

A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs

Miguel Areias and Ricardo Rocha
CRACS & INESC-TEC LA
Faculty of Sciences, University of Porto, Portugal
miguel-areias@dcc.fc.up.pt *ricroc@dcc.fc.up.pt*

Tabling in Prolog Systems

- **Tabling** is an implementation technique that overcomes some of the limitations of Prolog resolution:
 - ◆ Tabled subgoals are evaluated by storing their answers in an appropriate data space, called the **table space**
 - ◆ Repeated calls to tabled subgoals are resolved by **consuming** the answers already stored in the table instead of **being re-evaluated** against the program clauses.

Tabling in Prolog Systems

- **Tabling** is an implementation technique that overcomes some of the limitations of Prolog resolution:
 - ◆ Tabled subgoals are evaluated by storing their answers in an appropriate data space, called the **table space**
 - ◆ Repeated calls to tabled subgoals are resolved by **consuming** the answers already stored in the table instead of **being re-evaluated** against the program clauses.

- Implementations of **Tabling** are currently available in systems like:
 - ◆ XSB Prolog, **Yap Prolog**, B-Prolog, ALS-Prolog, Mercury, Ciao Prolog.

Tabling in Prolog Systems

- **Tabling** is an implementation technique that overcomes some of the limitations of Prolog resolution:
 - ◆ Tabled subgoals are evaluated by storing their answers in an appropriate data space, called the **table space**
 - ◆ Repeated calls to tabled subgoals are resolved by **consuming** the answers already stored in the table instead of **being re-evaluated** against the program clauses.

- Implementations of **Tabling** are currently available in systems like:
 - ◆ XSB Prolog, **Yap Prolog**, B-Prolog, ALS-Prolog, Mercury, Ciao Prolog.

- **Multithreading** combined with **Tabling**:
 - ◆ XSB Prolog
 - ◆ **Yap Prolog [ICLP 2012]**

Table Space - Example

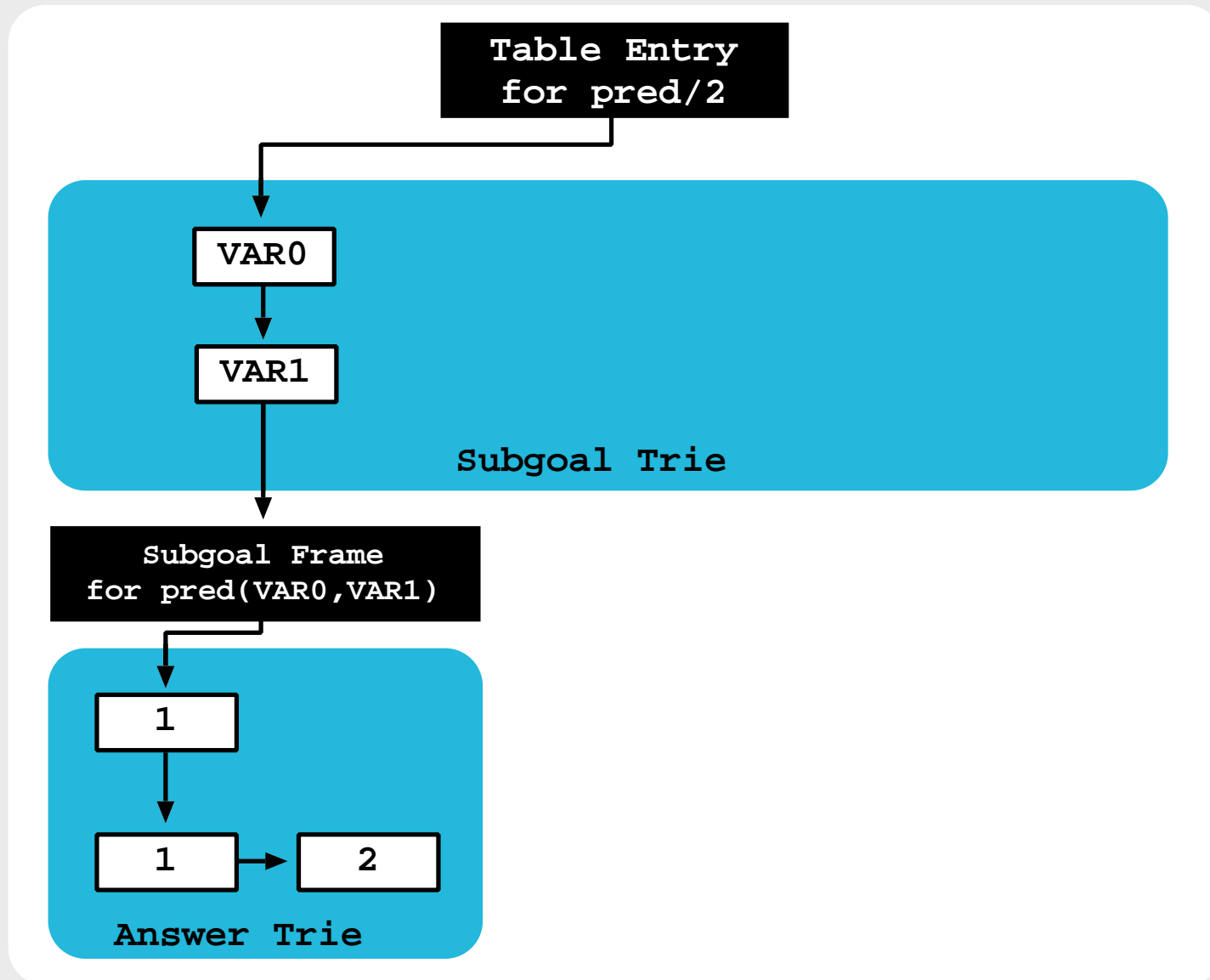


Table Space - Example

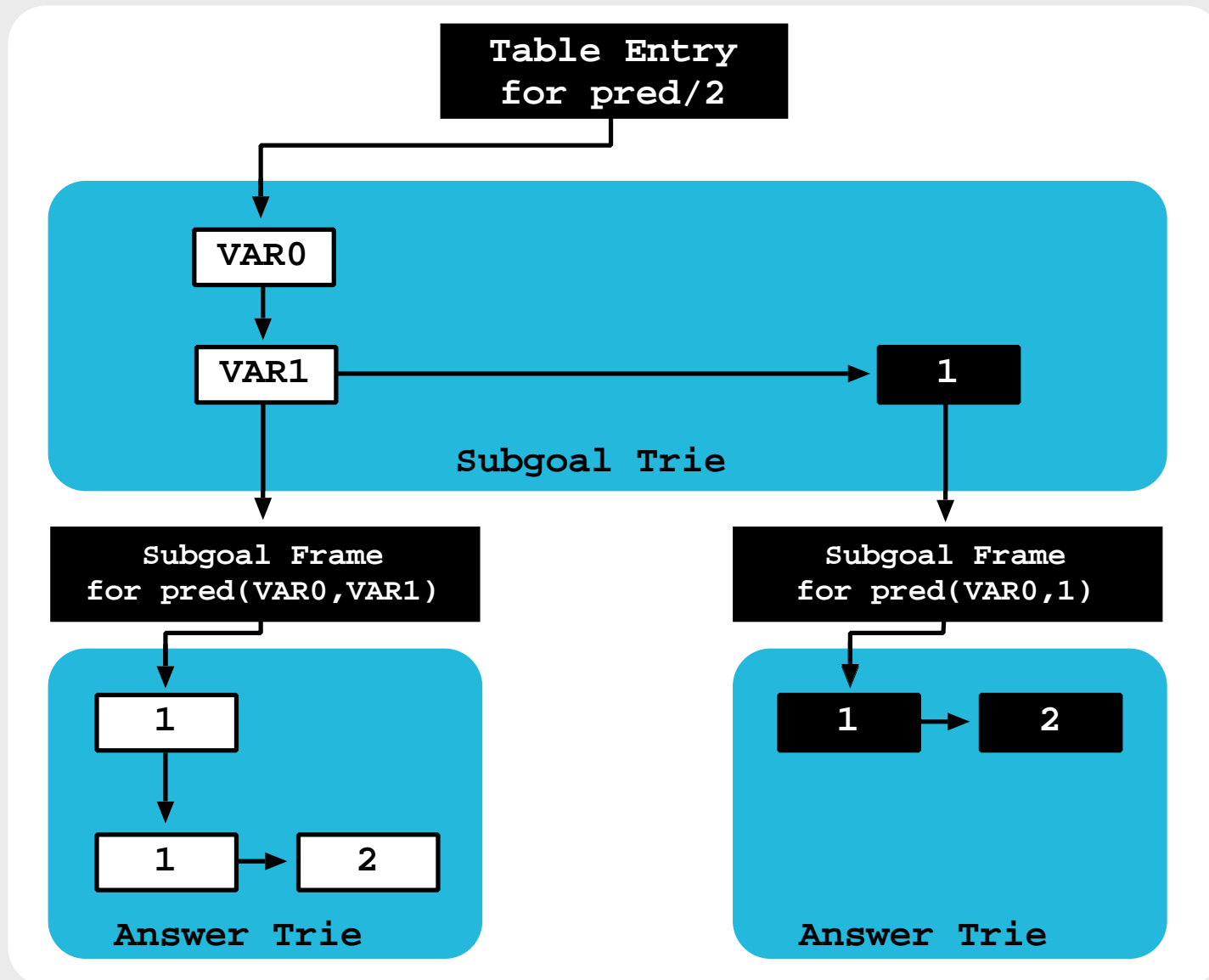


Table Space - Trie Level Internals

- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.



Table Space - Trie Level Internals

- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.

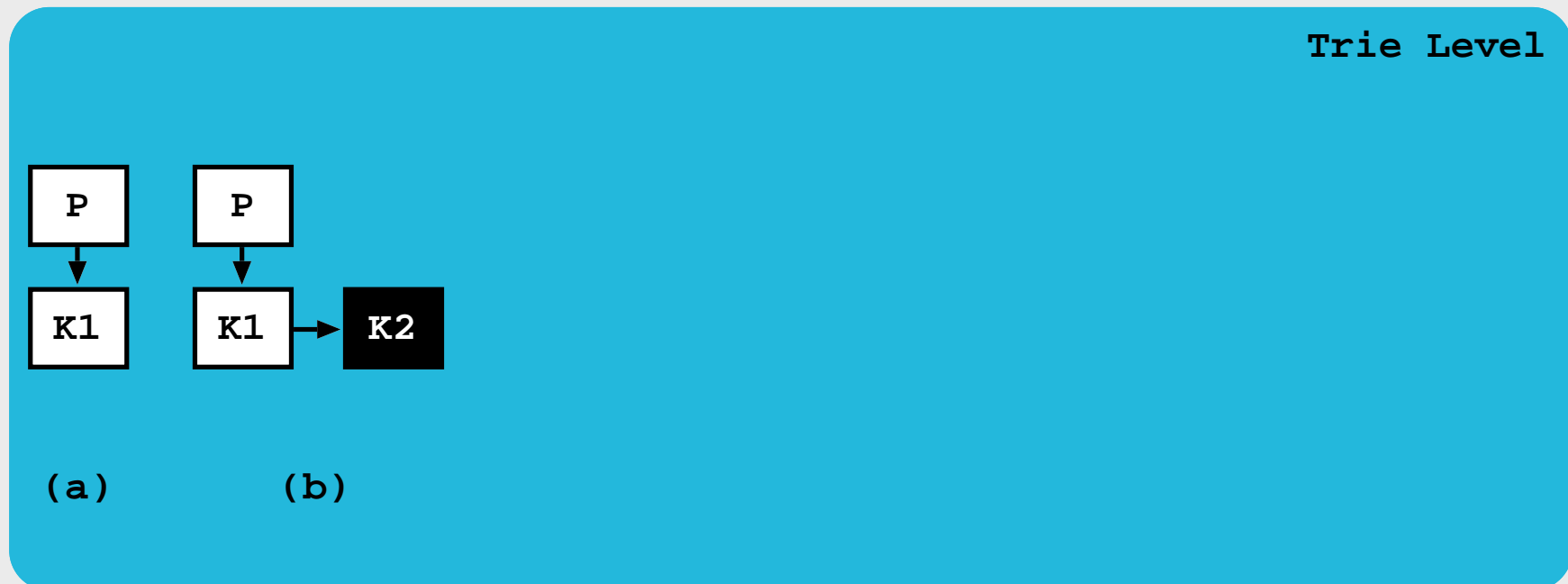


Table Space - Trie Level Internals

- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.

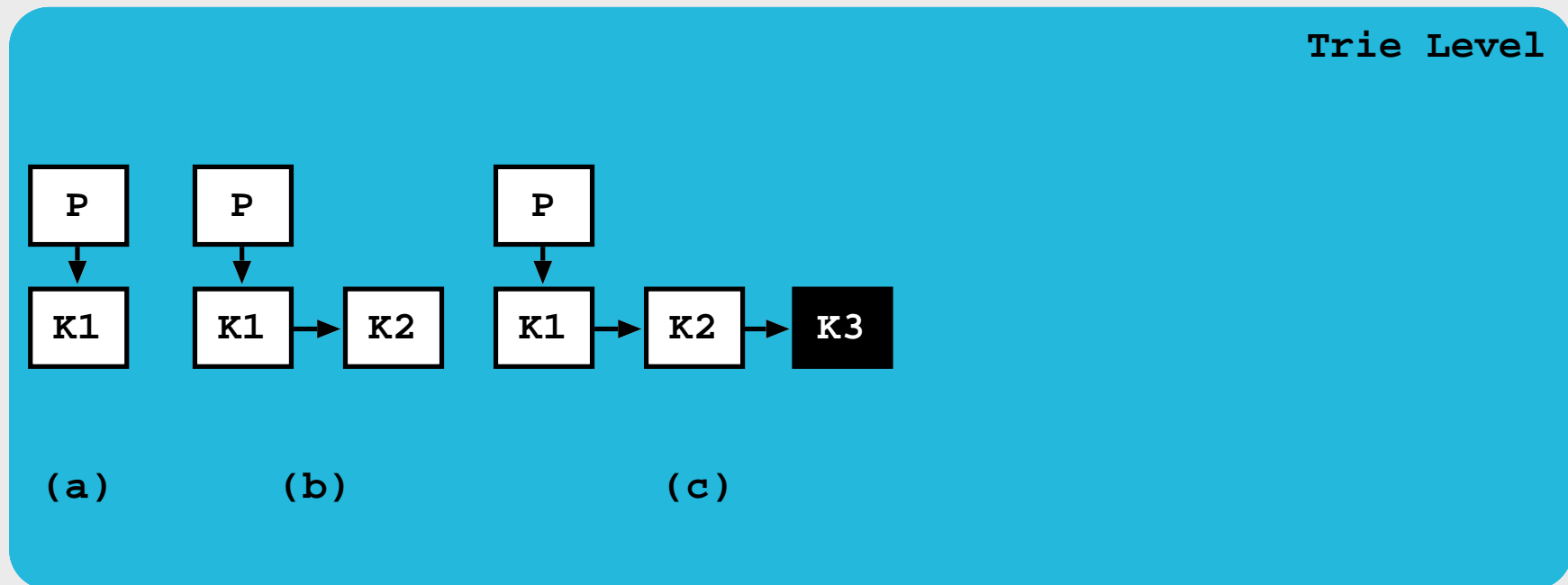


Table Space - Trie Level Internals

- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.

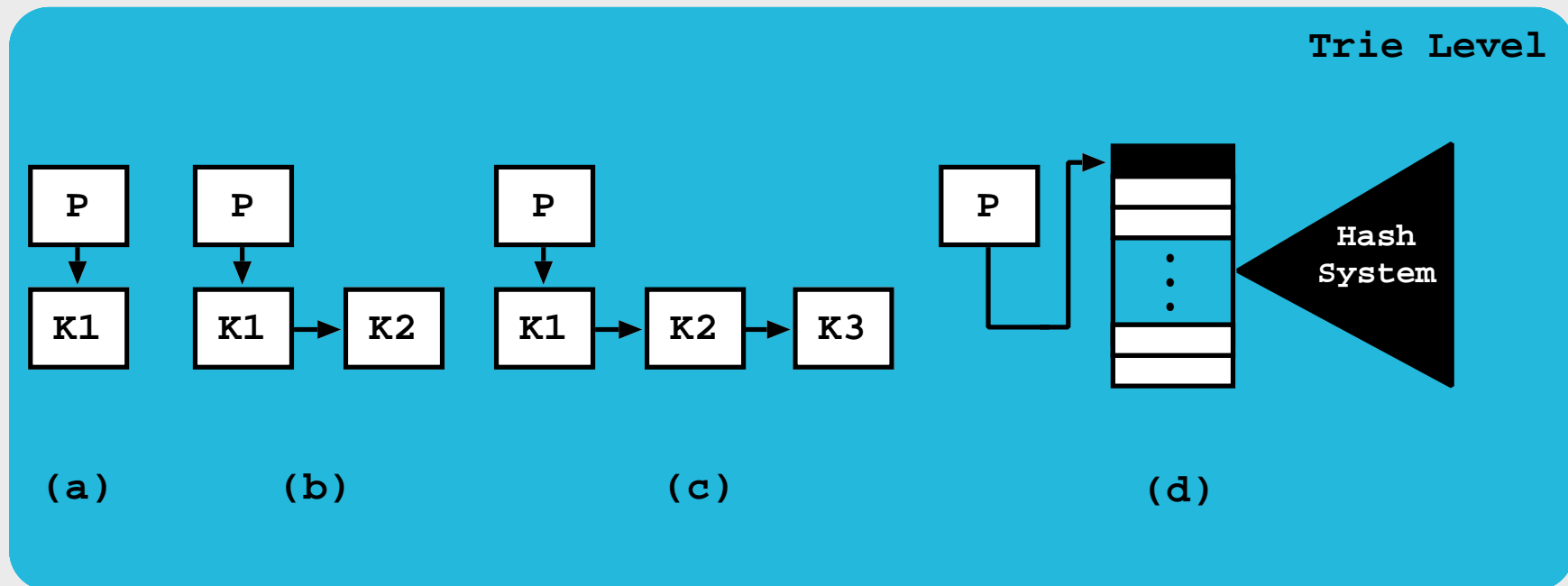


Table Space - Trie Level Internals

- A **trie level** is defined by a **parent (P)** node and at least one **child (K)** node.
- Only **lookup** and **insert** operations are executed.
- **Insertion** of new nodes is done in a **chain**, until a **threshold** is achieved and afterwards a **hashing system** is included in the trie level.

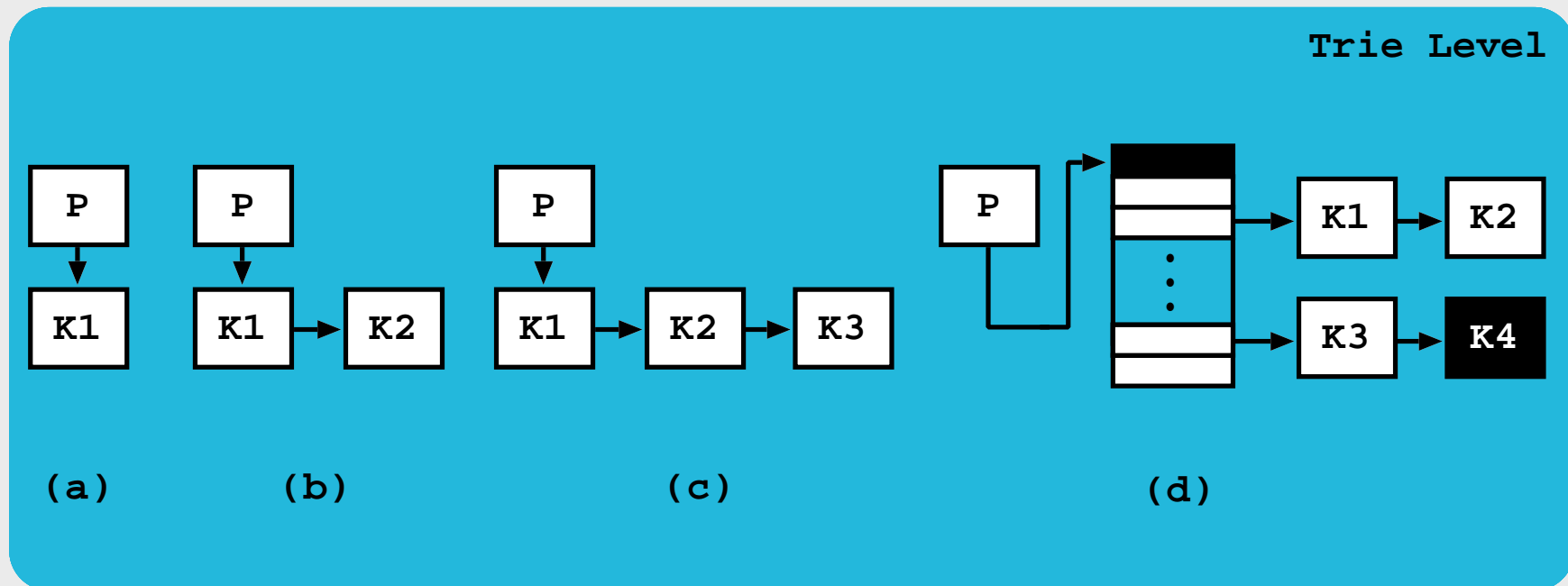
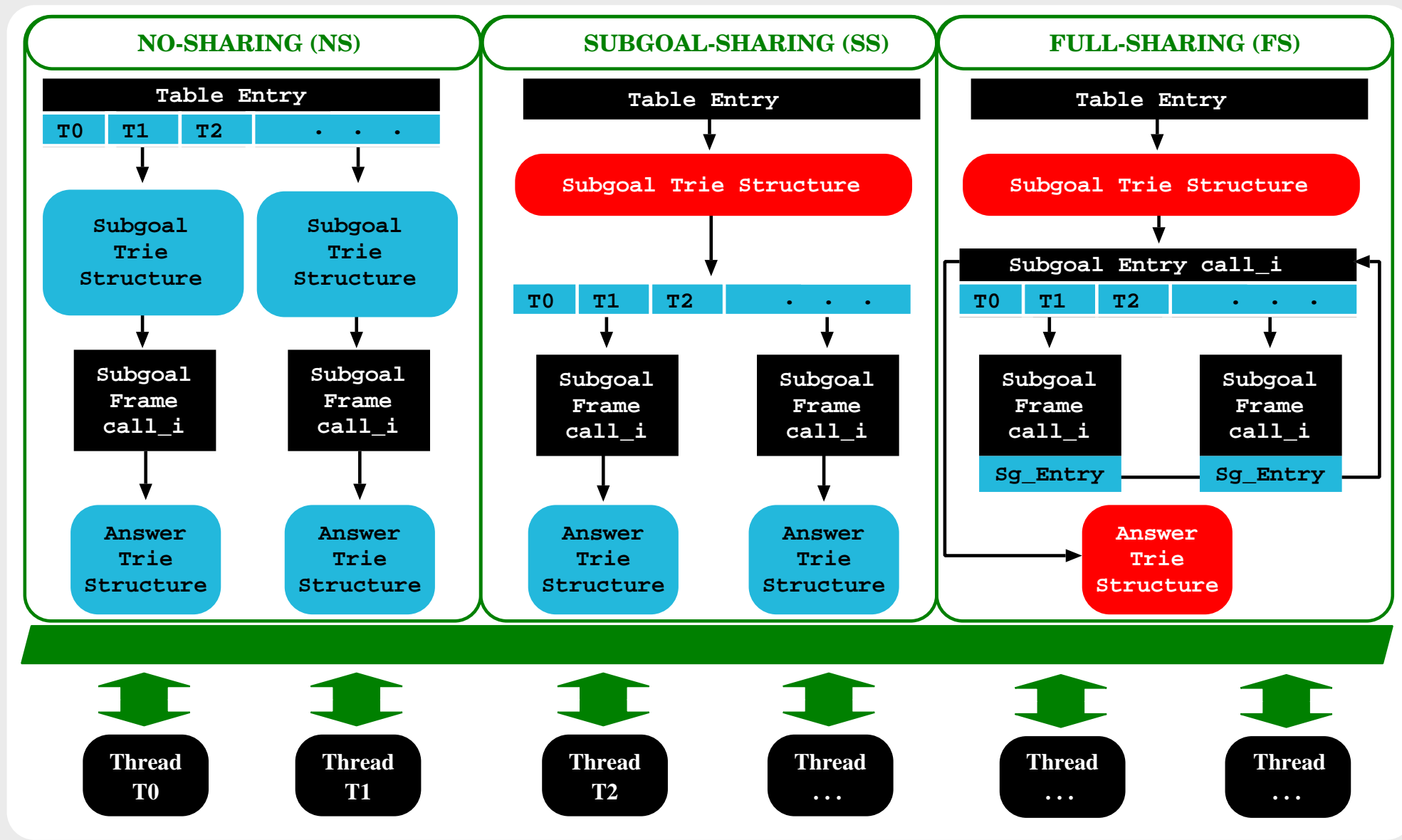


Table Space - Multithreaded Designs



Our Approach - Motivation

- **Until now** to deal with **concurrency** we used the **following mechanisms**:
 - ◆ Standard Locking [**Euro-Par 2004**] and Try Locking [**ICLP 2012**]
 - ◆ Different lock locations [**ICPADS 2012**]:
 - * Lock Field per trie node
 - * Global array of lock entries.
 - ◆ Lock-Free using **CAS (Compare-and-Swap)** operations [**PADL 2014**].

Our Approach - Motivation

- **Until now** to deal with **concurrency** we used the **following mechanisms**:
 - ◆ Standard Locking [**Euro-Par 2004**] and Try Locking [**ICLP 2012**]
 - ◆ Different lock locations [**ICPADS 2012**]:
 - * Lock Field per trie node
 - * Global array of lock entries.
 - ◆ Lock-Free using **CAS (Compare-and-Swap)** operations [**PADL 2014**].

- **Problems** faced with **these approaches**:
 - ◆ Locking mechanisms suffer from:
 - * **Contention**
 - * **Convoying**
 - * **Priority inversion.**

Our Approach - Motivation

- **Until now** to deal with **concurrency** we used the **following mechanisms**:
 - ◆ Standard Locking [**Euro-Par 2004**] and Try Locking [**ICLP 2012**]
 - ◆ Different lock locations [**ICPADS 2012**]:
 - * Lock Field per trie node
 - * Global array of lock entries.
 - ◆ Lock-Free using **CAS (Compare-and-Swap)** operations [**PADL 2014**].

- **Problems** faced with **these approaches**:
 - ◆ Locking mechanisms suffer from:
 - * **Contention**
 - * **Convoying**
 - * **Priority inversion.**

 - ◆ The **bucket array of entries** inside the hashing system:
 - * **Low dispersion** of the synchronization points
 - * **False sharing** (memory cache secondary effects).

Our Approach - Motivation

- Use **lock-free linearizable objects** because they permit **greater concurrency** since **semantically consistent** (non-interfering) operations **may execute in parallel**.
- Take ideas from the several **lock-free designs** that already exist:
 - ◆ Shalev and Shavit **Split-Ordered Lists**
 - ◆ Prokopec **Concurrent Tries**
 - ◆ Cliff's **Non-Blocking Hash Tables**.

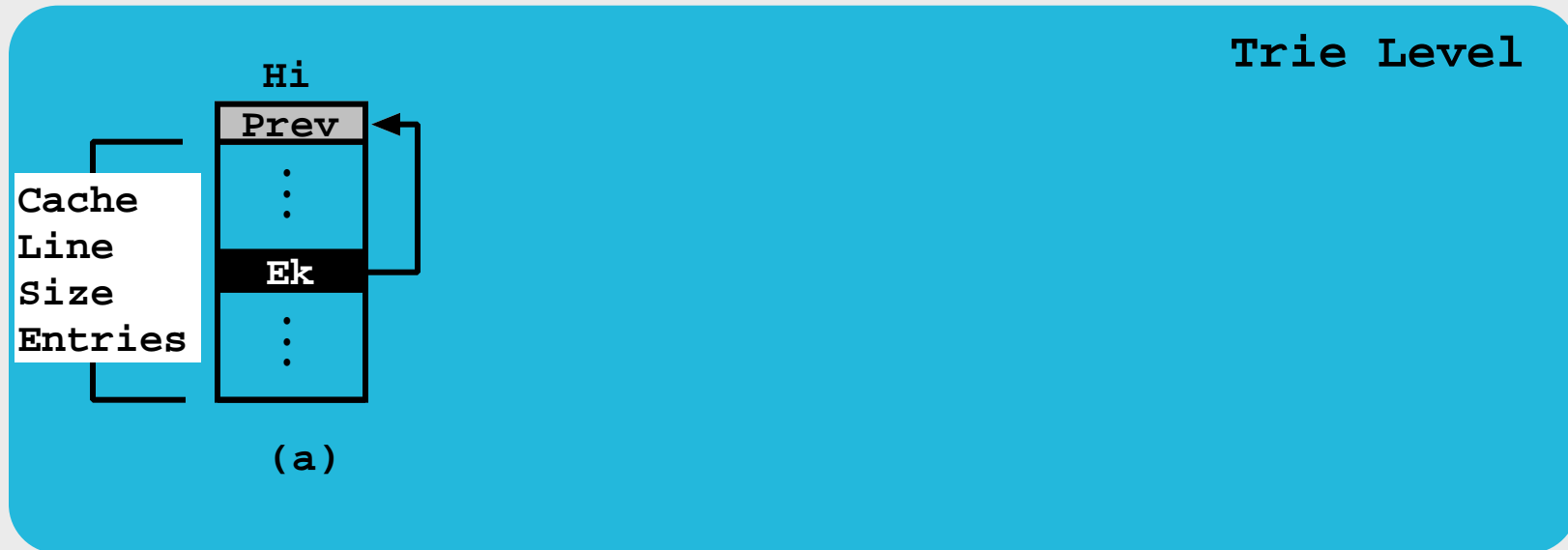
Our Approach - Motivation

- Use **lock-free linearizable objects** because they permit **greater concurrency** since **semantically consistent** (non-interfering) operations **may execute in parallel**.
- Take ideas from the several **lock-free designs** that already exist:
 - ◆ Shalev and Shavit **Split-Ordered Lists**
 - ◆ Prokopec **Concurrent Tries**
 - ◆ Cliff's **Non-Blocking Hash Tables**.
- But ... **none** of the designs is specifically **aimed** for an environment with the **characteristics** of our **tabling framework**.
 - ◆ Support for **concurrent deletion** of nodes **increases** the **complexity** of the designs.

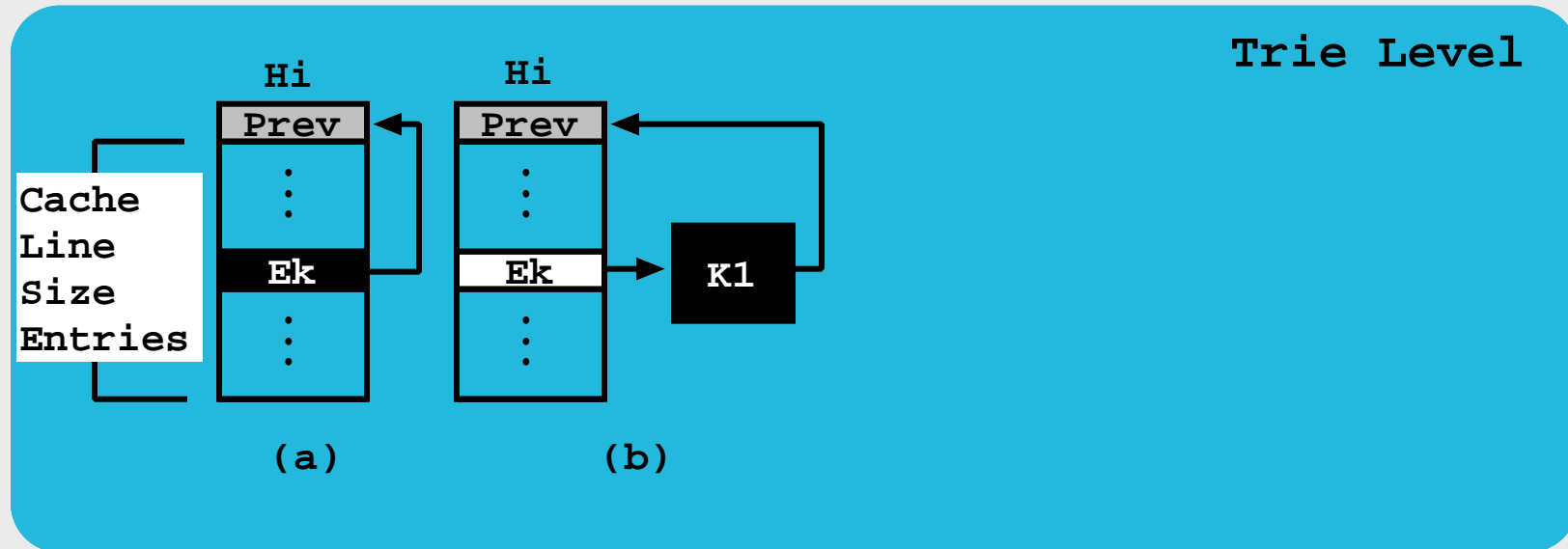
Our Approach - Motivation

- Use **lock-free linearizable objects** because they permit **greater concurrency** since **semantically consistent** (non-interfering) operations **may execute in parallel**.
- Take ideas from the several **lock-free designs** that already exist:
 - ◆ Shalev and Shavit **Split-Ordered Lists**
 - ◆ Prokopec **Concurrent Tries**
 - ◆ Cliff's **Non-Blocking Hash Tables**.
- But ... **none** of the designs is specifically **aimed** for an environment with the **characteristics** of our **tabling framework**.
 - ◆ Support for **concurrent deletion** of nodes **increases** the **complexity** of the designs.
- Create a new design (**LFHT Lock-Free Hash Tries**) that:
 - ◆ is as **efficient** as possible in **lookup and insert** operations
 - ◆ **minimizes** the **problems** associated with our **previous approaches**.

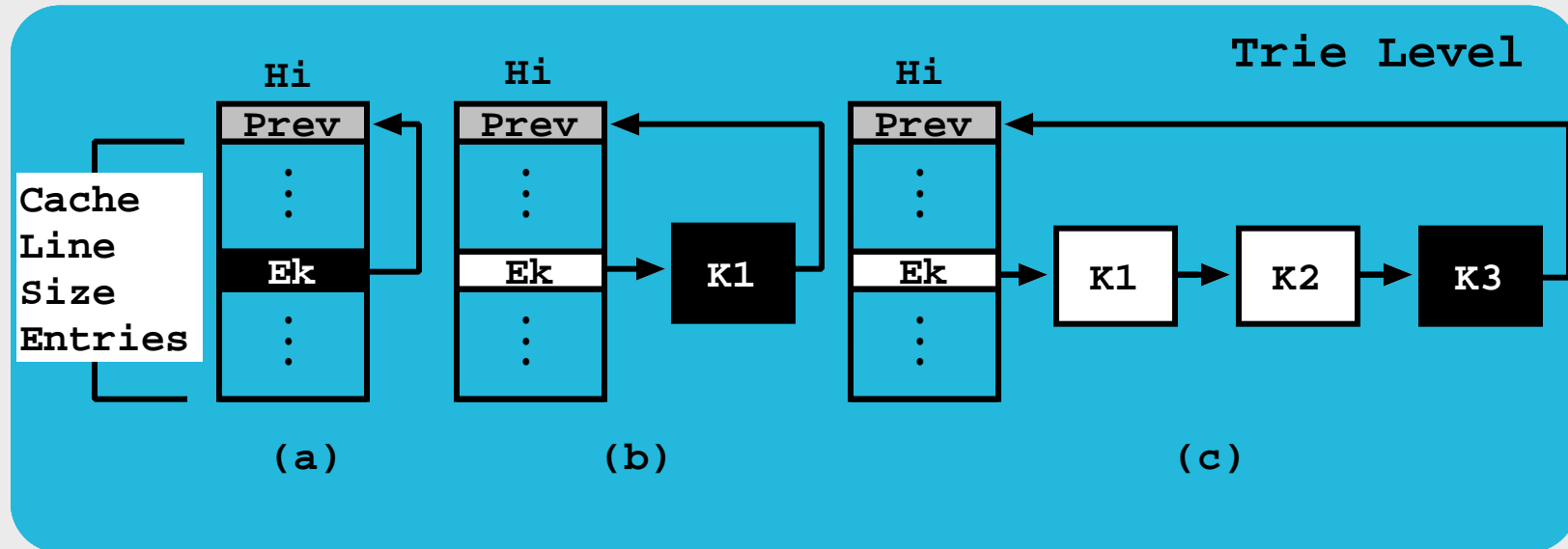
Lock-Free Hash Tries - Key Ideas



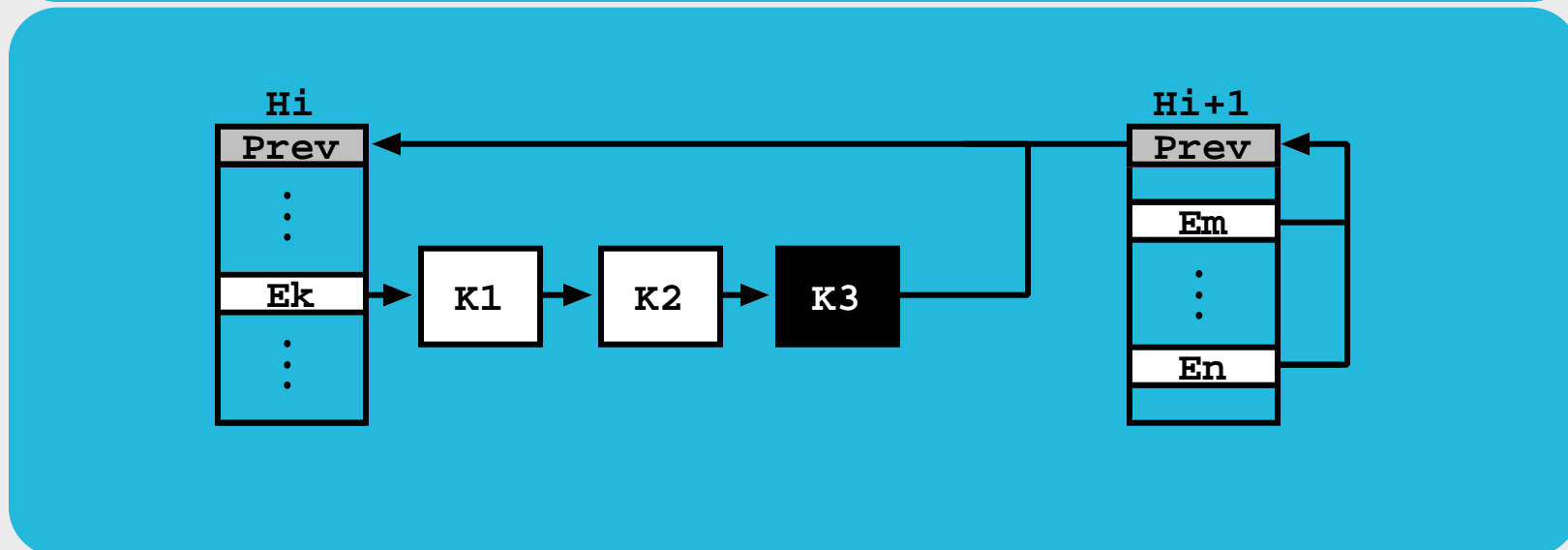
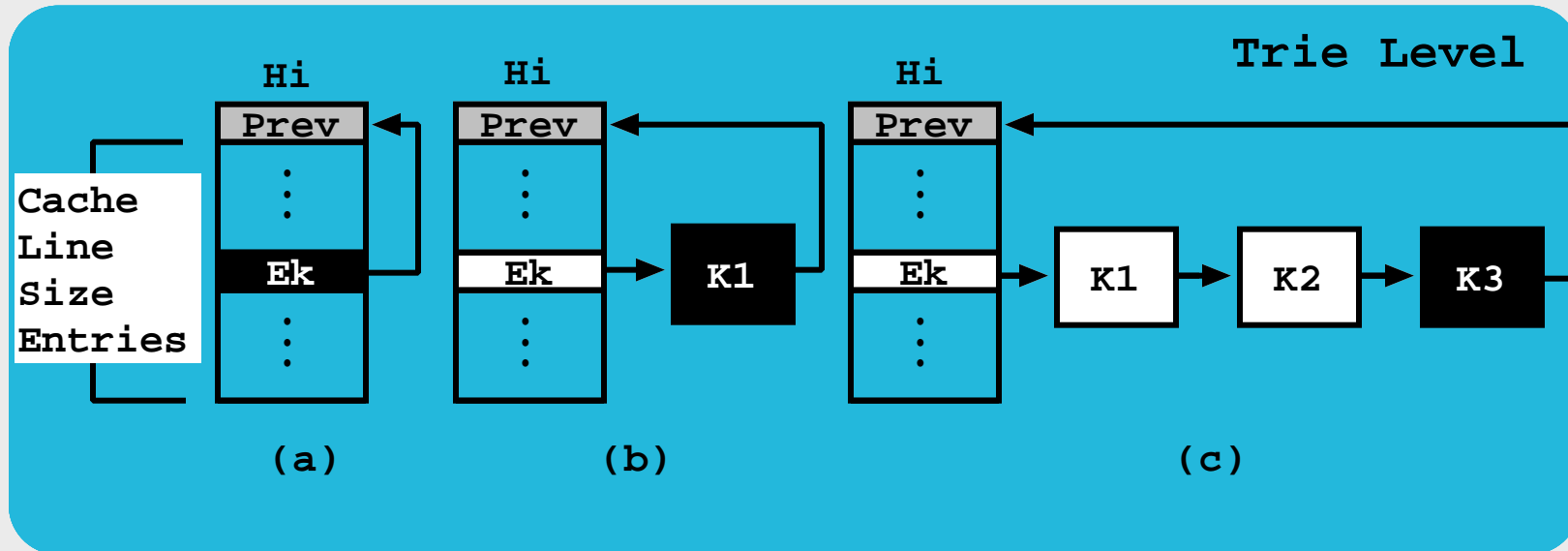
Lock-Free Hash Tries - Key Ideas



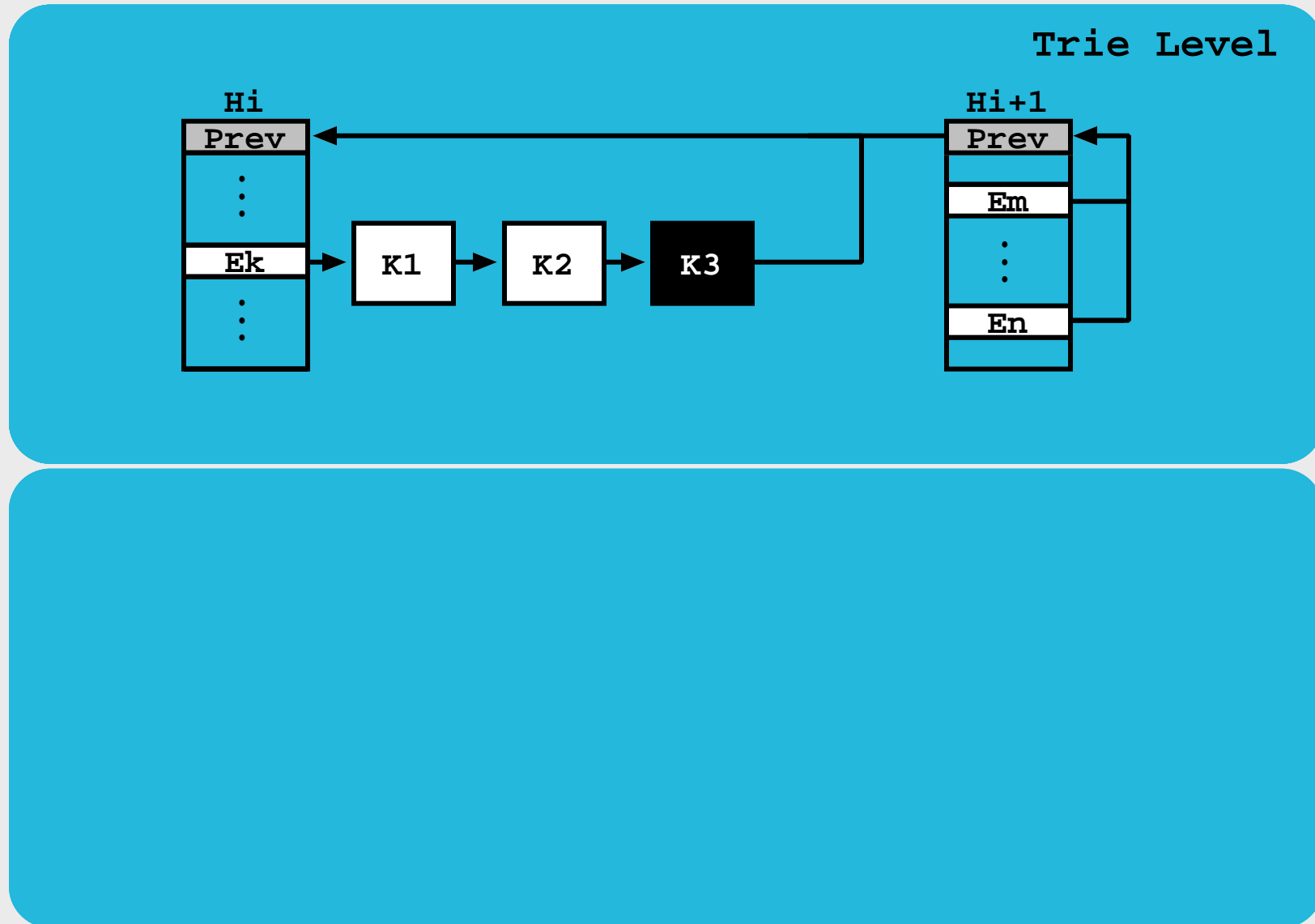
Lock-Free Hash Tries - Key Ideas



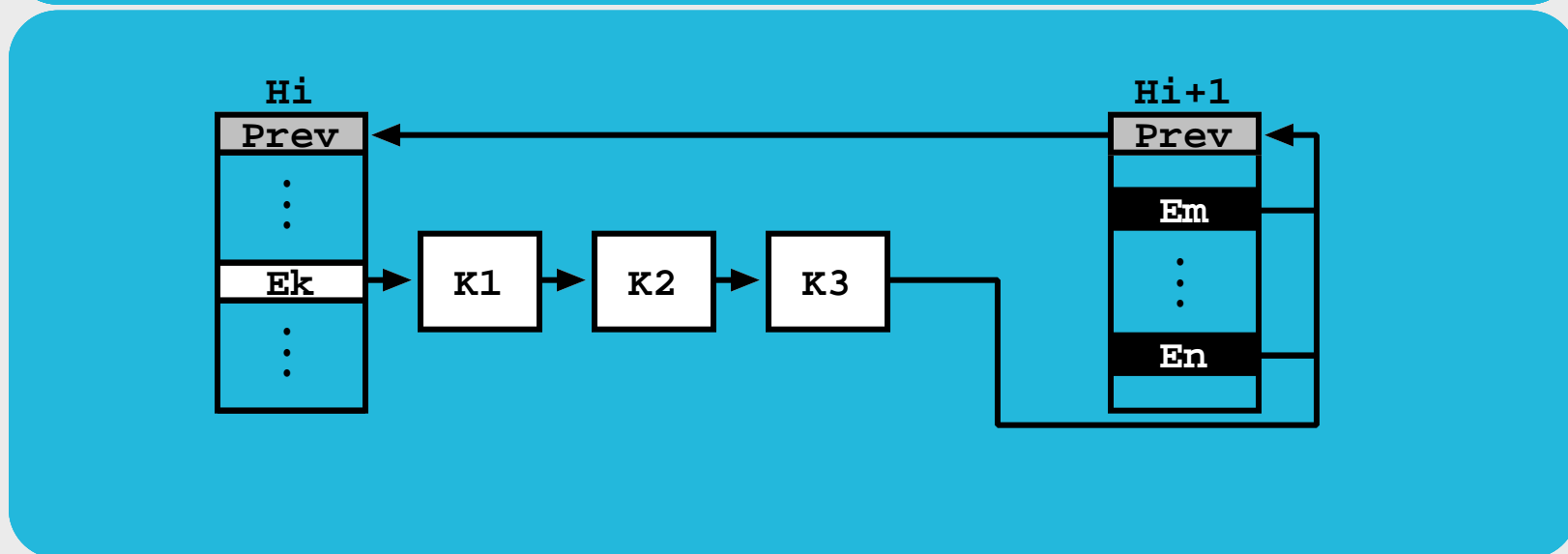
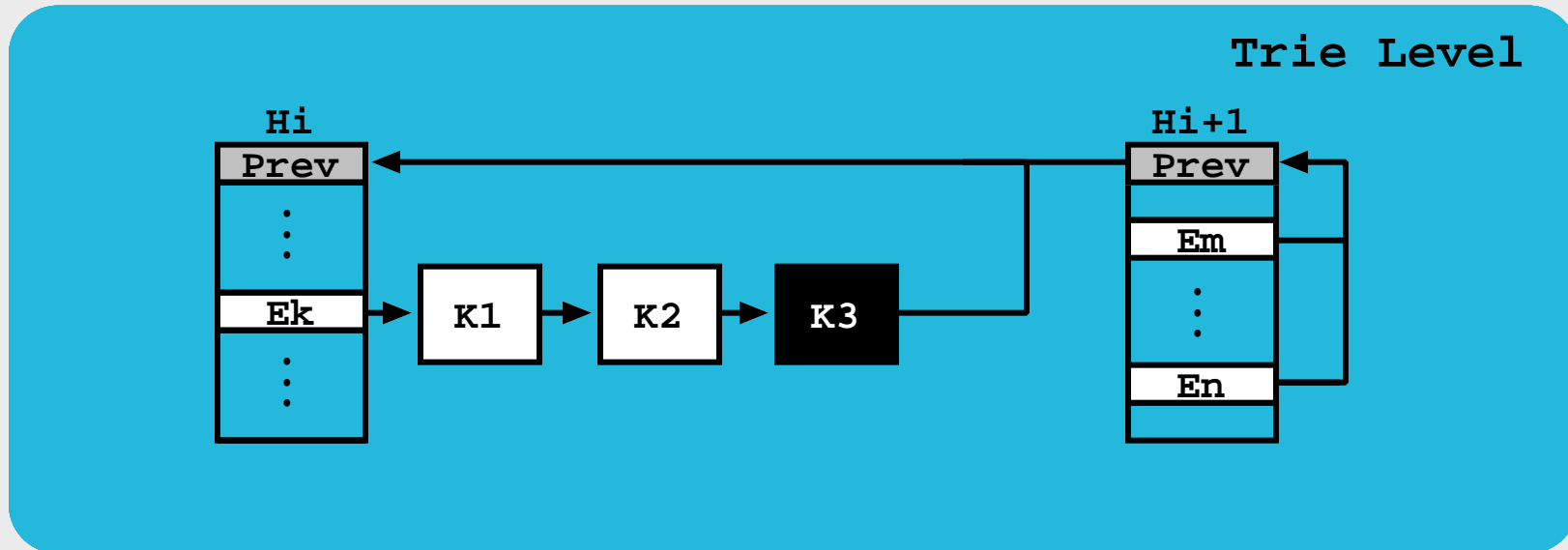
Lock-Free Hash Tries - Key Ideas



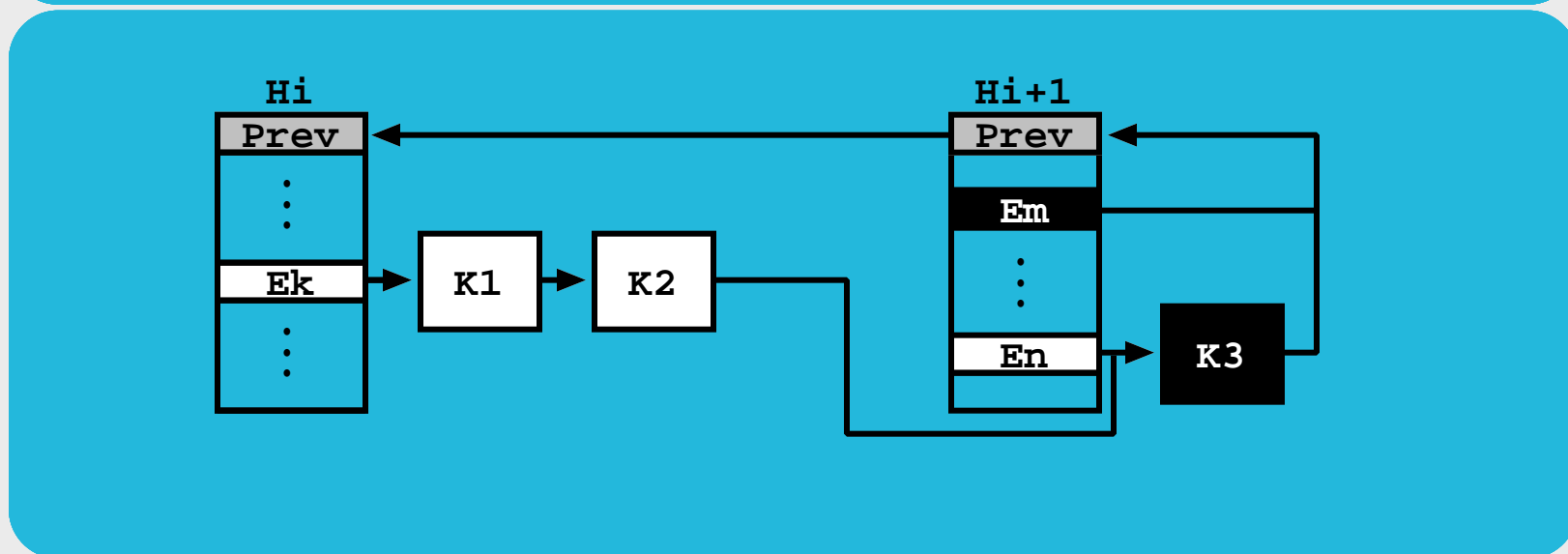
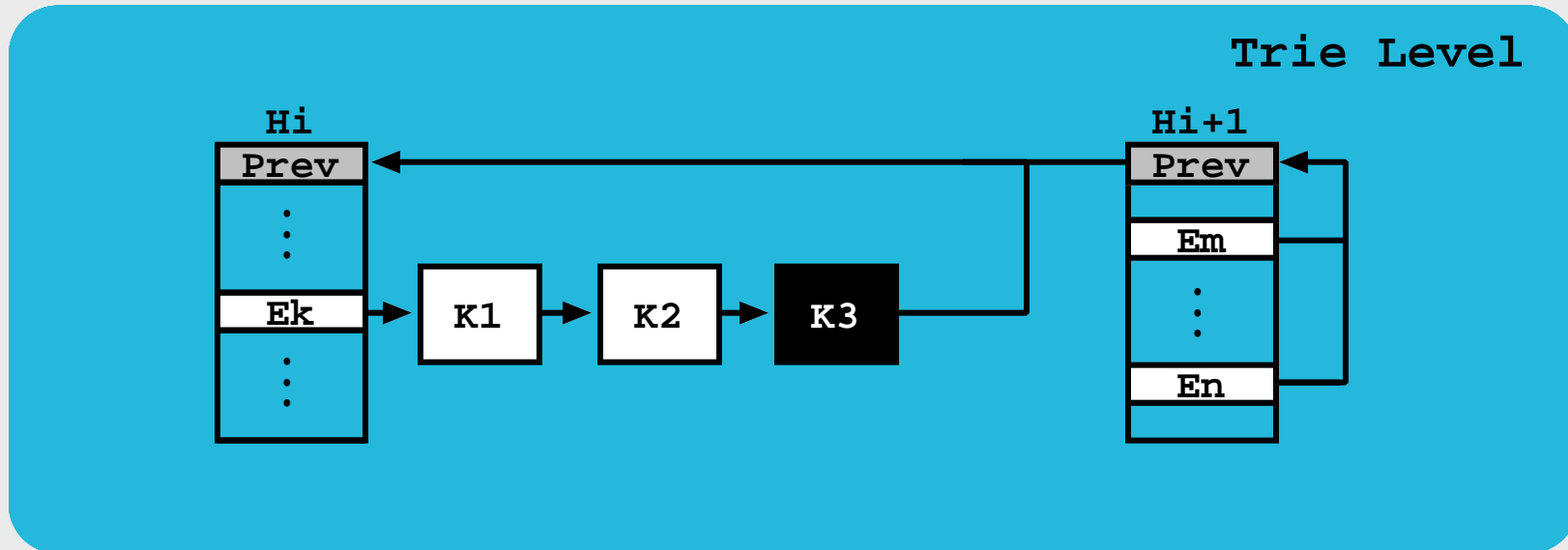
Lock-Free Hash Tries - Key Ideas



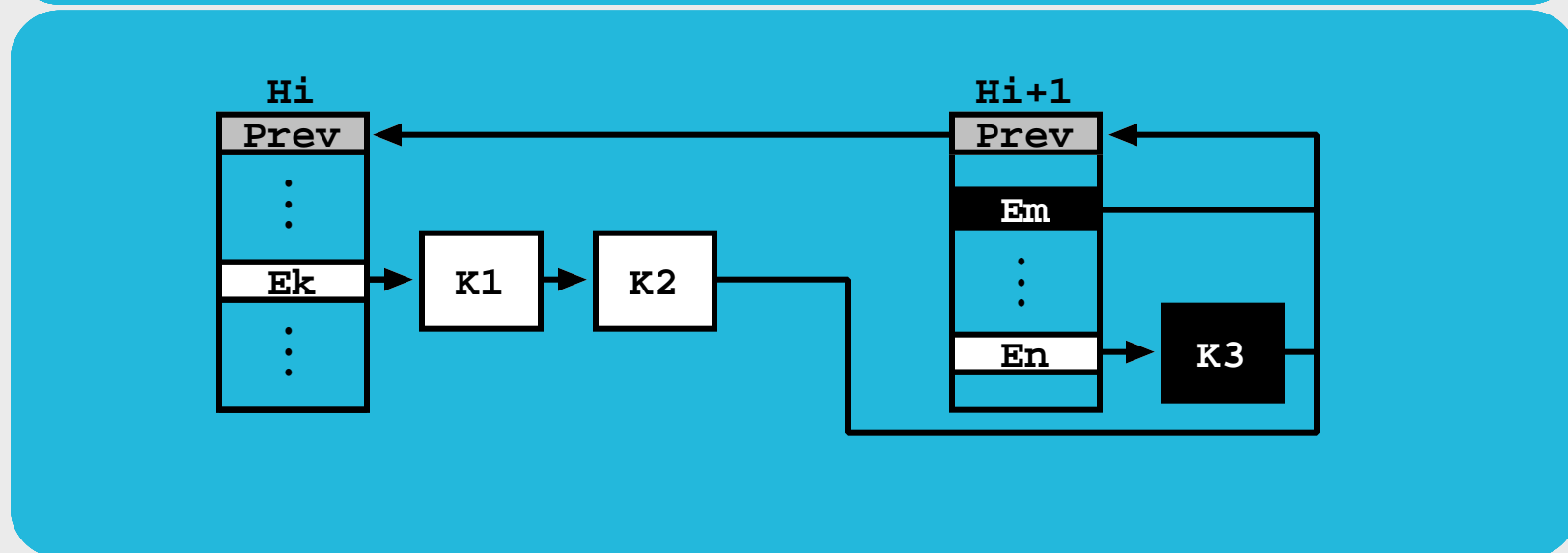
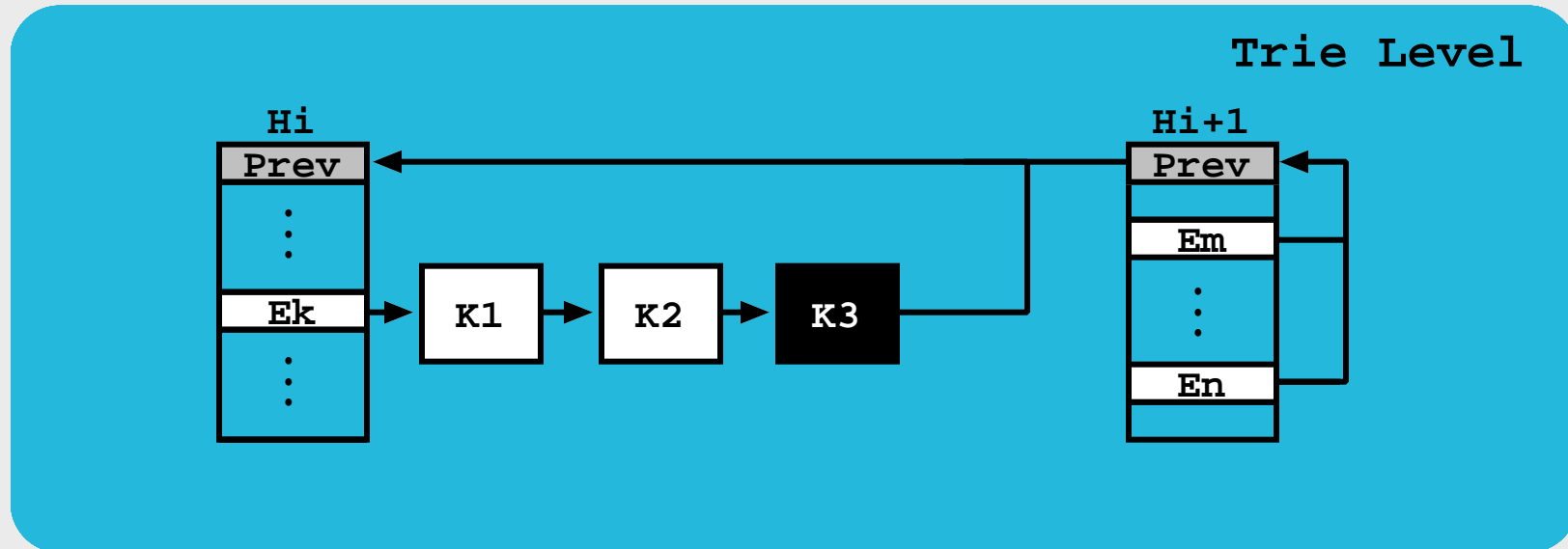
Lock-Free Hash Tries - Key Ideas



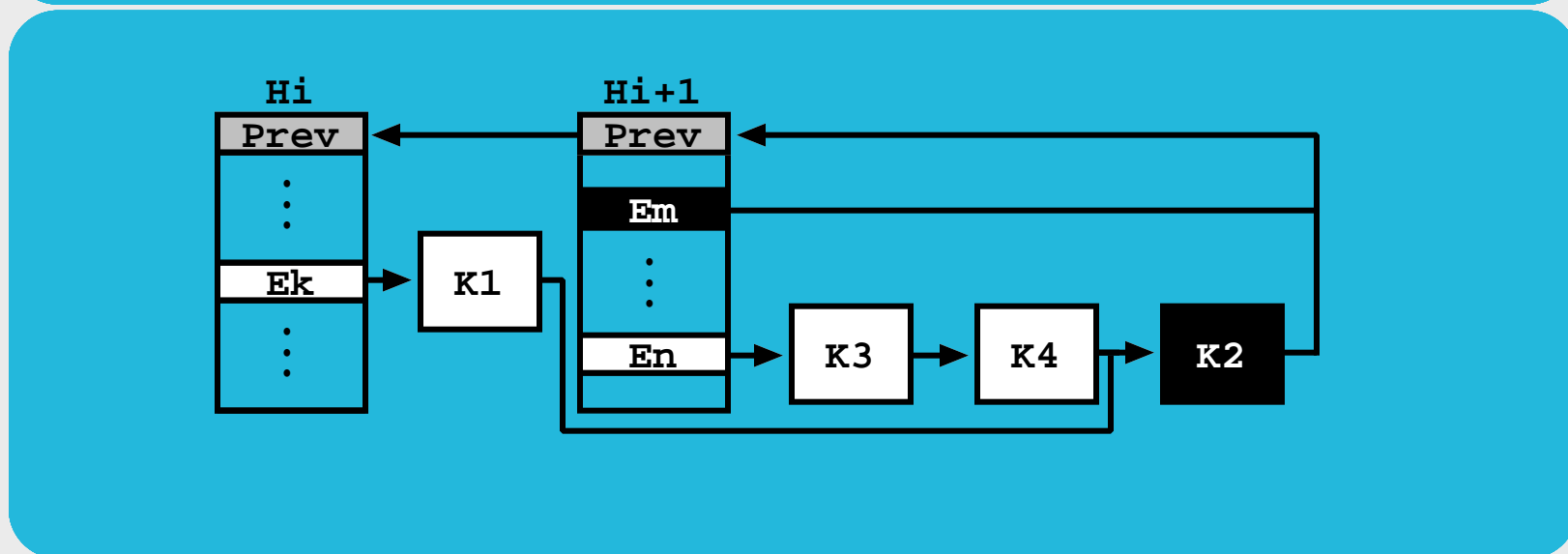
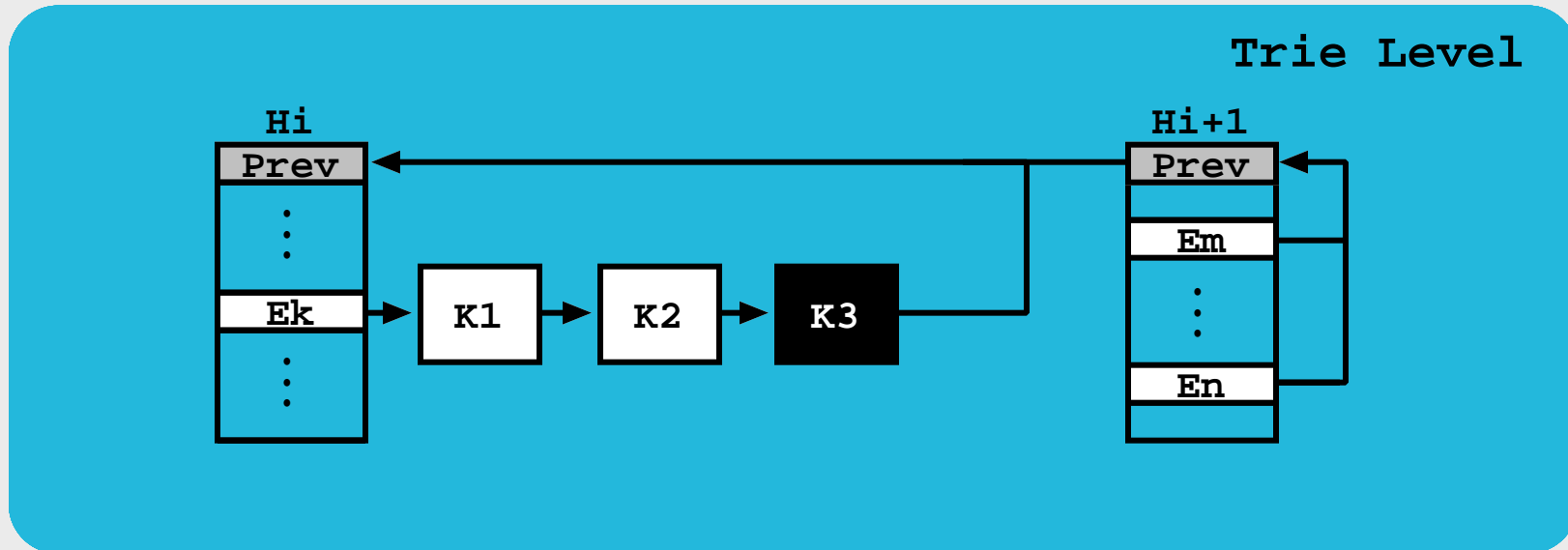
Lock-Free Hash Tries - Key Ideas



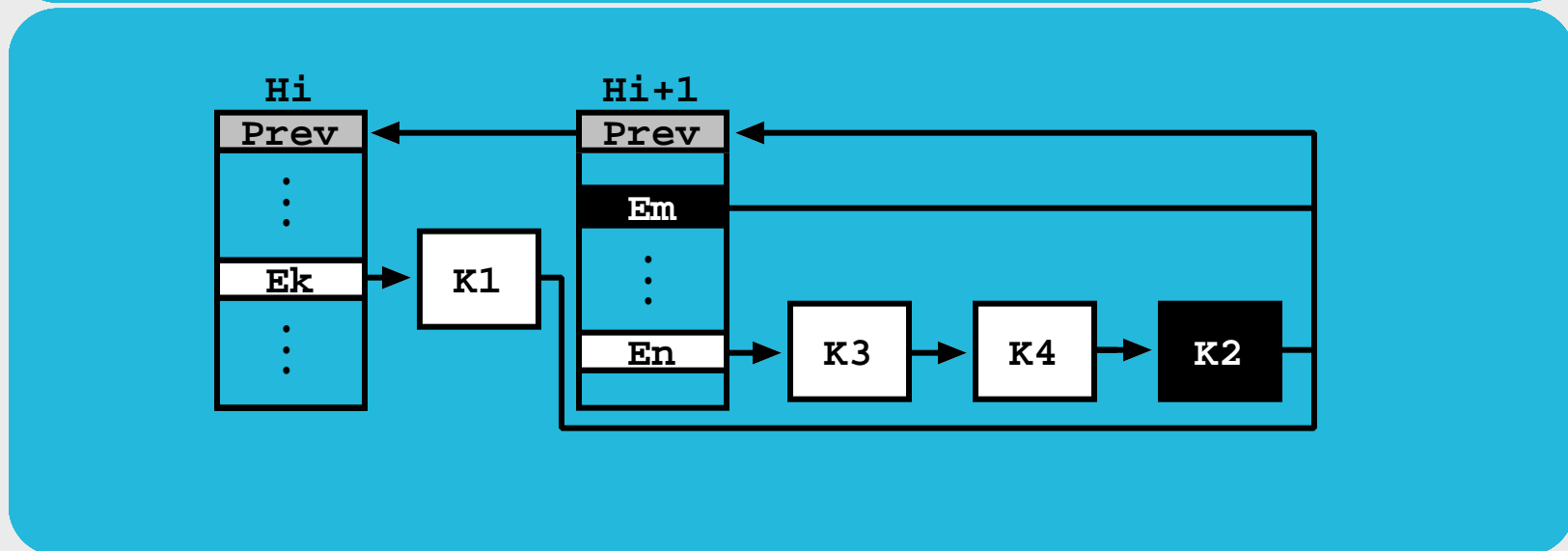
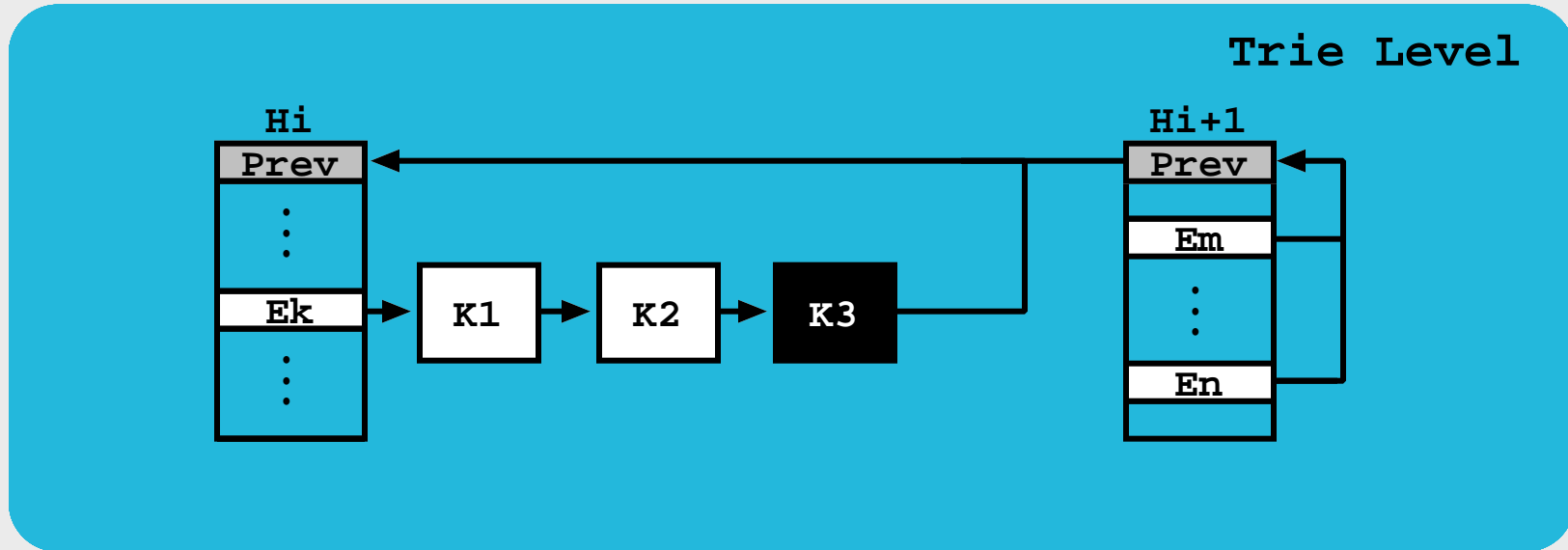
Lock-Free Hash Tries - Key Ideas



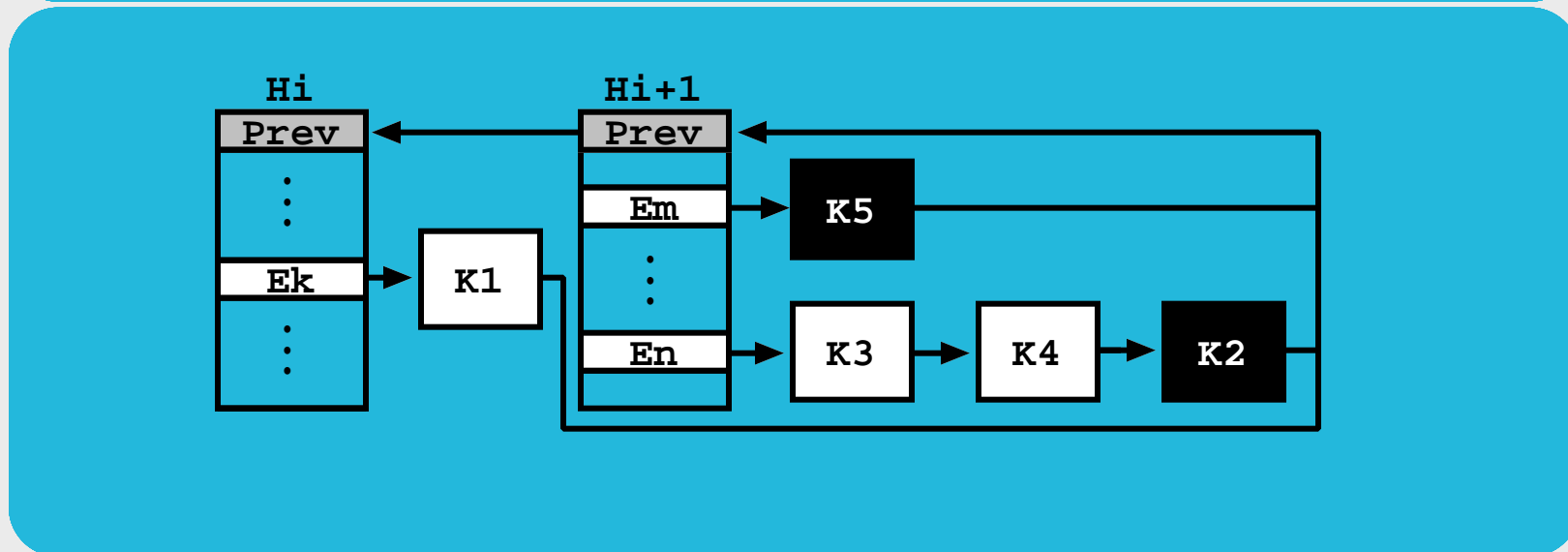
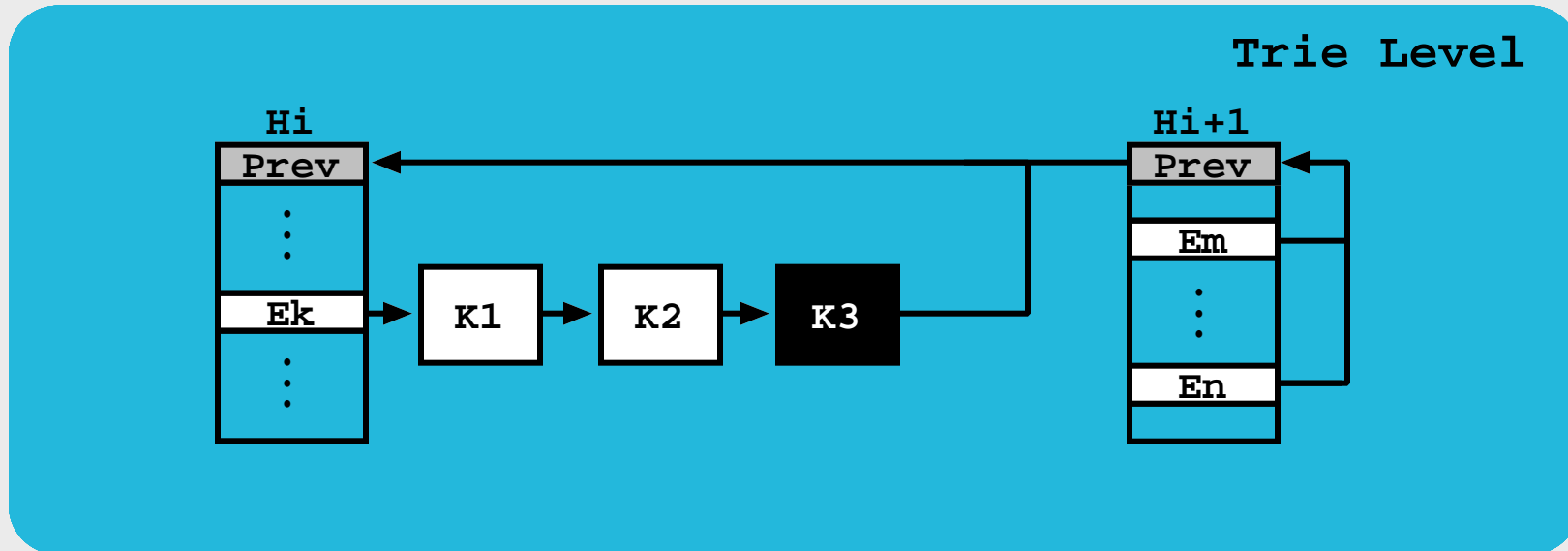
Lock-Free Hash Tries - Key Ideas



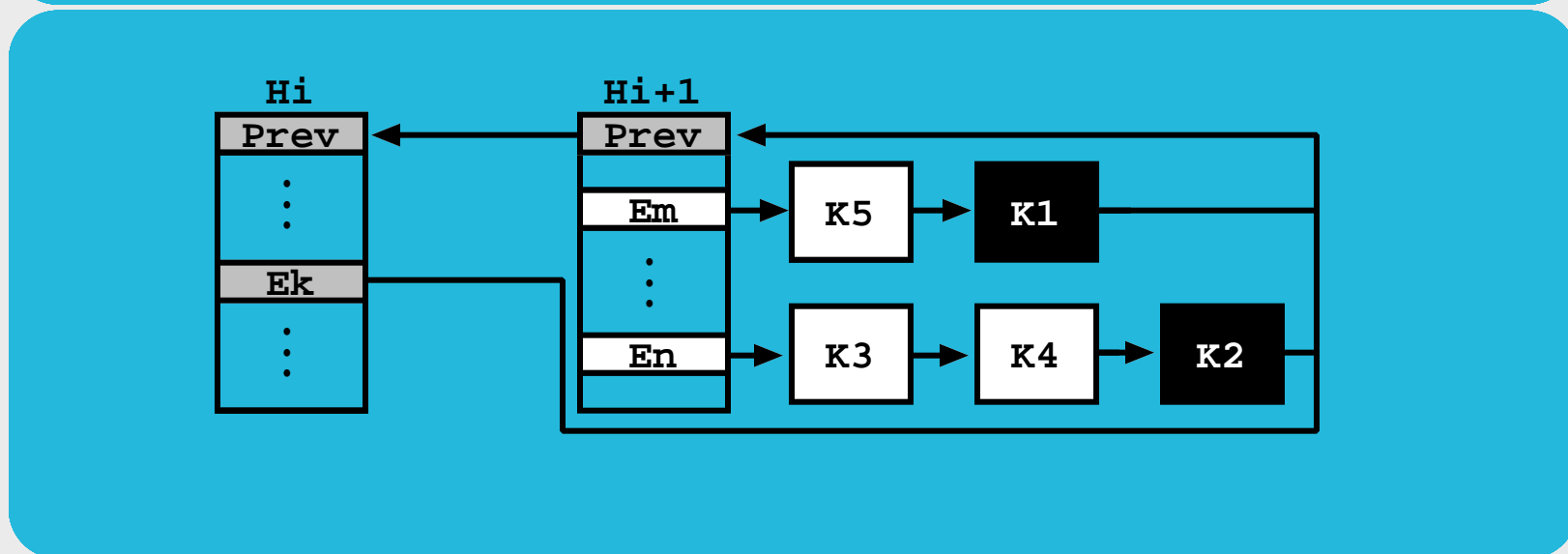
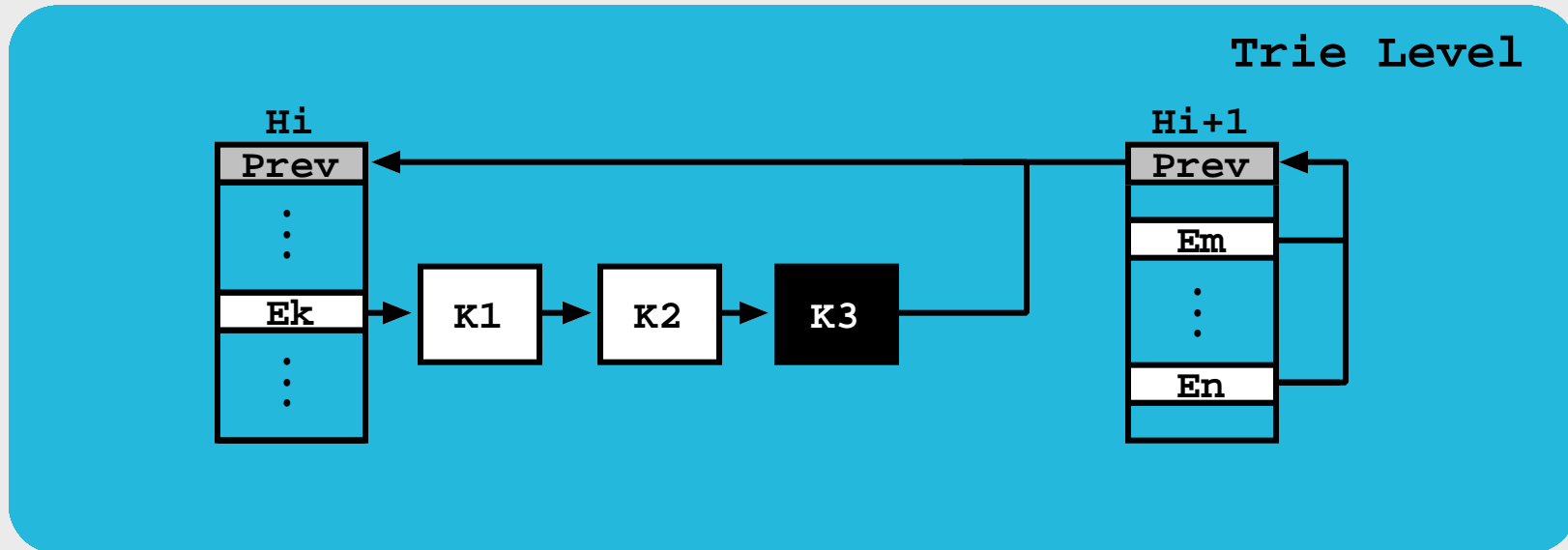
Lock-Free Hash Tries - Key Ideas



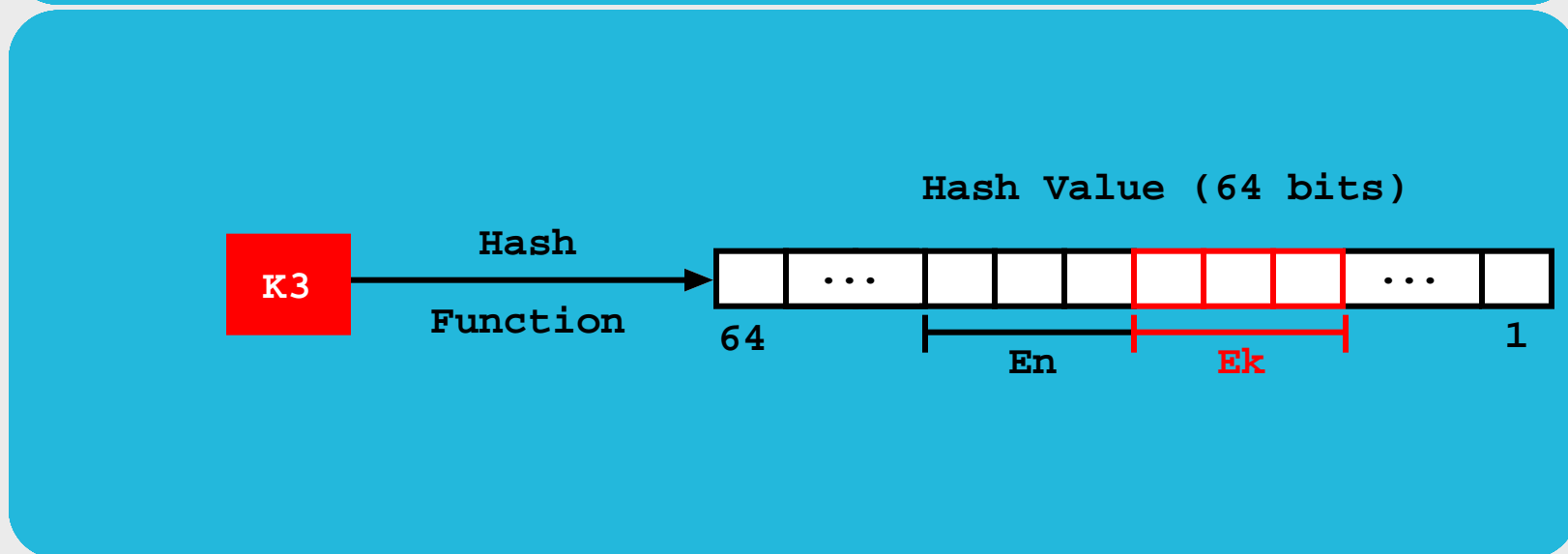
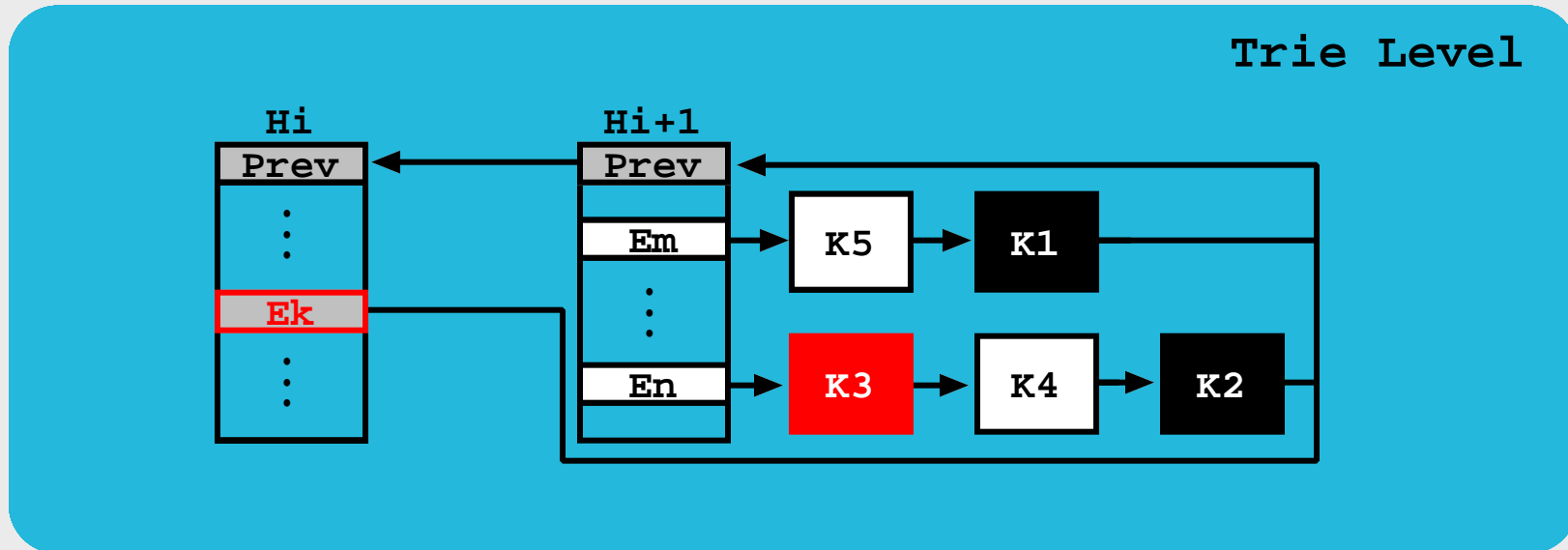
Lock-Free Hash Tries - Key Ideas



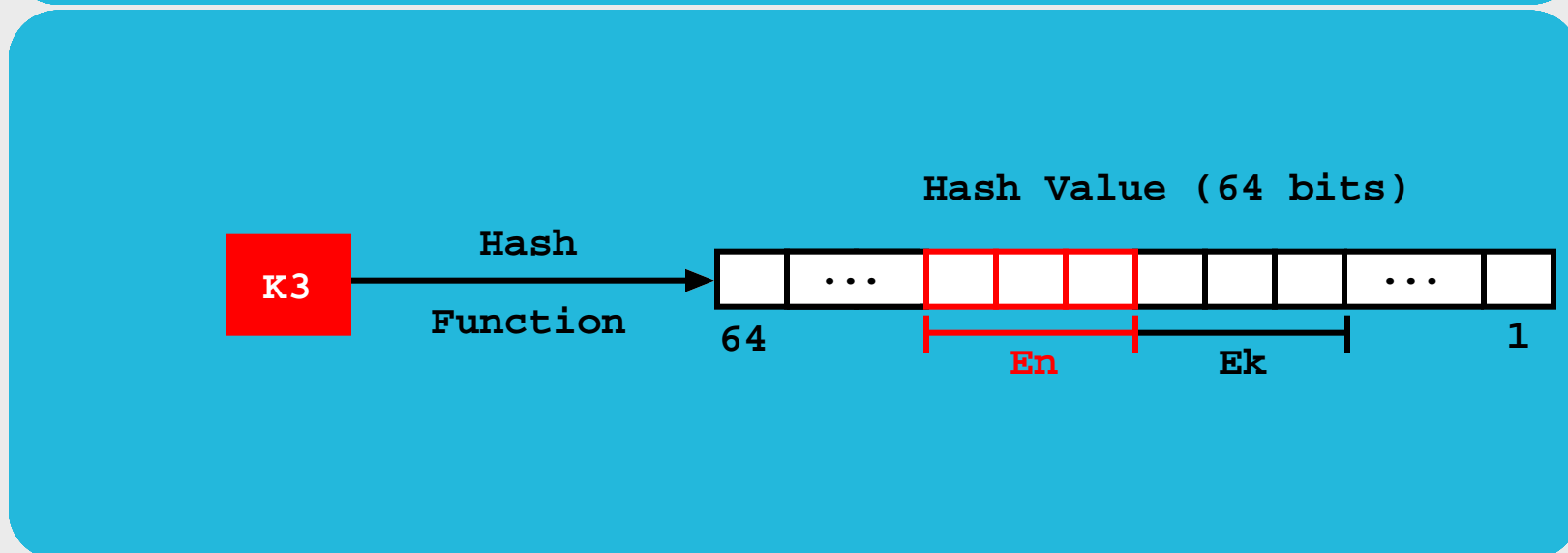
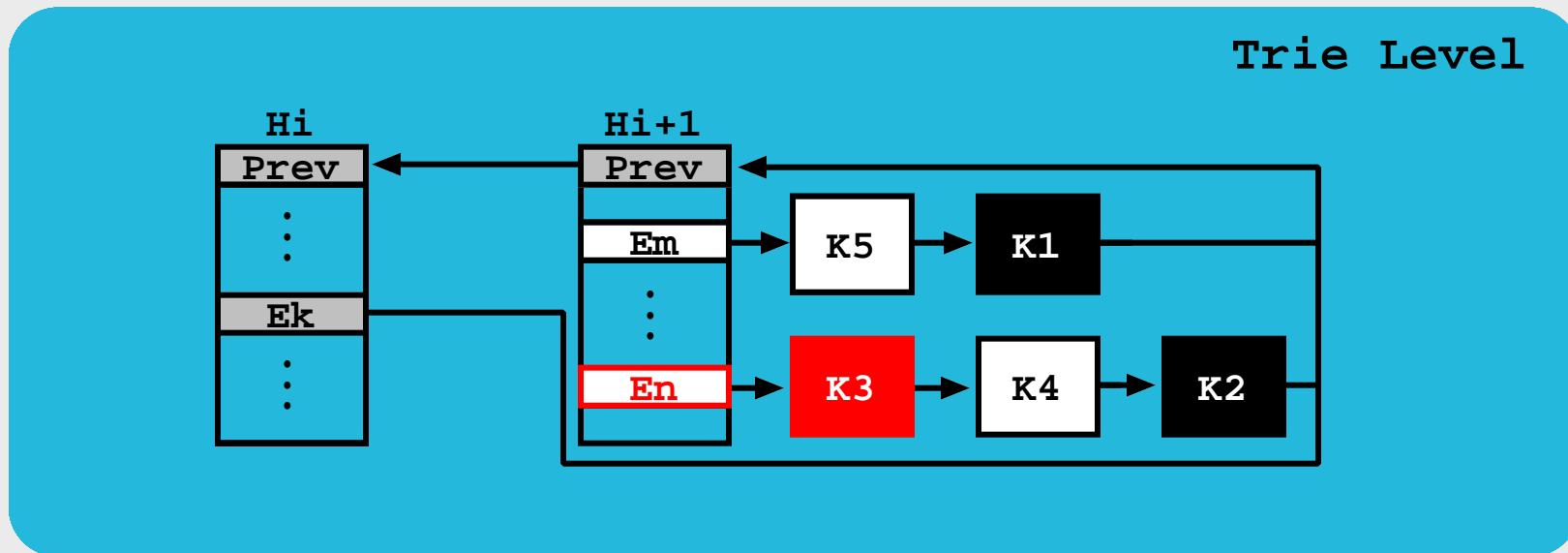
Lock-Free Hash Tries - Key Ideas



Lock-Free Hash Tries - Key Ideas

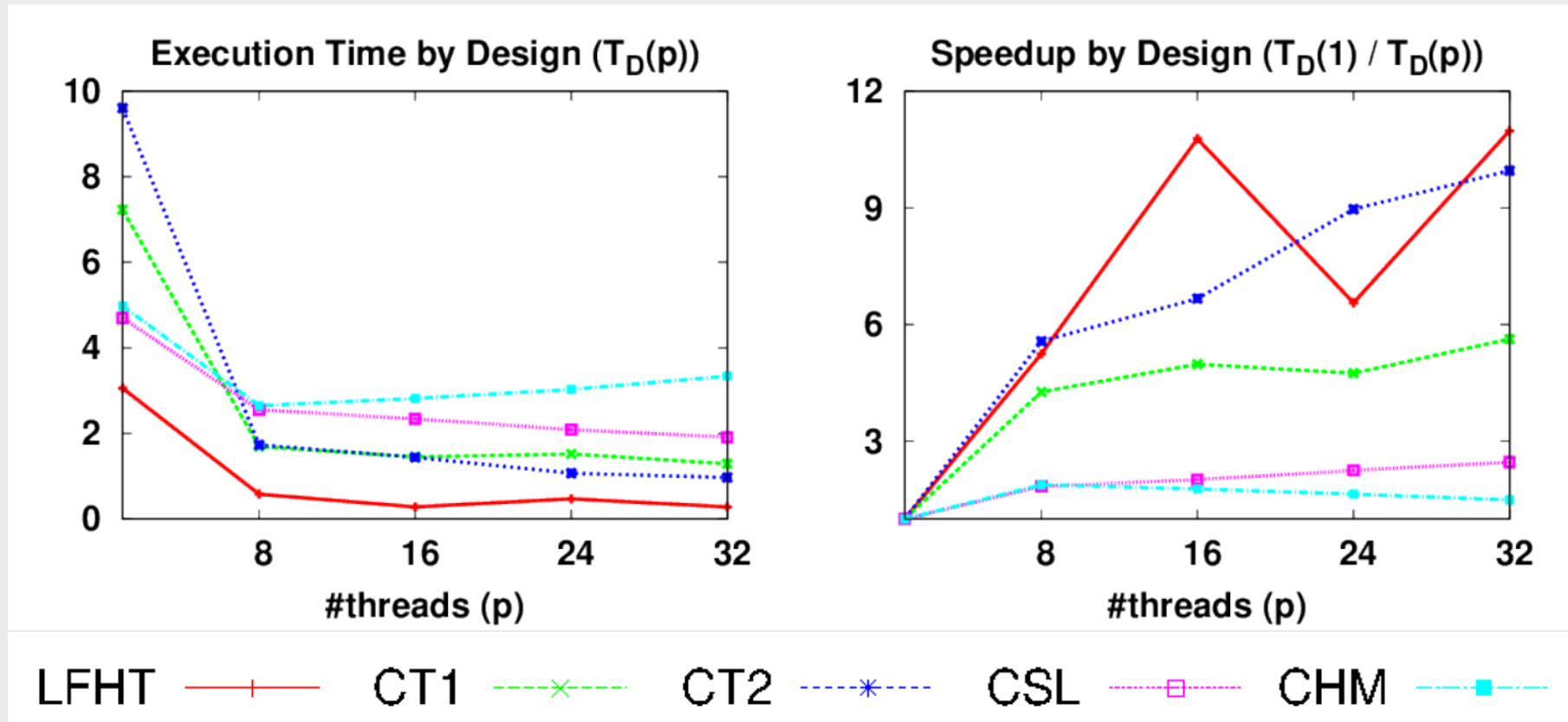


Lock-Free Hash Tries - Key Ideas



Experimental Results - External Framework

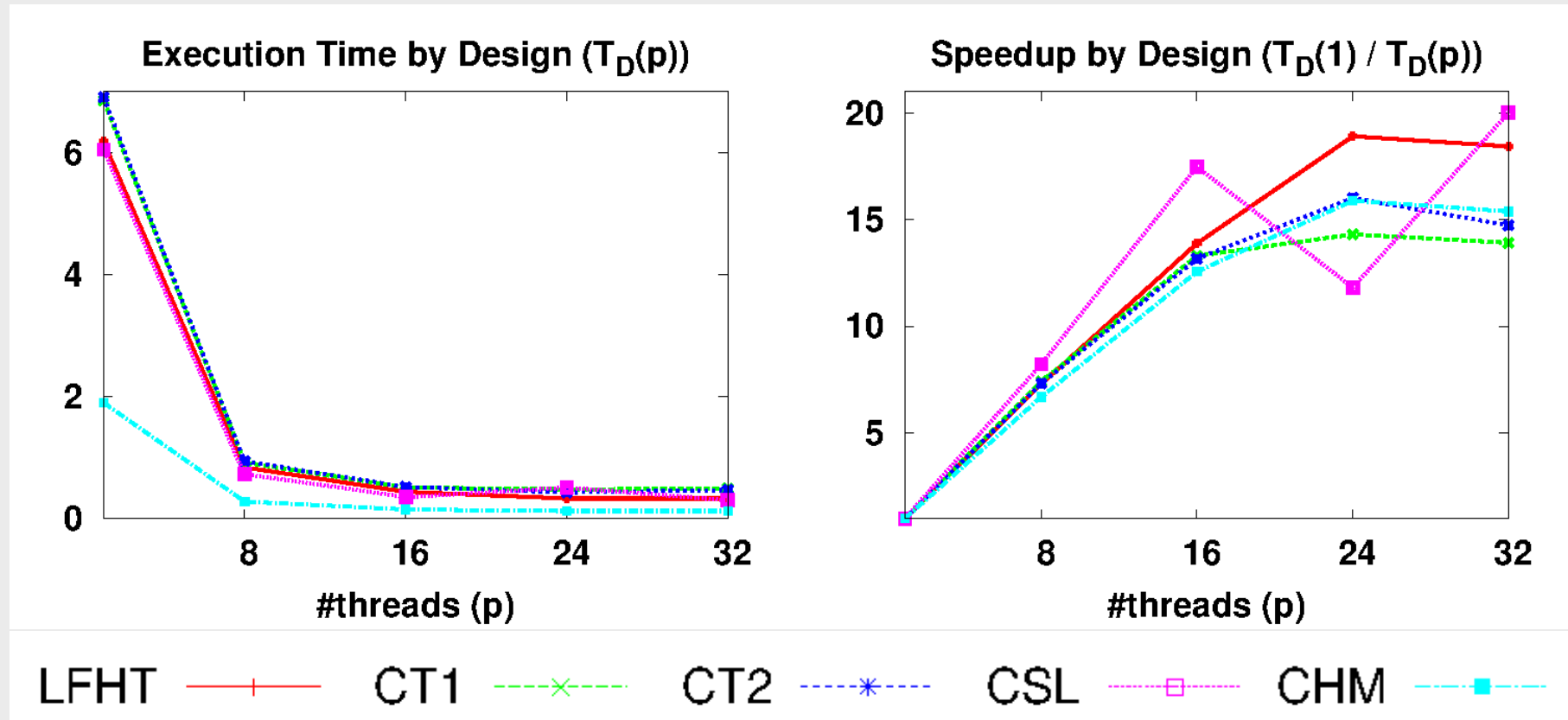
- Comparison in a **32 Core AMD** machine. Threads **insert** different items.



- **LFHT** Lock-Free Hash Tries - **CT1** /**CT2** C-Tries Versions (1/2)
- **CSL** Concurrent Skip Lists - **CHM** Concurrent Hash Maps

Experimental Results - External Framework

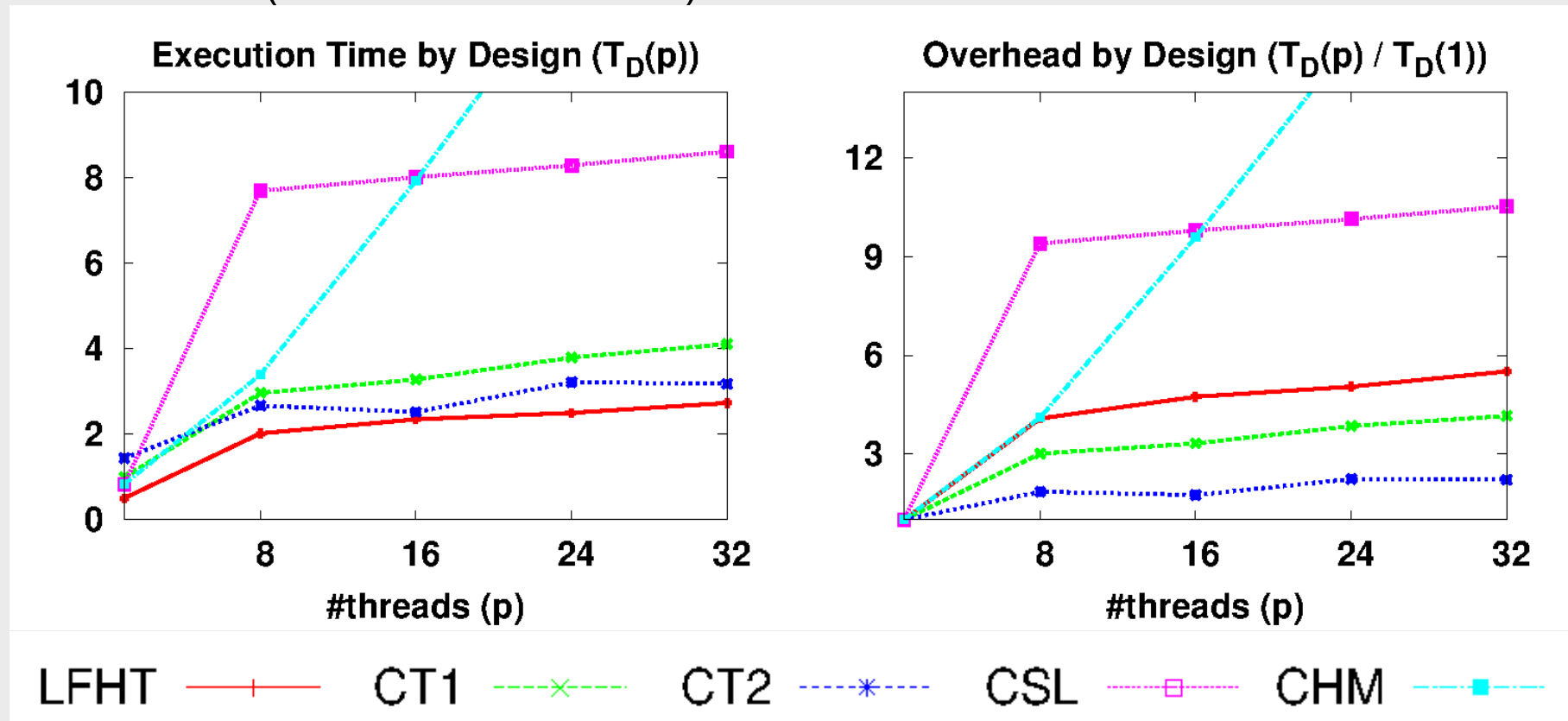
- Comparison in a **32 Core AMD** machine. Threads **lookup** for different items.



- **LFHT** Lock-Free Hash Tries - **CT1 /CT2** C-Tries Versions (1/2)
- **CSL** Concurrent Skip Lists - **CHM** Concurrent Hash Maps

Experimental Results - External Framework

- Comparison in a **32 Core AMD** machine. All threads **lookup and insert** the same items (**worst case** scenario).

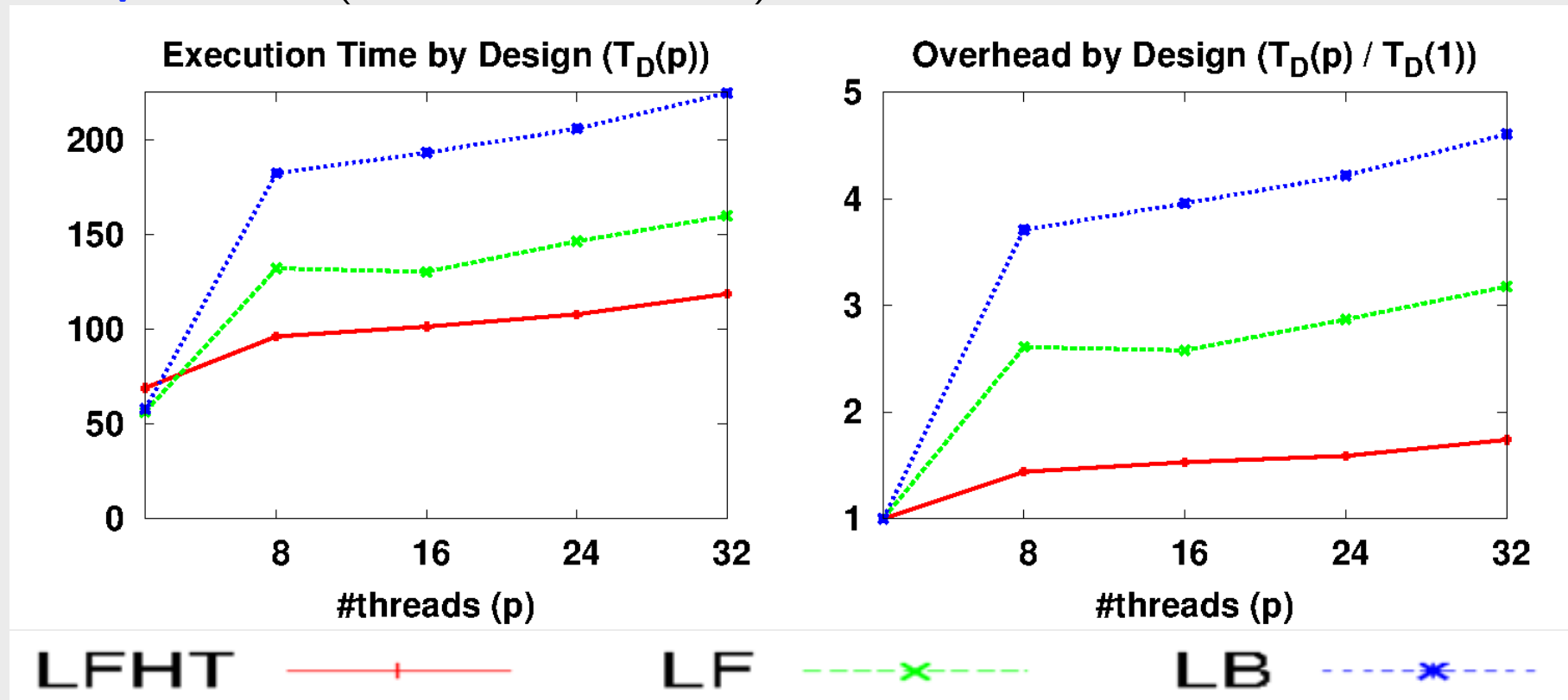


- **LFHT** Lock-Free Hash Tries - **CT1** /**CT2** C-Tries Versions (1/2)

- **CSL** Concurrent Skip Lists - **CHM** Concurrent Hash Maps

Experimental Results - Tabling Framework

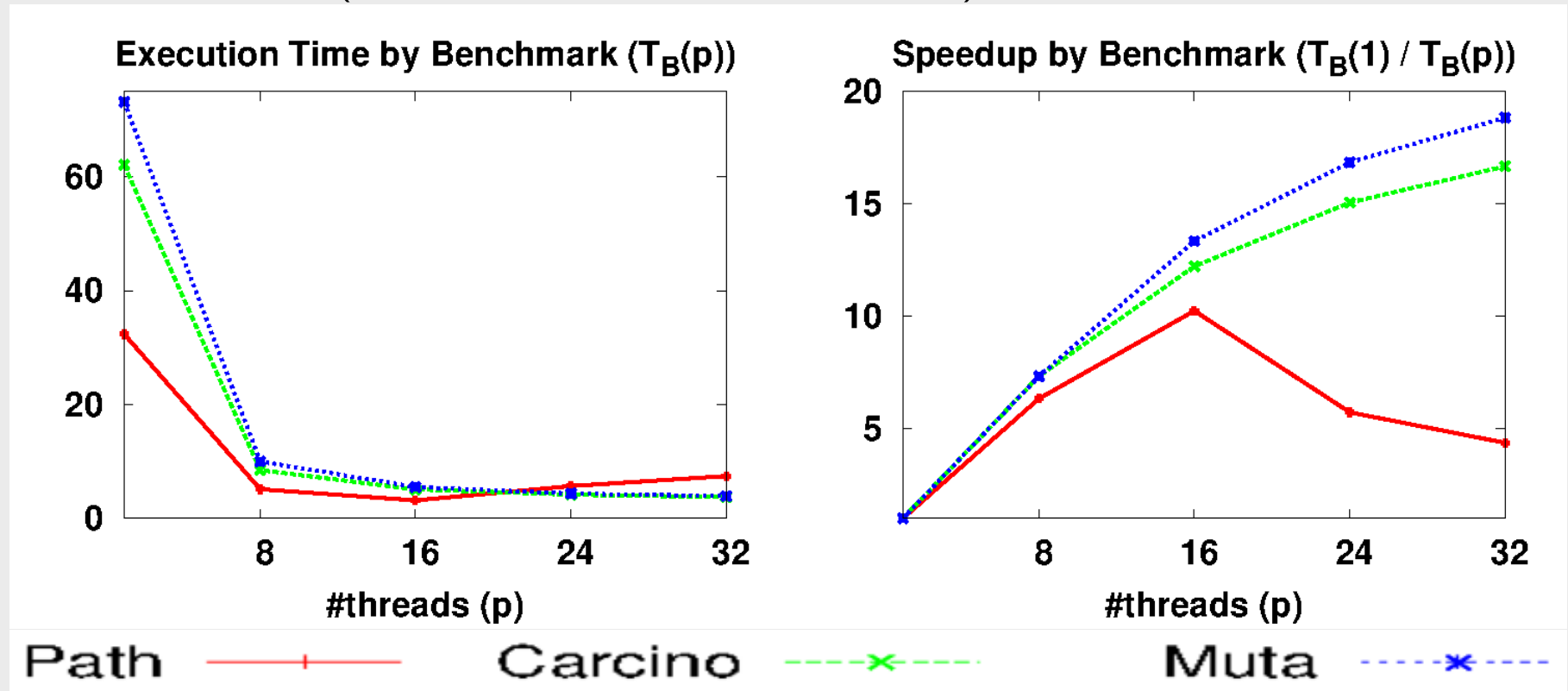
- Comparison in a **32 Core AMD** machine. All threads execute the **same sub-computations** (worst case scenario).



- **LFHT** Lock-Free Hash Tries - **LF** Lock-Free (old approach)
- **LB** Lock-Based (old approach)

Experimental Results - Tabling Framework

- Comparison in a **32 Core AMD** machine. All threads execute **different sub-computations** (**LFHT Lock-Free Hash Tries**).



- **Path** Path problem using a graph with a grid configuration
- **Carcino / Muta** (genesis) Inductive Logic Programming Benchmarks

Conclusions and Further Work

- We have presented a **novel**, **efficient** and **lock-free** design for a trie hash data structure applied to the multithreaded tabled evaluation of logic programs:
 - ◆ Improves the **efficiency** of the **concurrent lookup** and **insert operations** even in **worst case scenarios**.
 - ◆ The **paper discusses** the most relevant **implementation details** and **proves the correctness** of the design.
- Experimental results show that our approach can **effectively** reduce the **execution time** and **scale better**, when increasing the number of threads, than other designs.

Conclusions and Further Work

- We have presented a **novel**, **efficient** and **lock-free** design for a trie hash data structure applied to the multithreaded tabled evaluation of logic programs:
 - ◆ Improves the **efficiency** of the **concurrent lookup** and **insert operations** even in **worst case scenarios**.
 - ◆ The **paper discusses** the most relevant **implementation details** and **proves the correctness** of the design.
- Experimental results show that our approach can **effectively** reduce the **execution time** and **scale better**, when increasing the number of threads, than other designs.
- **Further work** will include:
 - ◆ Support the **concurrent deletion** of trie nodes
 - ◆ Extend the usage of the design to other parts of the Yap Prolog system.

Thank You !!!

Miguel Areias and Ricardo Rocha

CRACS & INESC-TEC LA

University of Porto, Portugal

miguel-areias@dcc.fc.up.pt *ricroc@dcc.fc.up.pt*

Yap Prolog : *http://www.dcc.fc.up.pt/~vsc/Yap*

Projects SIBILA and PEst: *http://cracs.fc.up.pt/*

FCT Grant: SFRH/BD/69673/2010

