An Efficient Wait-free Resizable Hash Table

Panagiota Fatourou[1,2], Nikolaos Kallimanis[1], Thomas Ropars[3]

[1] FORTH ICS
[2] University of Crete
[3] Univ. Grenoble Alpes

A new combination of properties for a hash table

Context

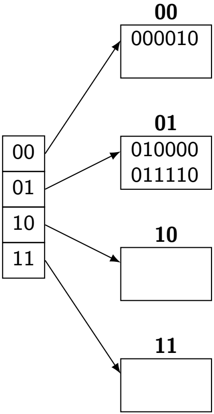
- Dictionary of Key-Value pairs
- Important data structure in several domains (OS, etc.)

A resizable hash table

- Provides the strongest progress guarantee (wait-freedom)
- Targets the most common load for a hash table
  - Large majority of LOOKUP operations
- Outperforms existing non-blocking algorithms for such workloads
  - By enforcing 2 important design rules

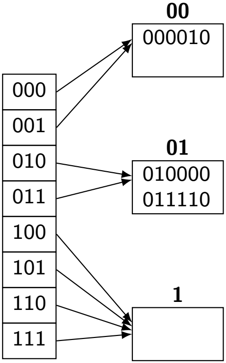
Hash tables

- A hash function associates items to buckets
  - Fixed-size buckets
- 3 operations:
  - INSERT(K, V) (If K already exists, V is updated)
  - DELETE(K)
  - LOOKUP(K)



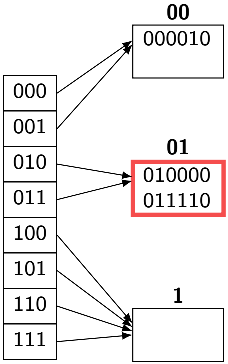
Dynamic hashing

- Adapts the number of buckets to the number of items
- Ensures constant average time for operations



Dynamic hashing

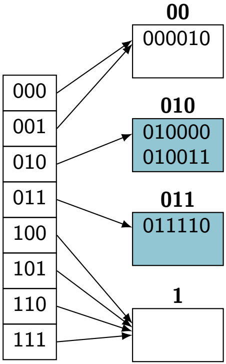
- Adapts the number of buckets to the number of items
- Ensures constant average time for operations



Insert(010011)

Dynamic hashing

- Adapts the number of buckets to the number of items
- Ensures constant average time for operations



000
001
010
011
100
101
110
111

00
000010

010
010000
010011

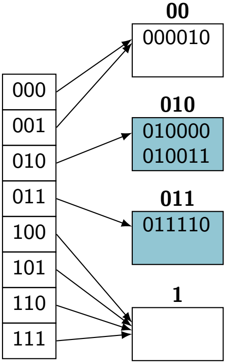
011
011110

1

Insert(010011)

Extendible hashing

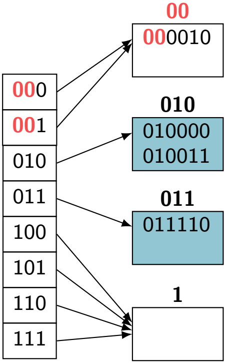
- Hash keys manipulated as bit strings
  - A prefix of the key is used to find the appropriate bucket
- Resizing actions are local
  - Splitting and merging buckets



Insert(010011)

Extendible hashing

- Hash keys manipulated as bit strings
  - A prefix of the key is used to find the appropriate bucket
- Resizing actions are local
  - Splitting and merging buckets

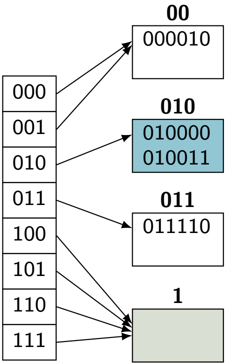


Insert(010011)

A wait-free concurrent hash table

Natural parallelism

- Operations applying to different parts of the hash table can run in parallel
- More complex with dynamic hashing



T_A: Insert(100000)

T_B: Insert(010011)

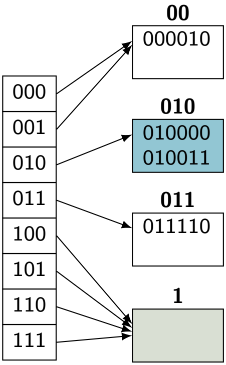
A wait-free concurrent hash table

Natural parallelism

- Operations applying to different parts of the hash table can run in parallel
- More complex with dynamic hashing

Non-blocking algorithm

- Lock freedom: At least one thread makes progress
- **Wait freedom**: Every operation completes in a finite number of steps



T_A: Insert(100000)

T_B: Insert(010011)

Towards an efficient resizable hash table: Insights

Most common load for a hash table

- Large majority of LOOKUP() operations
- Resizing actions are rare

Design rules to achieve best performance

- LOOKUP() operations should always be allowed to proceed without any synchronization
- When no resizing actions are executed, update operations applying to different buckets should be allowed to progress fully in parallel

Related work

The split-ordered list (LF-Split)

- Shalev and Shavit [PODC'03]
- LF-Split does not comply with our design rules
  - During LOOKUP() operations, threads have to help removing items marked for deletion.
  - A global counter is modified after every insertion/deletion.

LF/WF-Freeze

- Liu, Zhang, and Spear [PODC'14]
- WF-Freeze does not comply with our design rules
  - A global sequence number is required to tag update operations

Contributions

The design of a wait-free extendible hash table

- Follows our two design rules
- First algorithm to use several instances of the PSim universal construction [SPAA'11].
  - Appropriatly synchronized to ensure wait-freedom

Experiments demonstrate the new performance trade-off

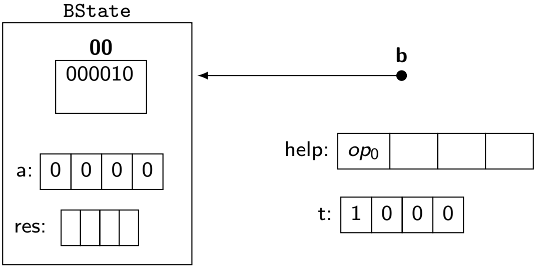
- Outperforms all existing non-blocking resizable hash tables when resizing actions are rare
- Slower resizing

Our Wait-Free Algorithm

The PSIM algorithm

Fatourou and Kallimanis [SPAA'11]

T_2: Insert(001110)



BState

00

000010

a: | 0 | 0 | 0 | 0 |

res: | | | | |

b

help: | op_0 | | | |

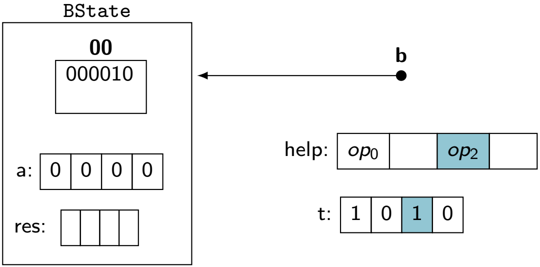
t: | 1 | 0 | 0 | 0 |

The PSIM algorithm

Fatourou and Kallimanis [SPAA'11]

```
Announce the operation to be executed
```

T_2: Insert(001110)



a:				
	0	0	0	0

res:				

help:				
	op_0		op_2	

t:				
	1	0	1	0

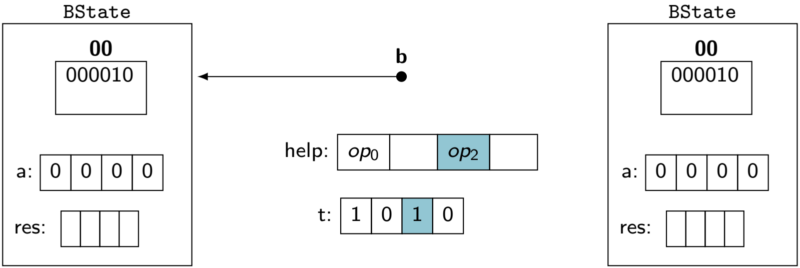
The PSim algorithm

Fatourou and Kallimanis [SPAA'11]

```
Announce the operation to be executed

    Make a local copy of the object to update
```

T_2: Insert(001110)



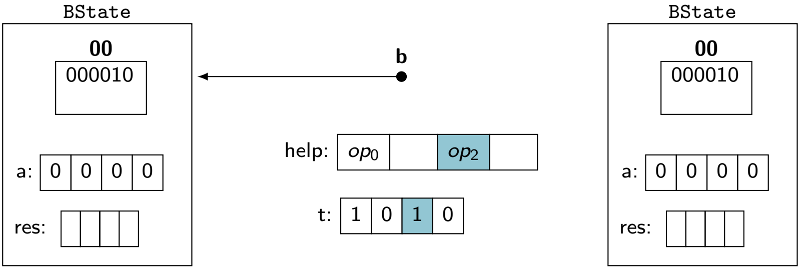
The PSIM algorithm

Fatourou and Kallimanis [SPAA'11]

```
Announce the operation to be executed

    Make a local copy of the object to update
    Apply all pending operations on the local object
```

T_2: Insert(001110)



BState

00

000010

a: 0 0 0 0

res:

b

help: op_0 | | op_2 |

t: 1 0 1 0

BState

00

000010

a: 0 0 0 0

res:

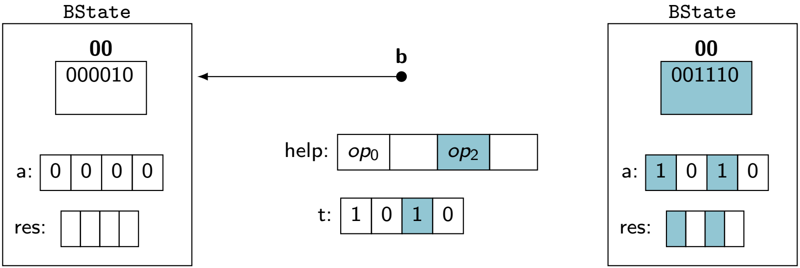
The PSim algorithm

Fatourou and Kallimanis [SPAA'11]

```
Announce the operation to be executed

    Make a local copy of the object to update
    Apply all pending operations on the local object
```

T_2: Insert(001110)



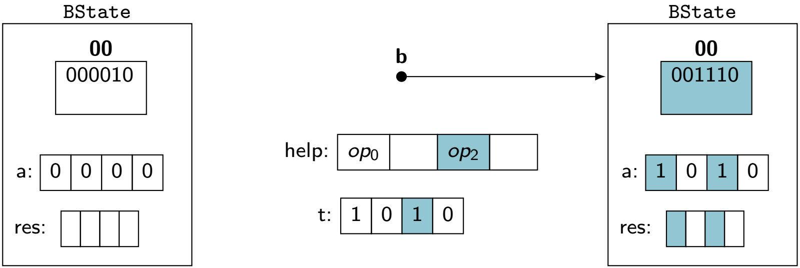
The PSIM algorithm

Fatourou and Kallimanis [SPAA'11]

```
Announce the operation to be executed

    Make a local copy of the object to update
    Apply all pending operations on the local object
  Try making the object globally visible using CAS
```

T_2: Insert(001110)

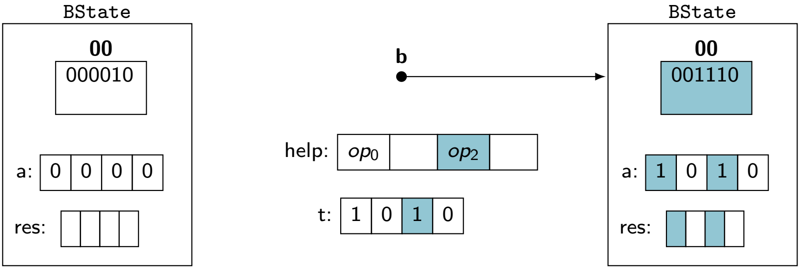


The PSIM algorithm

Fatourou and Kallimanis [SPAA'11]

```
Announce the operation to be executed
for k in 1..2:
    Make a local copy of the object to update
    Apply all pending operations on the local object
  Try making the object globally visible using CAS
```

T_2: Insert(001110)



BState

00

000010

a: 0 0 0 0

res:

b

help: op_0 op_2

t: 1 0 1 0

BState

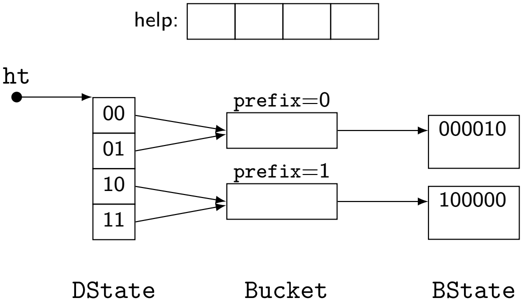
00

001110

a: 1 0 1 0

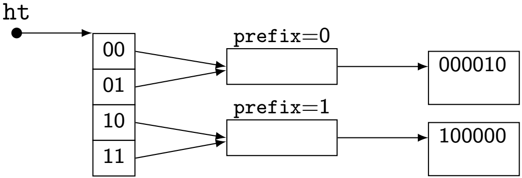
res:

The hash table structure



- Two levels of indirection
- One instance of PSIM for the DState and for each BState

INSERT (no resizing) and LOOKUP operations

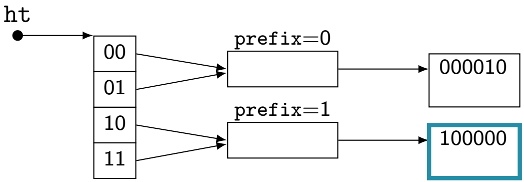


T_a: Insert(111100)

T_b: Lookup(100010)

- LOOKUP operations are executed without any synchronization (`BState` objects are immutable)
- INSERT operations on different buckets do not synchronize

INSERT (no resizing) and LOOKUP operations



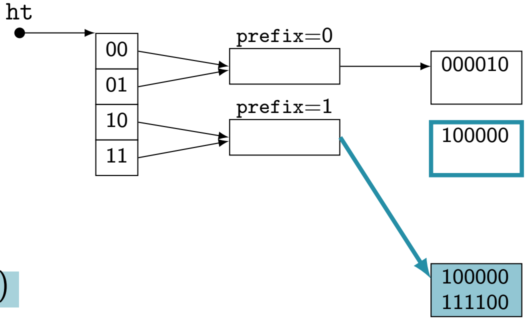
T_a: Insert(111100)

100000
111100

T_b: Lookup(100010)

- LOOKUP operations are executed without any synchronization (`BState` objects are immutable)
- INSERT operations on different buckets do not synchronize

INSERT (no resizing) and LOOKUP operations

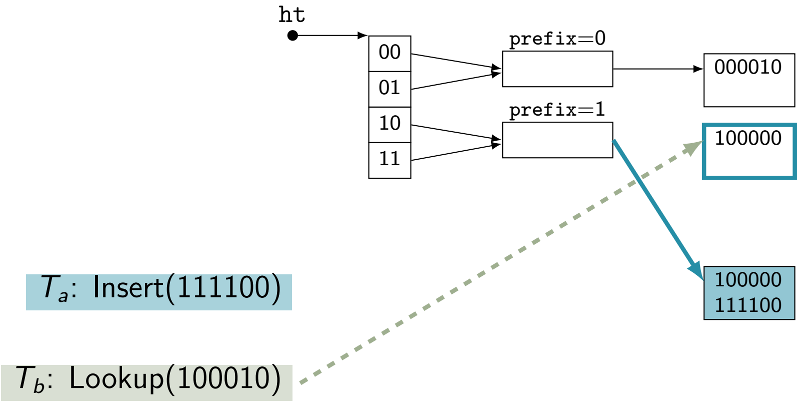


T_a: Insert(111100)

T_b: Lookup(100010)

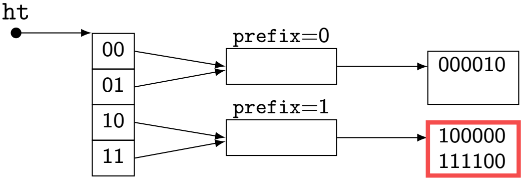
- LOOKUP operations are executed without any synchronization (BState objects are immutable)
- INSERT operations on different buckets do not synchronize

INSERT (no resizing) and LOOKUP operations



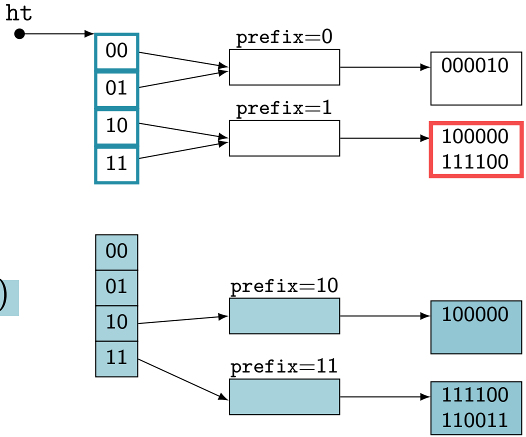
- LOOKUP operations are executed without any synchronization (BState objects are immutable)
- INSERT operations on different buckets do not synchronize

Splitting a bucket



T_a: Insert(110011)

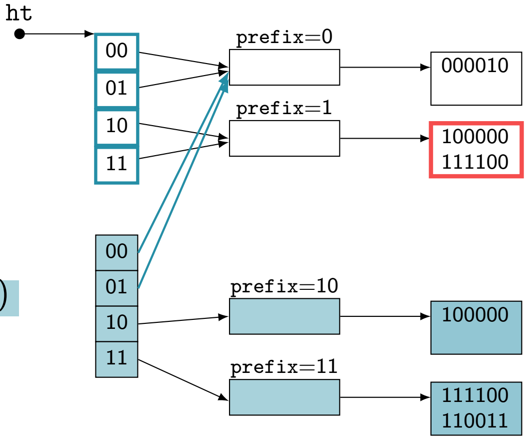
Splitting a bucket



T_a: Insert(110011)

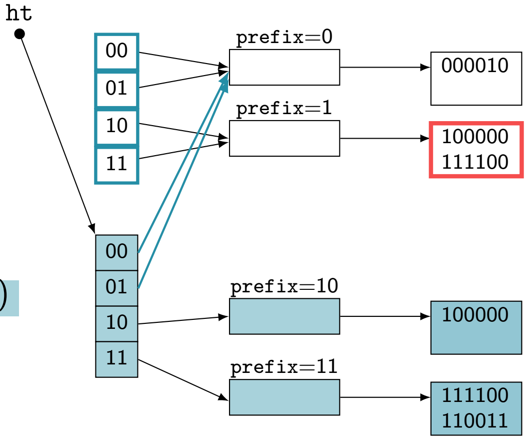
Splitting a bucket

T_a: Insert(110011)

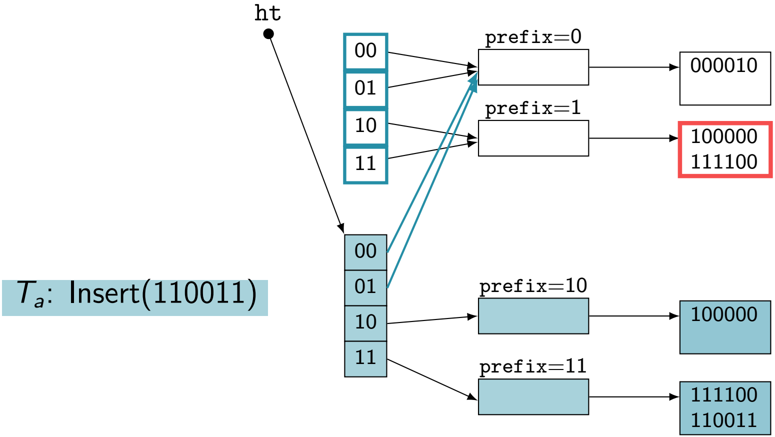


Splitting a bucket

T_a: Insert(110011)



Splitting a bucket

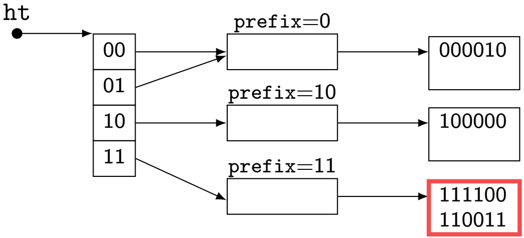


T_a: Insert(110011)

To avoid losing updates:

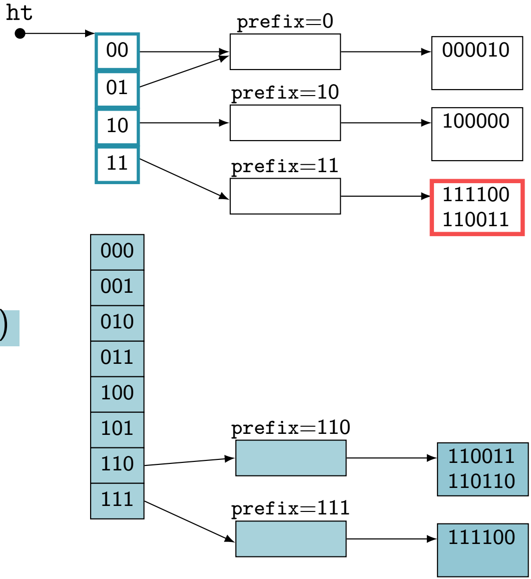
- Only full buckets can be replaced during resizing
- No update operation can be run on a full bucket

Increasing the directory size



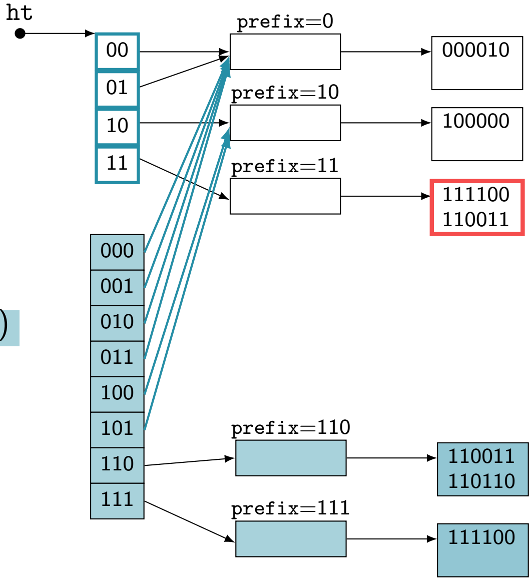
T_a: Insert(110110)

Increasing the directory size



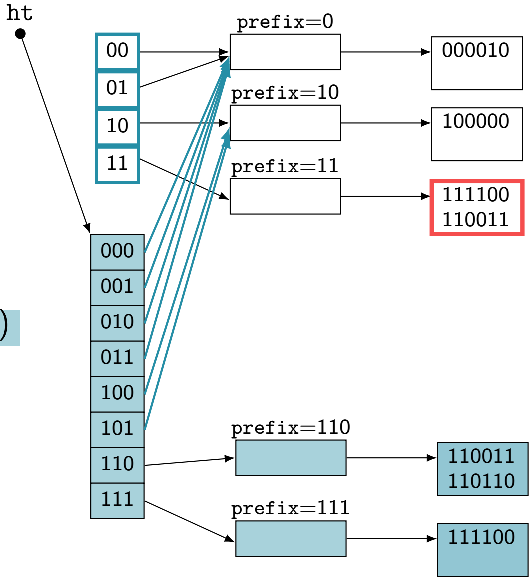
T_a: Insert(110110)

Increasing the directory size



T_a: Insert(110110)

Increasing the directory size



T_a: Insert(110110)

Ensuring wait-freedom

The problem

- Ensuring that updates on `DState` and `BState` objects are wait-free is not enough to ensure that the operations on the hash table are wait-free

Example

1. Thread T_a tries to insert in bucket B → full
2. T_a tries to update the directory → already done
3. T_a tries to insert in bucket B' → full
4. T_a tries to update the directory …

Ensuring wait-freedom

The problem

- Ensuring that updates on `DState` and `BState` objects are wait-free is not enough to ensure that the operations on the hash table are wait-free

Solution

- When resizing the directory, all pending updates applying to full buckets should be run

Executing each operation exactly once

The problem

- An INSERT operation can be applied directly on a `BState` or through a resizing action
  - How to ensure that an operation is never executed twice?

Example

1. Thread T_a wants to run an INSERT operation
   - It registers its operation in the help array
2. Thread T_b executes the operation of T_a during a resizing action
3. T_a access the bucket B where it should execute its operation
   - Has its operation already been executed?

Executing each operation exactly once

The problem

- An INSERT operation can be applied directly on a `BState` or through a resizing action
  - How to ensure that an operation is never executed twice?

Solution

- Per-thread sequence numbers are used to tag operations
- The sequence number of the last applied operation is stored in each `BState`
- Sequence numbers are evaluated before executing an update operation

Experimental Evaluation

Implementation

Our proposed algorithm (WF-Ext)

- Implementation in C
- Epoch-based memory reclamation
- Efficient memory allocation of `BState` objects

State-of-the-art algorithms

Reference C implementations and modified versions:

- LF-Split-M: Modified version to avoid the global counter
- LF-Freeze-M:
  - Implementation of our semantic for INSERT operations
  - Integration of our efficient memory allocator
  - **Recall**: WF-Freeze is much slower than LF-Freeze

Evaluation setup

Hardware

- 64-**core machine** with 4 NUMA nodes (Intel Broadwell)

Software

- System memory allocator: tests with the *glibc* allocator and TCMalloc
- NUMA policy: tests with *Local* and *Interleave* policies

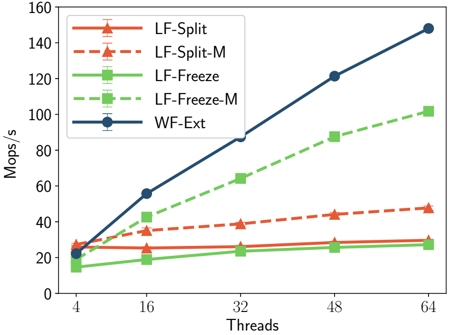
Methodology

- Average over 10 runs
- All combinations of parameters are tested for each algorithm

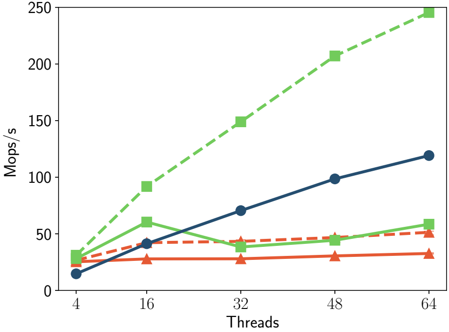
Throughput with 90% LOOKUPS (directory stable)

Description of the experiment:

- Initial state: half-full hash table
- 5% INSERT ops; 5% DELETE ops



1K items



256K items

Conclusion

A wait-free extendible hash table

- Follows two design rules to preserve the natural parallelism of such data structures
- Synchronizes several instances of the PSim algorithm to acheive wait-freedom

A new performance trade-off

- Outperforms existing lock-free algorithms when resizing actions are rare
- Slower resizing actions
  - Amortized over long runs

References

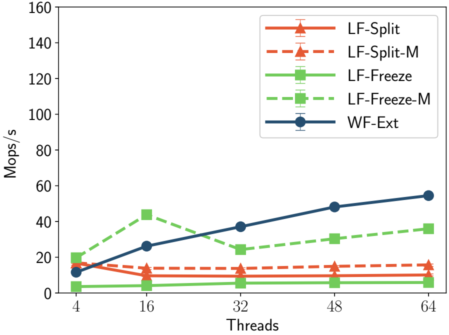
[1] Yujie Liu, Kunlong Zhang, and Michael Spear. "Dynamic-sized Nonblocking Hash Tables". *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*. PODC '14. Paris, France, 2014.

[2] Panagiota Fatourou and Nikolaos D. Kallimanis. "Highly-Efficient Wait-Free Synchronization". *Theory of Computing Systems* (2013), pp. 1–46.

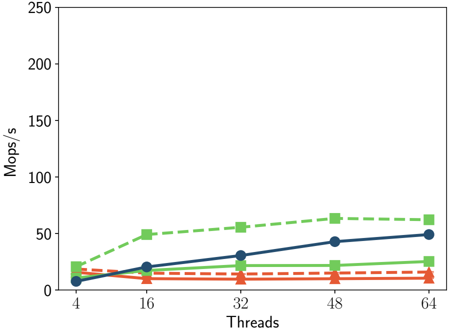
[3] Ori Shalev and Nir Shavit. "Split-ordered lists: Lock-free extensible hash tables". *Journal of the ACM* 53.3 (2006), pp. 379–405.

Thanks!

Throughput with 50% Lookups (directory stable)



1K items

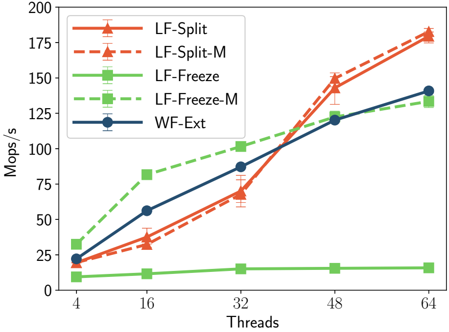


256K items

Performance with resizing

Description of the experiment

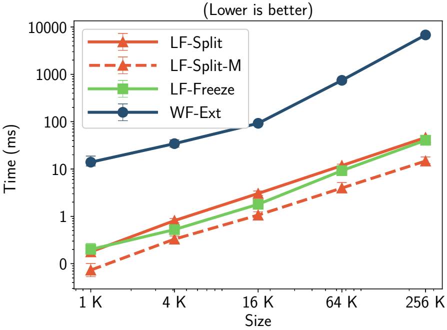
- Initial state: Empty hash table with 2 buckets
- 90% Lookup ops; 10% Insert ops
- Throughput with 1K items over a 5 second run



Resizing efficiency

Description of the experiment

- Inital state: empty hash table with 2 buckets
- 50% LOOKUP ops; 50% INSERT ops
- Measurement: Time to reach final size



Additional information

Merging buckets

- Buckets to be merged have to be *frozen*
- A merging action may fail

Compliance with our design rules

- LOOKUP operations are executed without any synchronization
- When no resizing is needed, an update operation is executed by the PSIM instance of the bucket

Avoiding losing updates

The problem

- Since an update operation on a bucket might be run in parallel with resizing the directory, how to avoid loosing updates?

Example

1. Thread T_a updates bucket B during an update operation
2. Thread T_b changes the directory during a resizing action
3. Is the update made by T_a visible in the new directory *published* by T_b?

Avoiding losing updates

The problem

- Since an update operation on a bucket might be run in parallel with resizing the directory, how to avoid loosing updates?

Solution

- For non-full buckets:
  - The two levels of indirection ensure that the update of T_a remains accessible
- For full buckets:
  - Updates on full buckets are not allowed