and start over

Locking Mechanism

Properties:

- 1 bit per bucket
- CAS(a,b,c) - Compare-and-Swap (**if** $a == b$ **then** $a \leftarrow c$)

Locking steps:

1. Search for both left and right bucket of cluster
2. Lock these buckets
3. If one of these locks fails $\rightarrow$ unlock and start over
4. Perform exclusive actions (read, write)

Dynamic Region Based Locking

```
left ← CL-LEFT(h)
right ← CL-RIGHT(h)
if ¬TRY-LOCK(T[left]) then
    RESTART
if ¬TRY-LOCK(T[right]) then
    UNLOCK(T[left])
    RESTART
if FIND(k) then                         ▷ exclusive read
    UNLOCK(T[left], T[right])
    return FOUND
PUT(k)                                  ▷ exclusive write
UNLOCK(T[left], T[right])
```

Benchmarks - Speedup

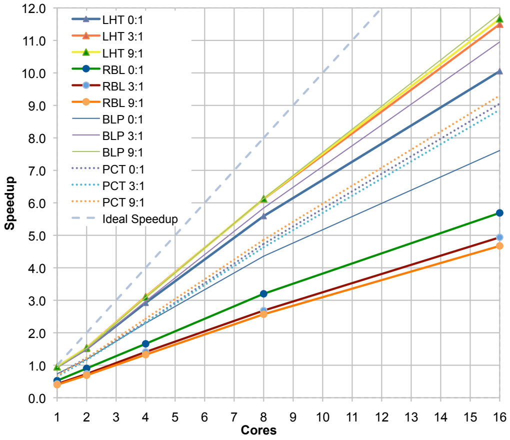

Figure: Speedups of BLP, RBL, LHT and PCT with r/w ratios 0:1, 3:1 and 9:1

Benchmarks - Runtime

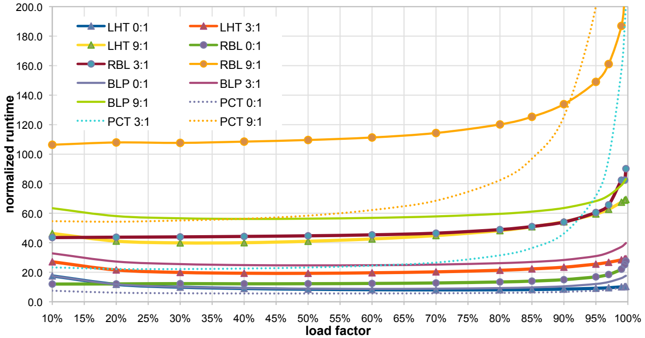

Figure: 16-core runtimes of BLP, RBL, LHT and PCT with r/w ratios 0:1, 3:1 and 9:1.

Results

- PCT performs very good with only inserts,
- PCT's performance drops when the load-factor becomes above the 85%
- With a high amount of reads ¿ (9:1) BLP eventually becomes faster than LHT
- Region based locking with OS-locks is very slow as can be seen in RBL
- scalability of both PCL and BLP is good.
- r/w ratio: r/w exclusion on clusters takes a toll.
  there is room for improvement if look at the higher load factors (when clusters are large)

Conclusion

- We have realized parallel cleary with high performance and scalability up to load-factors of 90%
  Since the compression ratio of compact hash tables can be high, this is acceptable
- Future work: Allow for concurrent reads with cleary to improve scalability of Cleary even more