Concurrent Robin Hood Hashing

IRISH RESEARCH COUNCIL
An Chomhairle um Thaighde in Éirinn

Supervisor: Barak A. Pearlmutter

Motivations

- Make improvements on a 10 year old state of the art.
- Provide the first concurrent Robin Hood Hashing in the literature.

Contributions

- First linearisable concurrent variant of Robin Hood Hashing.
- Strong application of new *K-CAS* developments [Arbel-Raviv,, Brown; 2016]
- Competitive performance compared to state of the art concurrent hash tables.

General talk structure

- Hash table and Robin Hood background
- Challenges with concurrent Robin Hood
- What are the options?
- Solution
- Correctness/Progress
- Evaluation

Hash Tables

- Constant time O(1) set/map structures
- Set operations:

1. **Contains(Key)**
2. **Add(Key)**
3. **Remove(Key)**

- No need for sorting of keys, unlike tree-based sets/maps
- Require a hash function for keys
- Applications: Search, Object representation in VMs/interpretors, caches...

Hash Tables

Divided into two camps: Open vs Closed Addressing.

Open Addressing.

- Items are stored in individual buckets only.
- If bucket is already taken find a new one: Collision algorithm.

Closed Addressing.

- Items are stored at original bucket only.
- Typically in a linked list structure.

Robin Hood Hashing (Open Addressing)

Robin Hood [Celis ;86]

Robin Hood [Celis ;86]

Motto: Steal from the rich and give to the poor.

Robin Hood [Celis ;86]

Motto: Steal from the rich and give to the poor.

Search: Linear probing with culling.

Robin Hood [Celis ;86]

Motto: Steal from the rich and give to the poor.

Search: Linear probing with culling.

Insertion: Linear probing with conditional recursive displacement.

Robin Hood [Celis ;86]

Motto: Steal from the rich and give to the poor.

Search: Linear probing with culling.

Insertion: Linear probing with conditional recursive displacement.

Removal: Backward shifting. More on that later.

Robin Hood [Celis ;86]

Motto: Steal from the rich and give to the poor.

Search: Linear probing with culling.

Insertion: Linear probing with conditional recursive displacement.

Removal: Backward shifting. More on that later.

Definition: The number of buckets away an entry is from its ideal bucket - **D**istance **F**rom **B**ucket (*DFB*)

Robin Hood [Celis ;86]

Motto: Steal from the rich and give to the poor.

Definition: The number of buckets away an entry is from its ideal bucket - **D**istance **F**rom **B**ucket (*DFB*)

Search: Linear probing with culling.

Insertion: Linear probing with conditional recursive displacement.

If relocated item has bigger *DFB* than than current, kick current out, take spot, and recursively insert current further down the table.

Removal: Backward shifting. More on that later.

Linear Probing vs Robin Hood

Initial Table, inserting V.

∅	X_0	Y_1	Z_1	W_1	∅

Linear Probing vs Robin Hood

Initial Table, inserting V.

Key:

- Moved item
- Inserted item

$\emptyset$	X_0	Y_1	Z_1	W_1	$\emptyset$

Linear Probing Table

$\emptyset$	X_0	Y_1	Z_1	W_1	V_4

Linear Probing vs Robin Hood

Initial Table, inserting V.

Key:

- Moved item
- Inserted item

∅	X_0	Y_1	Z_1	W_1	∅

Linear Probing Table

∅	X_0	Y_1	Z_1	W_1	V_4

Robin Hood Table

∅	X_0	Y_1	V_2	Z_2	W_2

Linear Probing vs Robin Hood

Key:

- Moved item
- Inserted item

Initial Table, inserting V.

∅	X_0	Y_1	Z_1	W_1	∅

Linear Probing Table

∅	X_0	Y_1	Z_1	W_1	V_4

- Less work

Robin Hood Table

∅	X_0	Y_1	V_2	Z_2	W_2

- Less distance variance

Robin Hood Search

Robin Hood Search

Linear probe as normal.

Robin Hood Search

Linear probe as normal.

When you see someone not as far away as you: Stop.

Robin Hood Search

Linear probe as normal.

U_0

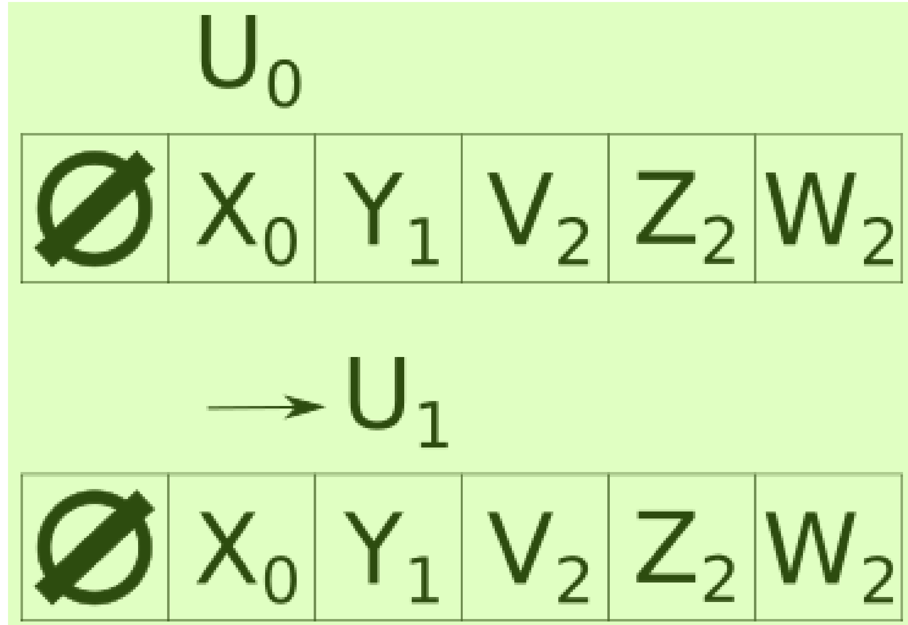
$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

When you see someone not as far away as you: Stop.

Robin Hood Search

Linear probe as normal.

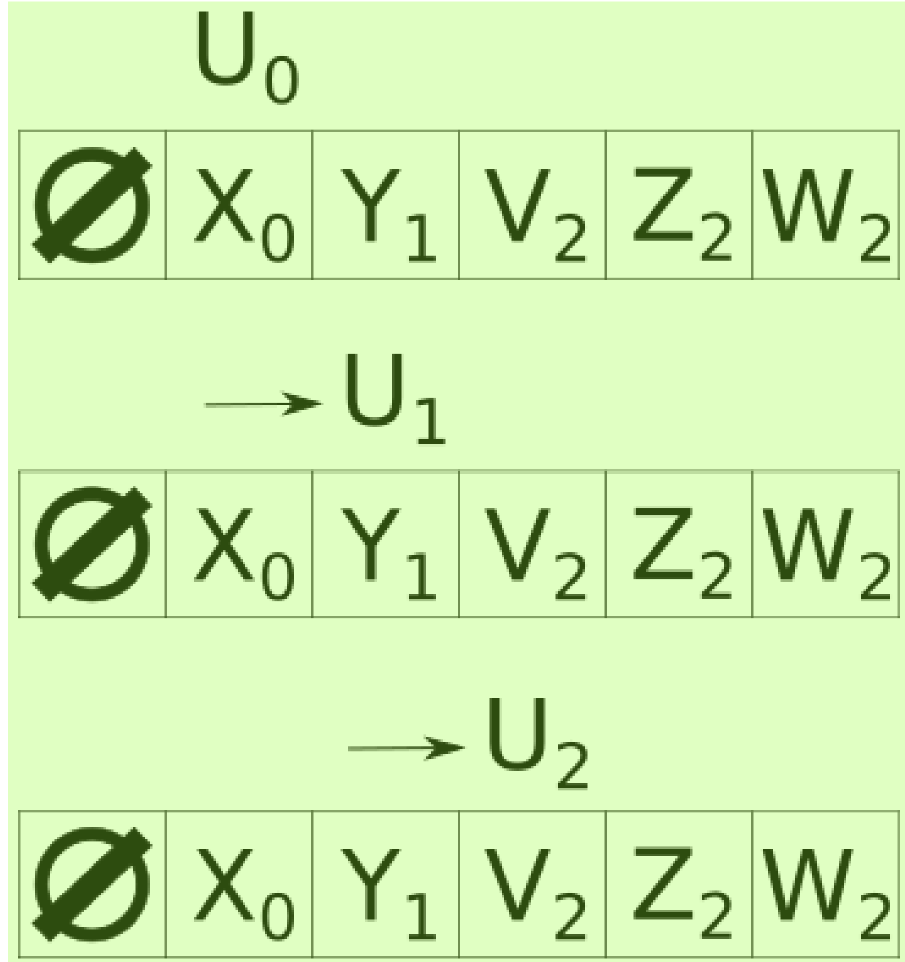
When you see someone not as far away as you: Stop.



Robin Hood Search

Linear probe as normal.

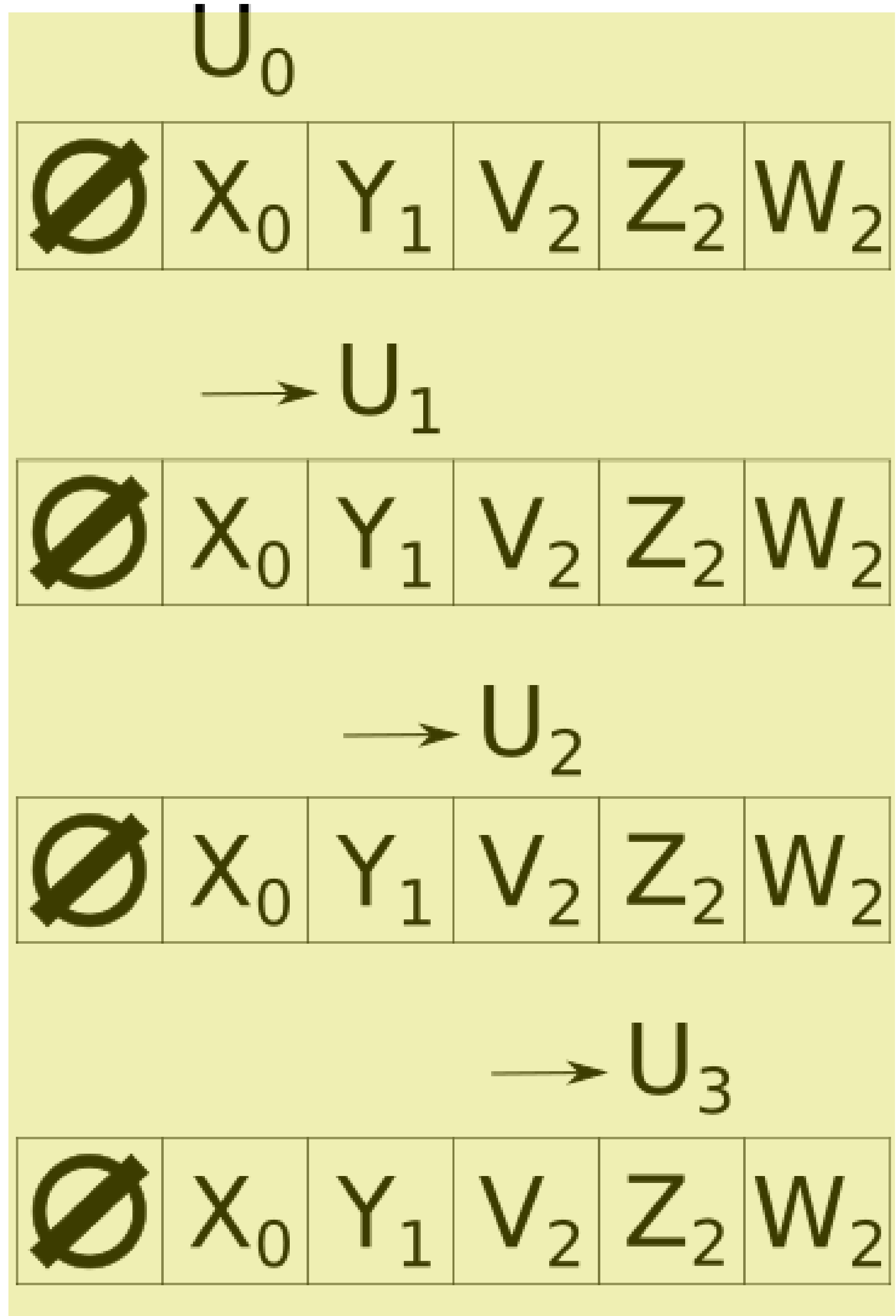
When you see someone not as far away as you: Stop.



Robin Hood Search

Linear probe as normal.

When you see someone not as far away as you: Stop.



Robin Hood benefits[1]

Robin Hood benefits

1. Fast, predictable performance:

- Optimised for reads - 2.6 probes per successful search.
- log(n) on failed search.
- Doesn't degenerate over time (poisoning).

Robin Hood benefits

1. Fast, predictable performance:

- Optimised for reads - 2.6 probes per successful search.
- log(n) on failed search.
- Doesn't degenerate over time (poisoning).

2. Relatively simple:

- No linked list or pointer manipulation.

Robin Hood benefits

1. Fast, predictable performance:

- Optimised for reads - 2.6 probes per successful search.
- log(n) on failed search.
- Doesn't degenerate over time (poisoning).

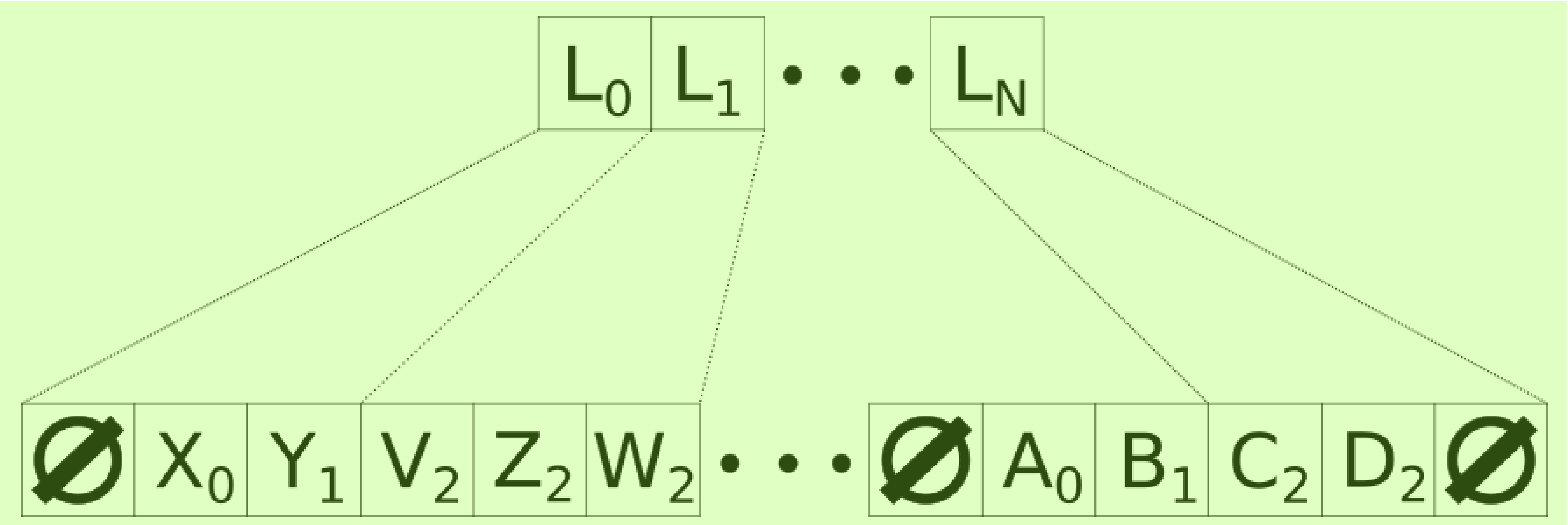
2. Relatively simple:

- No linked list or pointer manipulation.

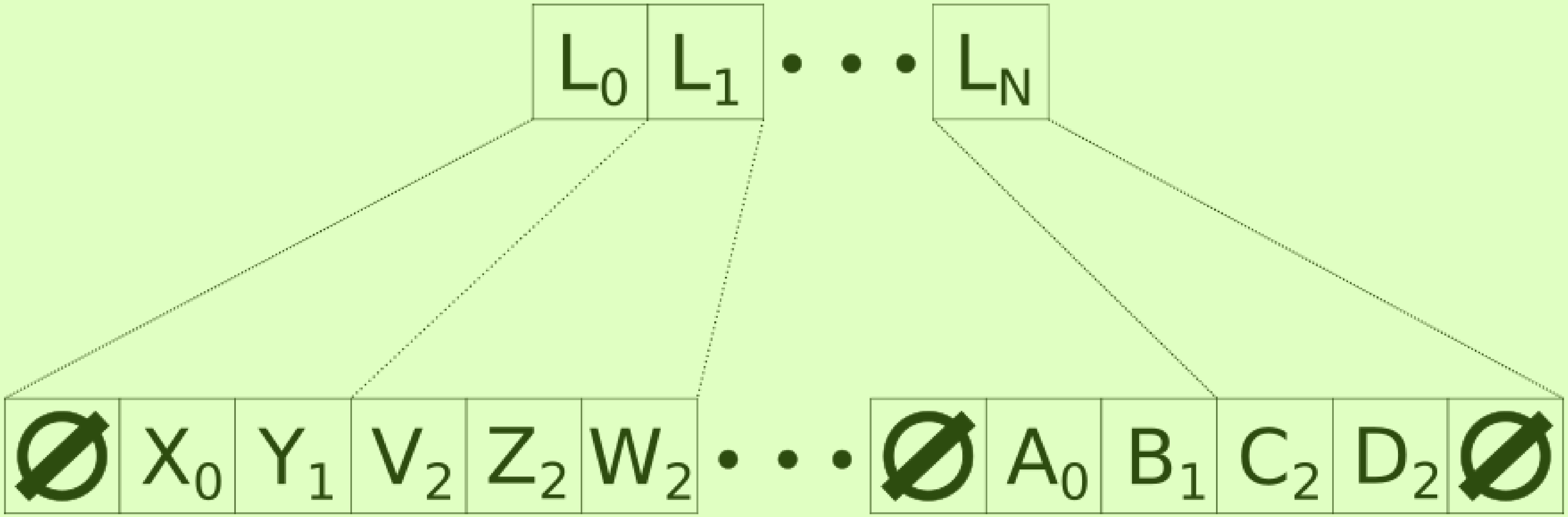
3. Cache efficient.

- Flat data, low probes.
- No dynamic allocation.
- Probes are generally on a single cache line.

Standard Solution: *Sharded* locks

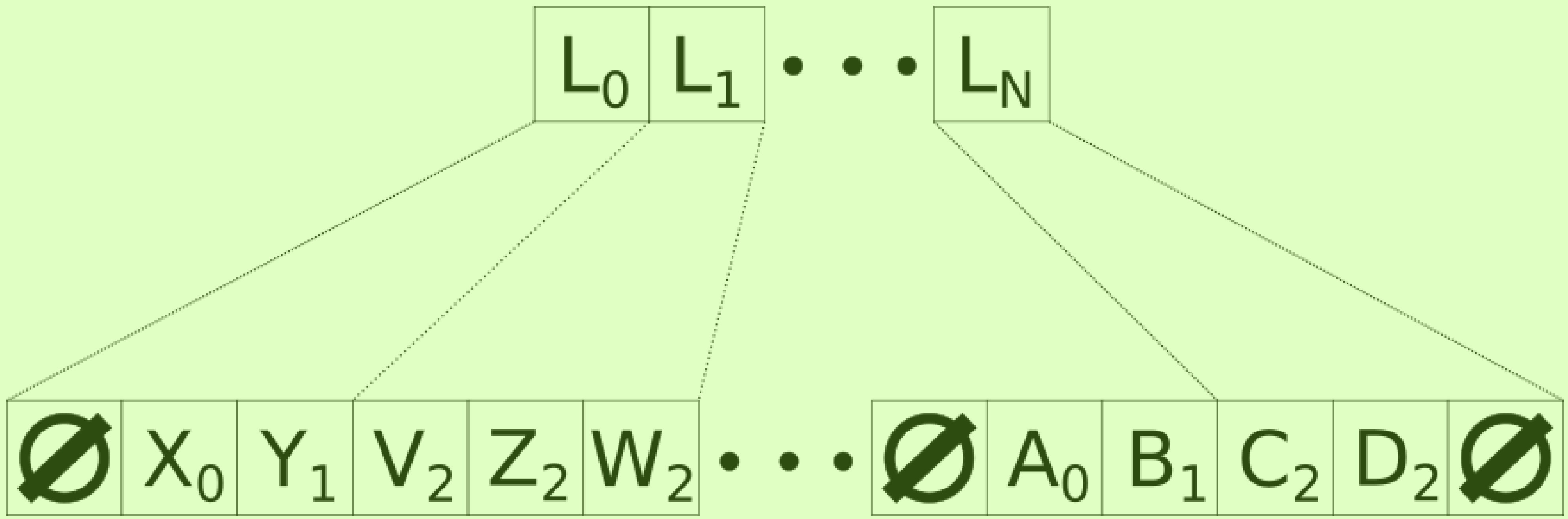


Standard Solution: *Sharded* locks



- Could grab multiple locks.
- Could result in deadlock, if allowed to wrap around.

Standard Solution: *Sharded* locks



- Could grab multiple locks.
- Could result in deadlock, if allowed to wrap around.
- Not very clean for our case: Need extra phantom segment to stop deadlock.
- Hacky. Slow. Lots of contention.

Contention: Remove

Contention: Remove

First part is normal search.

Contention: Remove

First part is normal search.

When you find the entry delete it, move everyone back by 1.

Contention: Remove

First part is normal search.

When you find the entry delete it, move everyone back by 1.

Stopping condition: The entry you're moving back has a distance of 0 or it's a Null.

Contention: Remove

First part is normal search.

When you find the entry delete it, move everyone back by 1.

Stopping condition: The entry you're moving back has a distance of 0 or it's a Null.

Initial table.

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

We want to delete Y.

Contention: Remove

First part is normal search.

When you find the entry delete it, move everyone back by 1.

Stopping condition: The entry you're moving back has a distance of 0 or it's a Null.

Initial table.

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

We want to delete Y.

Y deleted.

$\varnothing$	X_0	V_1	Z_1	W_1	$\varnothing$

Contention: Remove

First part is normal search.

When you find the entry delete it, move everyone back by 1.

Stopping condition: The entry you're moving back has a distance of 0 or it's a Null.

Great source of contention.

Initial table.

∅	X_0	Y_1	V_2	Z_2	W_2

We want to delete Y.

Y deleted.

∅	X_0	V_1	Z_1	W_1	∅

Problems with creating Concurrent Robin Hood

Problems with creating Concurrent Robin Hood

- Lots of moving parts. Performance problem.

Problems with creating Concurrent Robin Hood

- Lots of moving parts. Performance problem.
- An insert can trigger a global table reorganisation.

Problems with creating Concurrent Robin Hood

- Lots of moving parts. Performance problem.
- An insert can trigger a global table reorganisation.
- Cyclic lock grabbing. If locks are *sharded*, deadlock is possible.
- Huge contention on **Remove**.

Problems with creating Concurrent Robin Hood

- Lots of moving parts. Performance problem.
- An insert can trigger a global table reorganisation.
- Cyclic lock grabbing. If locks are *sharded*, deadlock is possible.
- Huge contention on **Remove**.

Possible Solutions

- Bespoke non-blocking solution
- Transactional Memory
- *K-CAS* (Multi-word compare and swap)

Issues with bespoke

Issues with bespoke

1. Use of dynamic memory, ruins cache locality. Slow.

Issues with bespoke

1. Use of dynamic memory, ruins cache locality. Slow.

2. Horrendously complicated. Difficult to even to get insertion working. Robin Hood invariant must hold all the time.

Issues with bespoke

1. Use of dynamic memory, ruins cache locality. Slow.

2. Horrendously complicated. Difficult to even to get insertion working. Robin Hood invariant must hold all the time.

3. End up reimplementing *K-CAS*. Why not just use *K-CAS*?

Issues with bespoke

1. Use of dynamic memory, ruins cache locality. Slow.

2. Horrendously complicated. Difficult to even to get insertion working. Robin Hood invariant must hold all the time.

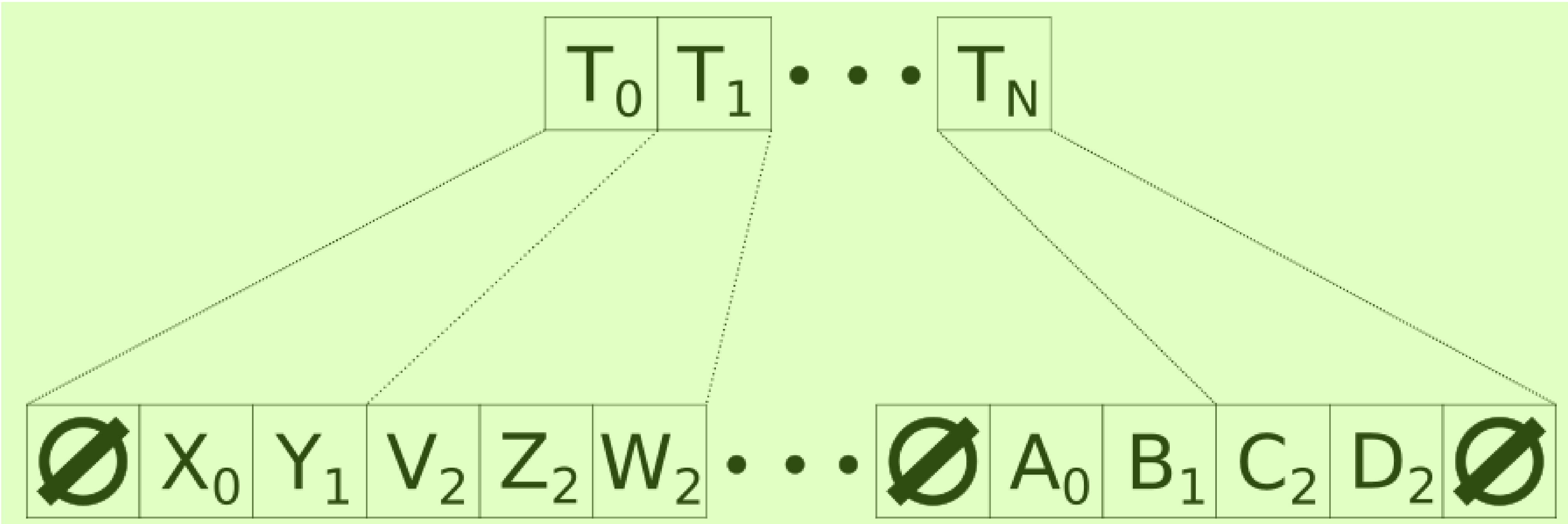
3. End up reimplementing *K-CAS*. Why not just use *K-CAS*?

Method Chosen: *K-CAS*

K-CAS is a *multi-word compare-and-swap* primitive. Each table operation is described as one large *K-CAS*.

Our Solution:

Sharded timestamps



Similar to lock-base *sharding*. Groups of timestamps *protect* the table.

Each relocation operation increments the timestamp. Except relocations can be done in bulk.

K-CAS without timestamps

Say our *K-CAS* solution just encapsulates every modifying operation {**Add**, **Remove**} into a *K-CAS* operation.

K-CAS without timestamps

Say our *K-CAS* solution just encapsulates every modifying operation {**Add**, **Remove**} into a *K-CAS* operation.

Readers of the table may miss entries that are moved backwards by remove.

K-CAS without timestamps

Try find V.

V_0

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

Say our *K-CAS* solution just encapsulates every modifying operation {**Add**, **Remove**} into a *K-CAS* operation.

Readers of the table may miss entries that are moved backwards by remove.

K-CAS without timestamps

Say our *K-CAS* solution just encapsulates every modifying operation {**Add**, **Remove**} into a *K-CAS* operation.

Readers of the table may miss entries that are moved backwards by remove.

Try find V.

V_0

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

$\longrightarrow V_1$

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

K-CAS without timestamps

Try find V.

V_0

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

$\longrightarrow V_1$

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

Say our *K-CAS* solution just encapsulates every modifying operation {**Add**, **Remove**} into a *K-CAS* operation.

Before check V get interrupted.

$\longrightarrow U_2$

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

Readers of the table may miss entries that are moved backwards by remove.

K-CAS without timestamps

Say our *K-CAS* solution just encapsulates every modifying operation {**Add**, **Remove**} into a *K-CAS* operation.

Readers of the table may miss entries that are moved backwards by remove.

Try find V.

V_0

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

$\longrightarrow V_1$

$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

Before check V get interrupted.

$\longrightarrow U_2$

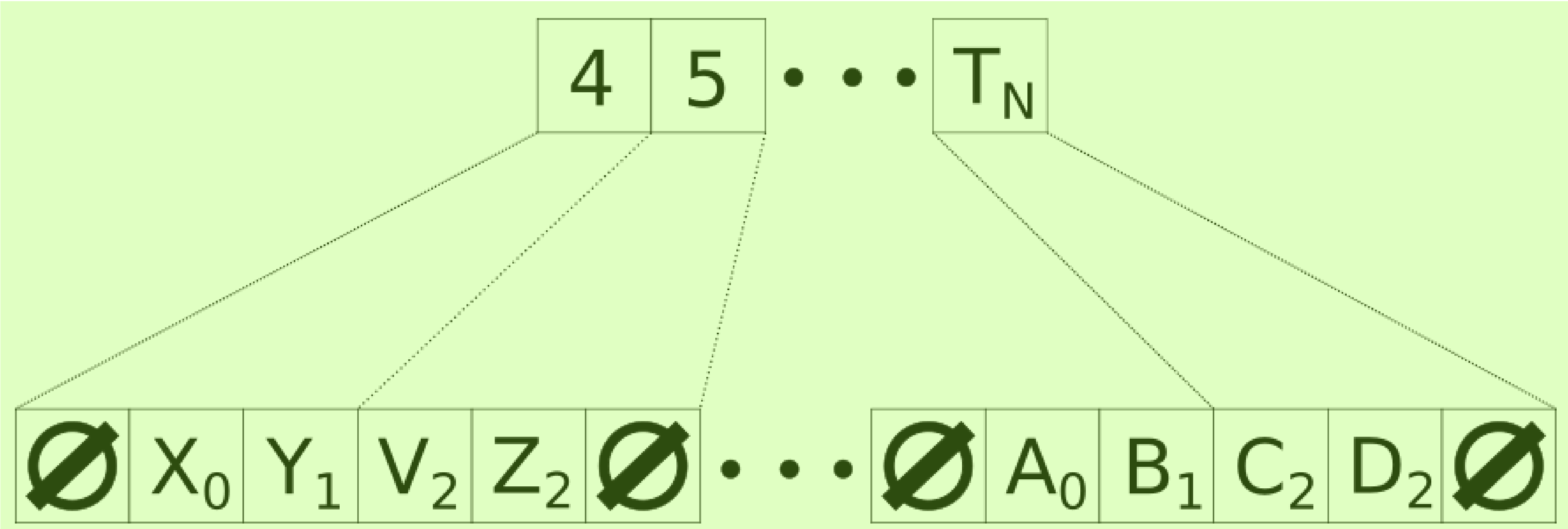
$\varnothing$	X_0	Y_1	V_2	Z_2	W_2

Delete Y. Move V back. Find Z, exit.

V_2

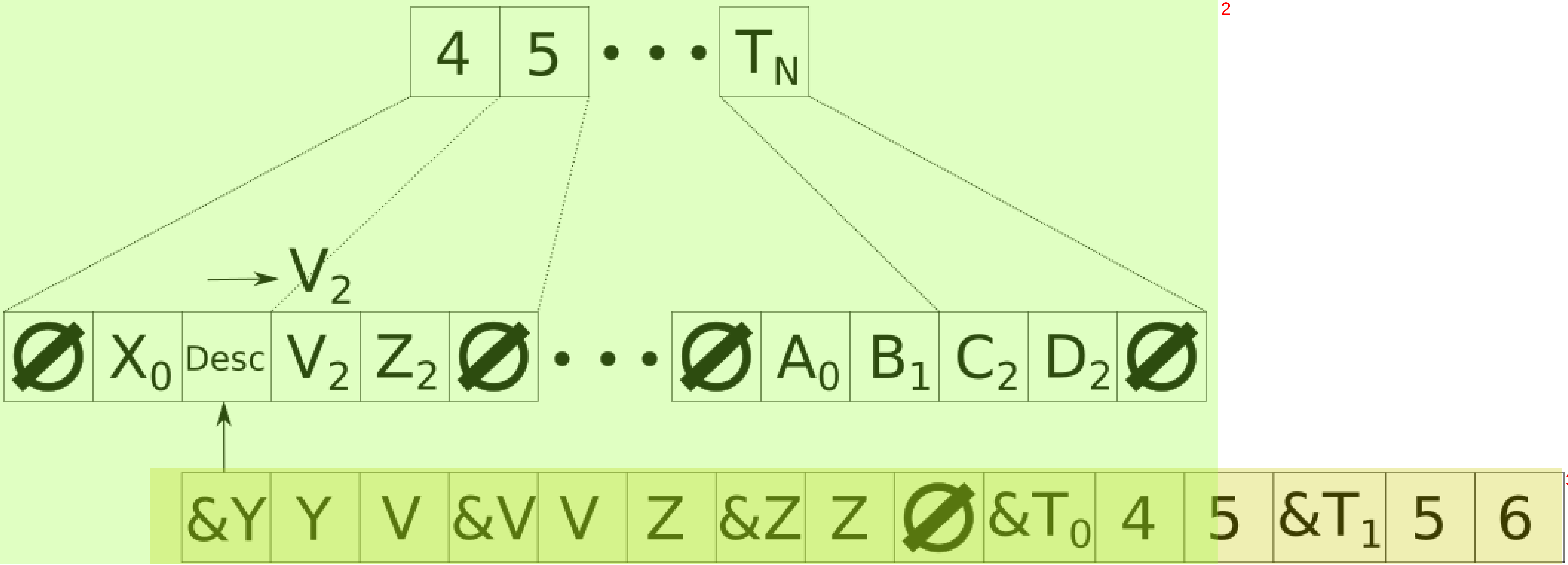
$\varnothing$	X_0	V_1	Z_1	W_1	$\varnothing$

Our Solution: Example



Going to delete Y from table, with concurrent reader.

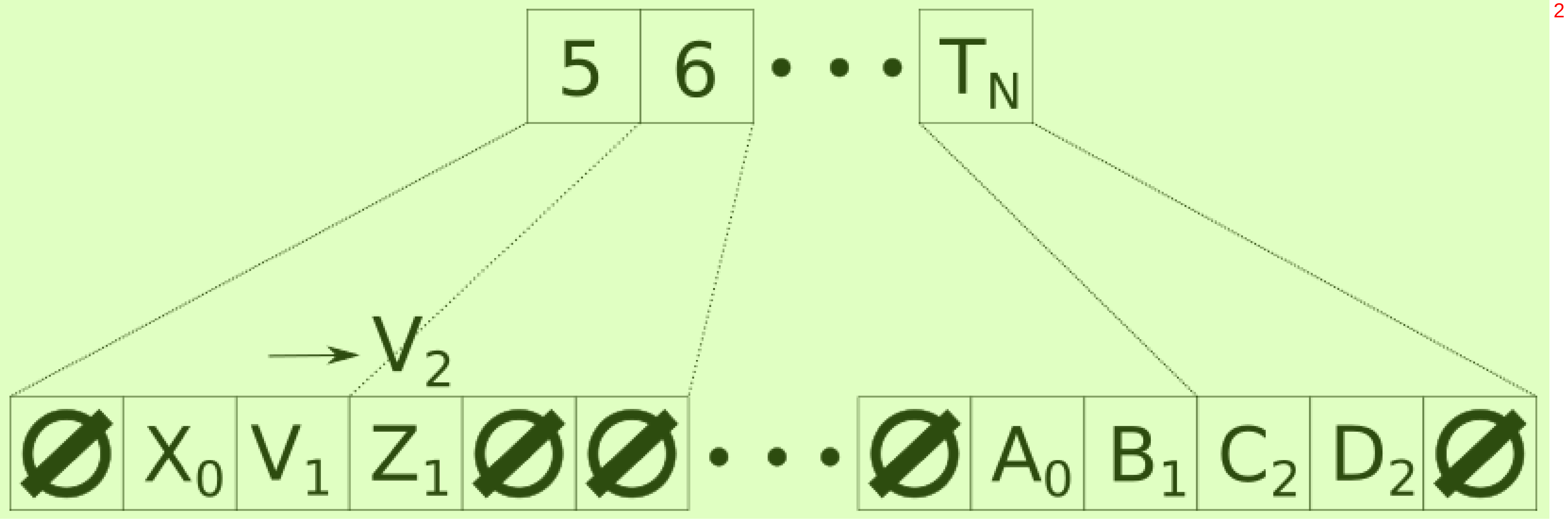
Our Solution: Example



Ugly little array is a deletion descriptor.

Moves items. Increments two timestamps.

Our Solution: Example



Reader misses V, due to deletion of Y.

Reader sees timestamp change, restarts operation.

Our Solution: Benefits

Our Solution: Benefits

Relatively simple design, close to the sequential algorithm.

Our Solution: Benefits

Relatively simple design, close to the sequential algorithm.

No dynamic memory, great cache performance. Minimal memory overhead.

Similar amount of CAS operations as bespoke dynamic memory solution.

Our Solution: Benefits

Relatively simple design, close to the sequential algorithm.

No dynamic memory, great cache performance. Minimal memory overhead.

Similar amount of CAS operations as bespoke dynamic memory solution.

Use of *K-CAS* allows for thread collaboration. Well defined non-blocking progress guarantees.

Bulk relocation greatly reduces contention. Fast.

Our Solution: Correctness

Our Solution: Correctness

Simple design means simple proof. Correctness is informally argued.

Our Solution: Correctness

IRISH RESEARCH COUNCIL
An Chomhairle um Thaighde in Éirinn

Simple design means simple proof. Correctness is informally argued.

Every modifying operation is a *K-CAS* operation. Cannot be seen midway.

Every reader must *remember* every timestamp seen.

Before any actions attempts to take effect they re-read timestamps. If any discrepancies are seen, retry operation.

Our Solution: Progress

Our Solution: Progress

Solution is obstruction-free/lock-free. Obstruction-free **Contains**, lock-free **Add** and **Remove**.

Our Solution: Progress

Solution is obstruction-free/lock-free. Obstruction-free **Contains**, lock-free **Add** and **Remove**.

Every operation checks timestamps before the operation completes. Timestamps are coarse so operations can impede each other.

The impeding of **Contains** means potentially no **Contains** will pass, but *at least* one **Add** or **Remove** will get through.

Benchmarking setup

Benchmarking setup

Hardware

- 4 x Intel® Xeon® CPU E7-8890 v3, 18 cores each, 2 threads per core, 144 threads in total
- HyperThreading avoided until the end
- PAPI used to measure various CPU artefacts
- *numactl* to control memory allocation

Benchmarking setup

Hardware

- 4 x Intel® Xeon® CPU E7-8890 v3, 18 cores each, 2 threads per core, 144 threads in total
- HyperThreading avoided until the end
- PAPI used to measure various CPU artefacts
- *numactl* to control memory allocation

Software

- Microbenchmark measuring operations per microsecond
- A number of strong performing concurrent hash tables
- Four load factors of 20%, 40%, 60%, and 80%
- Two read/write workloads of 10% and 20%

Tables - Explainer

Tables - Explainer

- Hopscotch Hashing [Herlihy, Shavit, Tzafrir; 2008]: Flattened separate chaining.

Tables - Explainer

- Hopscotch Hashing [Herlihy, Shavit, Tzafrir; 2008]: Flattened separate chaining.
- Lock-Free Linear Probing [Nielsen, Karlsson; 2016] : State simplified Purcell, Harris Table.

Tables - Explainer

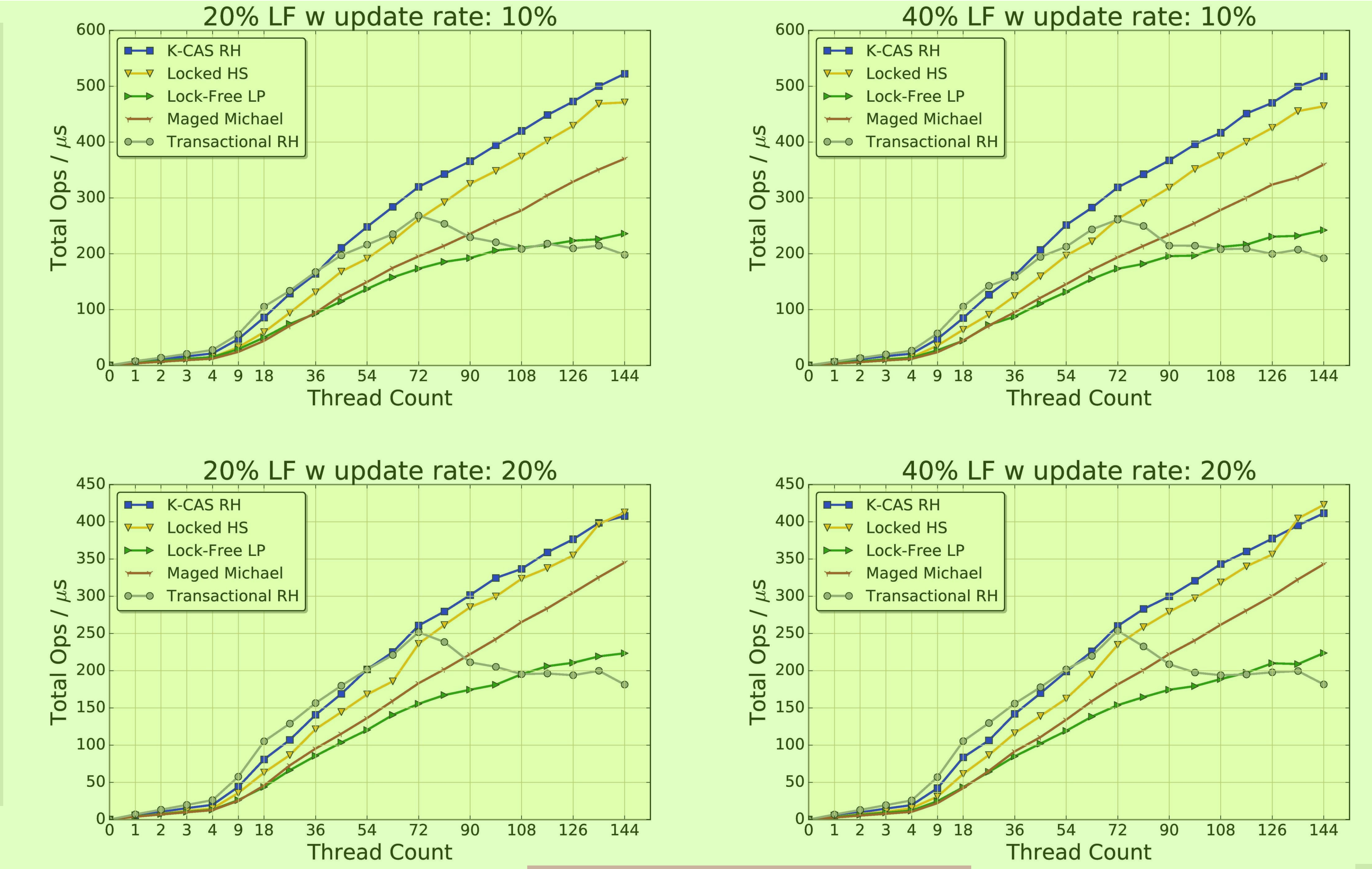
- Hopscotch Hashing [Herlihy, Shavit, Tzafrir; 2008]: Flattened separate chaining.
- Lock-Free Linear Probing [Nielsen, Karlsson; 2016] : State simplified Purcell, Harris Table.
- Separate Chaining [Maged Michael; 2003]: Per-bucket lock-free linked lists.

Tables - Explainer

- Hopscotch Hashing [Herlihy, Shavit, Tzafrir; 2008]: Flattened separate chaining.
- Lock-Free Linear Probing [Nielsen, Karlsson; 2016] : State simplified Purcell, Harris Table.
- Separate Chaining [Maged Michael; 2003]: Per-bucket lock-free linked lists.
- Lock-Elision Robin Hood. Serial algorithm with hardware transactional lock-elision wrapper
- *K-CAS* Robin Hood Hash. *K-CAS* with sharded timestamps.

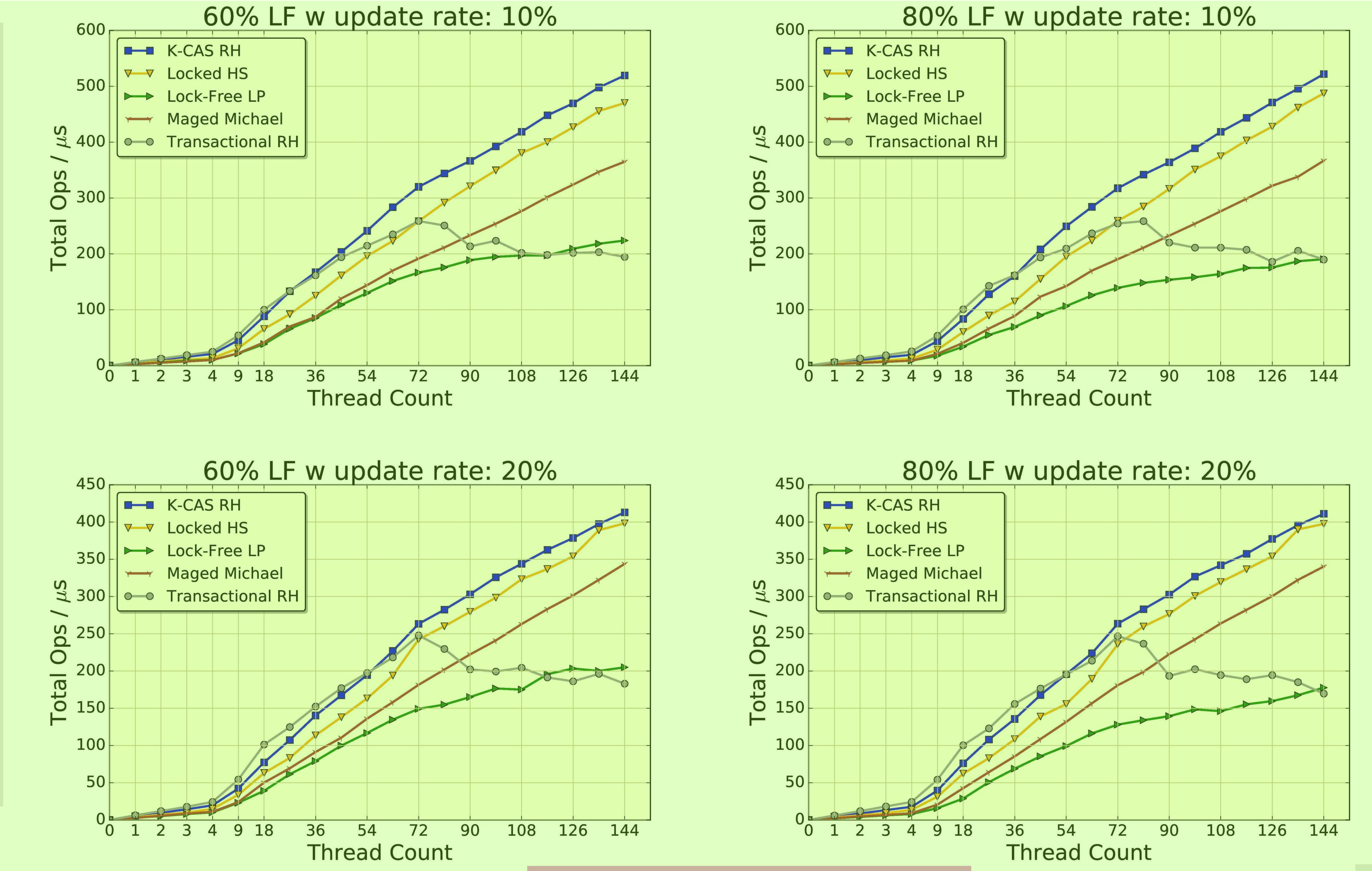
Performance 20%/40%

Total Operations per microsecond.



Number of threads.

Performance 60%/80%



Total Operations per microsecond.

Number of threads.

Performance Summary

Performance Summary

Robin Hood scales best in almost all workloads.
Otherwise very competitive with Hopscotch Hashing.

Comfortably ahead (10%) with 10% update load.
More competitive (5%) with 20% updates.

Performance Summary

Robin Hood scales best in almost all workloads.
Otherwise very competitive with Hopscotch Hashing.

Comfortably ahead (10%) with 10% update load.
More competitive (5%) with 20% updates.

Robin Hood dominates other concurrent hash tables.
Gap narrows during Hyperthreading.

Transactional Robin Hood scales very strongly until Hyperthreading. Then it dies and never recovers.

Conclusion

- First linearisable concurrent variant of Robin Hood Hashing.
- Strong application of new *K-CAS* developments.
- Competitive performance compared to state of the art concurrent hash tables.

Future Work

- Extended Robin Hood work (different timestamp encodings/placements, cache aware, vectorised, various lock-based solutions)
- Yahoo benchmark (YCSB)

Thank you!

Questions and Comments?