The Internals of a Novel Lock-Free Hash Map Design

Miguel Areias
joint work with Ricardo Rocha

CRACS & INESC-TEC LA
Faculty of Sciences, University of Porto, Portugal

Parallel Computing

U.PORTO
FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Talk Overview

- In **this talk** we will:
  - introduce key concepts about progress in **concurrent systems** and **lock-free progress**.

Talk Overview

- In **this talk** we will:
  - introduce key concepts about progress in **concurrent systems** and **lock-free progress**.
  - explain why **lock-free** is important in **highly** concurrent environments.

Talk Overview

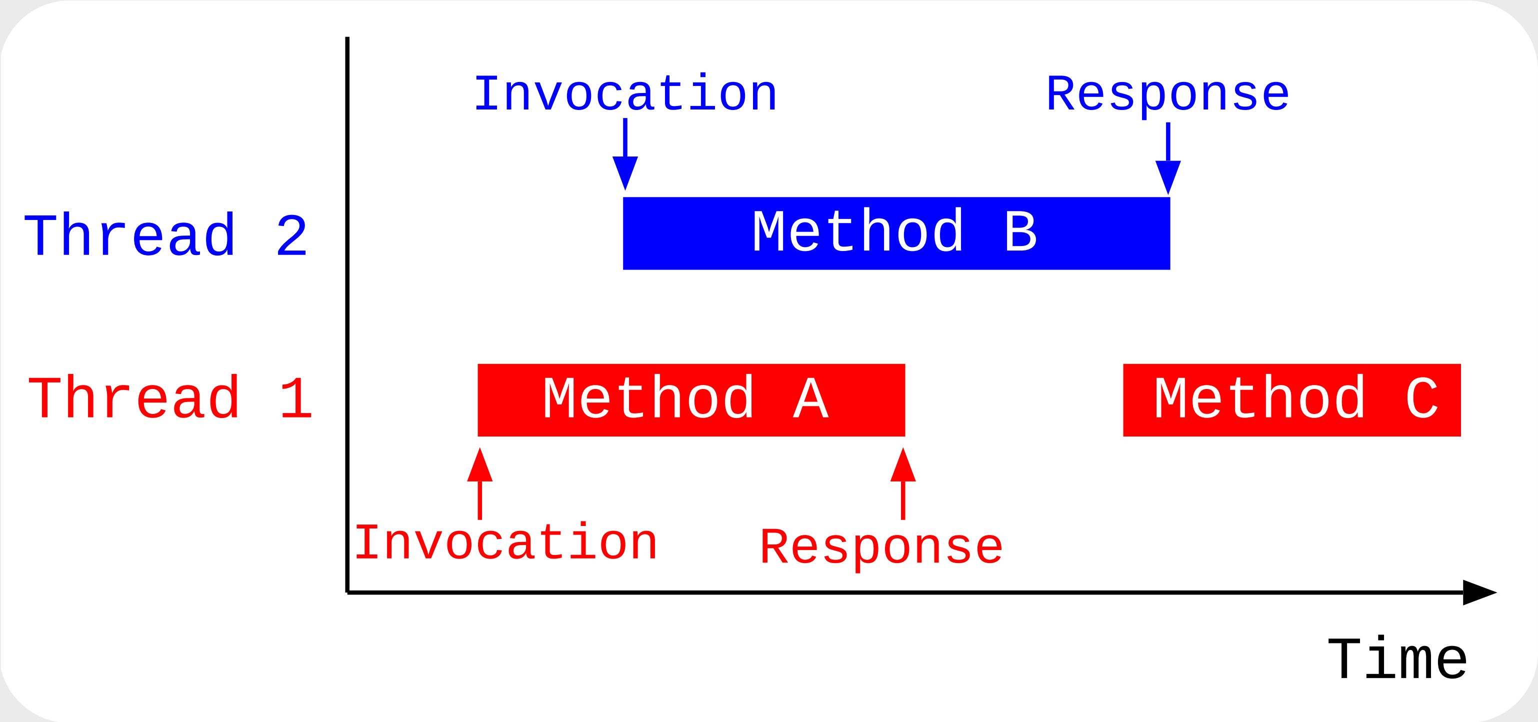
- In **this talk** we will:
  - introduce key concepts about progress in **concurrent systems** and **lock-free progress**.
  - explain why **lock-free** is important in **highly** concurrent environments.
  - present the **internals** of a **novel** and **lock-free hash map**.

Talk Overview

- In **this talk** we will:
  - introduce key concepts about progress in **concurrent systems** and **lock-free progress**.
  - explain why **lock-free** is important in **highly** concurrent environments.
  - present the **internals** of a **novel** and **lock-free hash map**.
  - present a **performance analysis** comparison between **state-of-the-art** concurrent **hash map designs**.

Progress in Concurrent Systems

- In 2011, Herlihy and Shavit presented a **grand unified explanation** for the **progress properties**. **Progress** is seen as the **number of steps** that threads take to **complete methods** within a concurrent object, i.e., the **number of steps** that threads take to **execute methods** between their **invocation** and their **response**.



Progress in Concurrent Systems

- **Progress** conditions are placed in a two-dimensional **periodical table**, where one of the axis defines the assumptions of the **operating system (OS)** scheduler, which might be **scheduler independent** or **scheduler dependent**, and the other axis defines the **maximal progress** and **minimal progress** provided by a method.

Dependency on the operating system scheduler

Dependency vs Progress	Non-Blocking Independent	Non-Blocking Dependent	Blocking Dependent
Every thread makes progress	Wait-Free	Obstruction-Free	Starvation-Free
Some thread make progress	Lock-Free	?	Deadlock-Free

Level of Progress

Maximal vs Minimal

Dependent vs Independent

Blocking vs Non-Blocking

Progress in Concurrent Systems

Dependency on the operating system scheduler

Dependency vs Progress	Non-Blocking Independent	Non-Blocking Dependent	Blocking Dependent
Every thread makes progress	Wait-Free	Obstruction-Free	Starvation-Free
Some thread make progress	Lock-Free	?	Deadlock-Free

Level of Progress

Maximal vs Minimal

Dependent vs Independent

Blocking vs Non-Blocking

- For the assumptions about the **OS scheduler**, a scheduler **independent** assumption, guarantees **progress** as long as threads are **scheduled** and no matter how they are scheduled. A scheduler **dependent** assumption, means that the **progress** of threads relies on the **OS scheduler** to satisfy certain properties.

Progress in Concurrent Systems

Dependency on the operating system scheduler

Dependency vs Progress	Non-Blocking Independent	Non-Blocking Dependent	Blocking Dependent
Every thread makes progress	Wait-Free	Obstruction-Free	Starvation-Free
Some thread make progress	Lock-Free	?	Deadlock-Free

Level of Progress

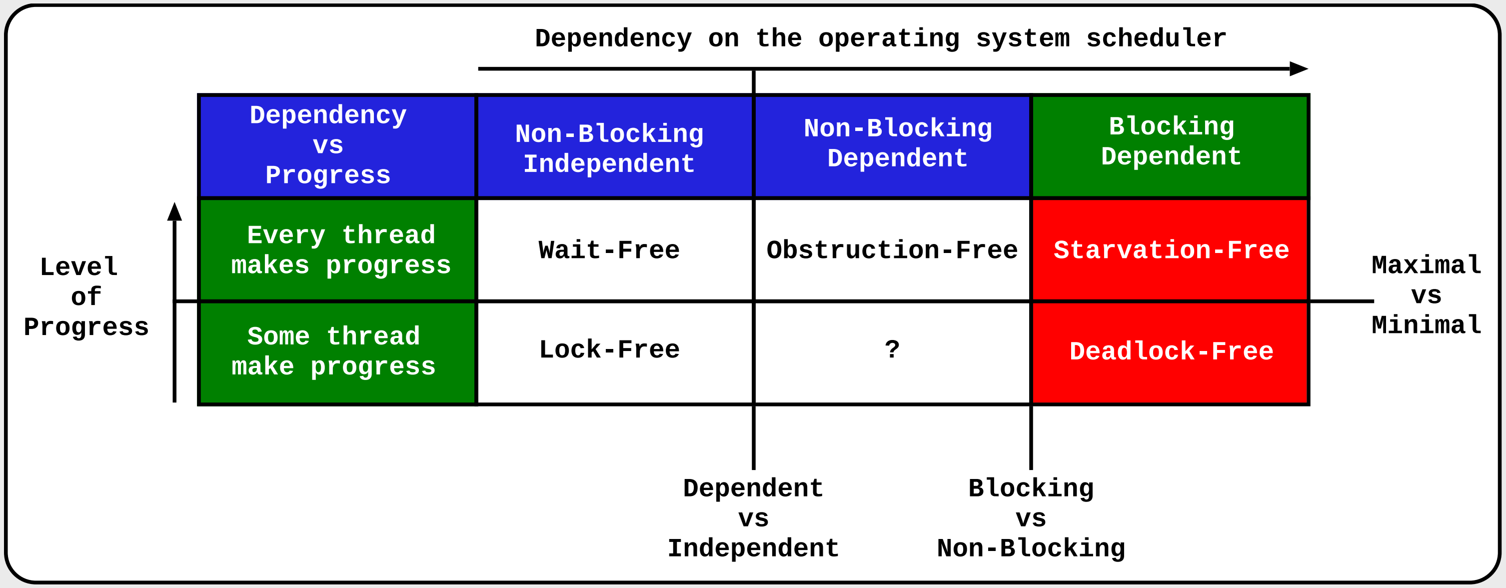
Maximal vs Minimal

Dependent vs Independent

Blocking vs Non-Blocking

- For the assumptions about the **OS scheduler**, a scheduler **independent** assumption, guarantees **progress** as long as threads are **scheduled** and no matter how they are scheduled. A scheduler **dependent** assumption, means that the **progress** of threads relies on the **OS scheduler** to satisfy certain properties.
- For the **level of progress**, a method provides the **minimal progress**, if a thread calling the method can take an **infinite number of steps** without returning. A method provides the **maximal progress**, if a thread calling the method takes a **finite number of steps** to return from the method.

Progress in Concurrent Systems



- **Deadlock-free** (threads cannot delay each other perpetually) and **starvation-free** (a critical region cannot be denied to a thread perpetually) properties **guarantee progress**, however, they **depend** on the assumption that the **OS scheduler** will **allow** each thread within a **critical region** to be **able to run** a **sufficient amount** of time, so that it can **leave** the **critical section** (**blocking dependent**).

Progress in Concurrent Systems

Dependency on the operating system scheduler

Dependency vs Progress	Non-Blocking Independent	Non-Blocking Dependent	Blocking Dependent
Every thread makes progress	Wait-Free	Obstruction-Free	Starvation-Free
Some thread make progress	Lock-Free	?	Deadlock-Free

Level of Progress

Maximal vs Minimal

Dependent vs Independent

Blocking vs Non-Blocking

- **Obstruction-free** (a thread runs within a critical region in a bounded number of steps) requires the **OS scheduler** to allow each thread to **run in isolation** for a **sufficient** amount of time (**non-blocking dependent**).
- **Wait-free** (a thread is able to make progress in a finite number of steps) provides **maximal progress** and has **no requirements** on the **OS scheduler** (**non-blocking independent**).

Progress in Concurrent Systems

Dependency on the operating system scheduler

Dependency vs Progress	Non-Blocking Independent	Non-Blocking Dependent	Blocking Dependent
Every thread makes progress	Wait-Free	Obstruction-Free	Starvation-Free
Some thread make progress	Lock-Free	?	Deadlock-Free

Level of Progress

Maximal vs Minimal

Dependent vs Independent

Blocking vs Non-Blocking

- **Lock-free** provides **minimal progress** and has **no requirements** on the **OS scheduler** (**non-blocking independent**).
- **Lock-free** guarantees then that, on **every instant** of the execution of methods (between their invocation and their response), **at least one thread** is doing **progress** on its work.

Lock-Free Progress

- **Lock-Free objects** allow **greater concurrency** than **lock-based objects** since **semantically consistent** (non-interfering) methods **may execute in parallel**.
- **Lock-Free** techniques **do not use** traditional **locking** mechanisms.
  - **Avoid problems** such as:
    - **deadlocks** - threads **delaying** each other **perpetually**.
    - **convoying** - a thread holding a lock is **descheduled** by an **interrupt**.
    - **kill-tolerant** - a thread **is not immune** to the **dead** of other threads during the execution.

Lock-Free Progress

- **Lock-Free objects** allow **greater concurrency** than **lock-based objects** since **semantically consistent** (non-interfering) methods **may execute in parallel**.
- **Lock-Free** techniques **do not use** traditional **locking** mechanisms.
  - **Avoid problems** such as:
    - **deadlocks** - threads **delaying** each other **perpetually**.
    - **convoying** - a thread holding a lock is **descheduled** by an **interrupt**.
    - **kill-tolerant** - a thread **is not immune** to the **dead** of other threads during the execution.
    - **priority inversion** - a thread with **high priority** is **preempted** by a thread with **lower priority**.
    - **contention** - a thread **waiting** for a lock that is being **held by another** thread.

Lock-Free Progress

- Instead, they are **based** in placing simple **atomic operations** in key **concurrency spots**, to **improve performance** and **ensure correctness** (formal proof of **linearization**).
  - **Atomic** operations **cannot** be interrupted (intrinsically **thread safe**).

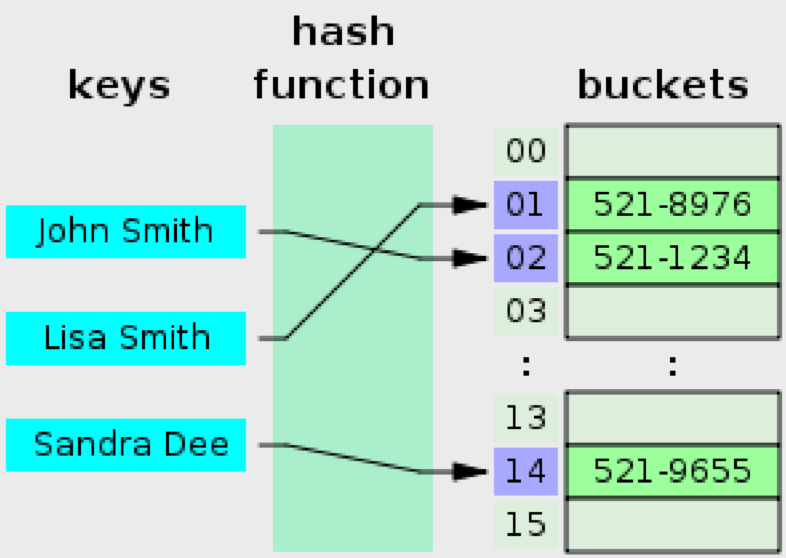
Lock-Free Progress

- Instead, they are **based** in placing simple **atomic operations** in key **concurrency spots**, to **improve performance** and **ensure correctness** (formal proof of **linearization**).
  - **Atomic** operations **cannot** be interrupted (intrinsically **thread safe**).

- At the **implementation level**, they take advantage of the **CAS (Compare-and-Swap) atomic operation**, that nowadays **can be found** in many common **hardware** architectures.
  - **CAS**(**Memory_Reference**, **Expected_Value**, **New_Value**).

Hash Maps

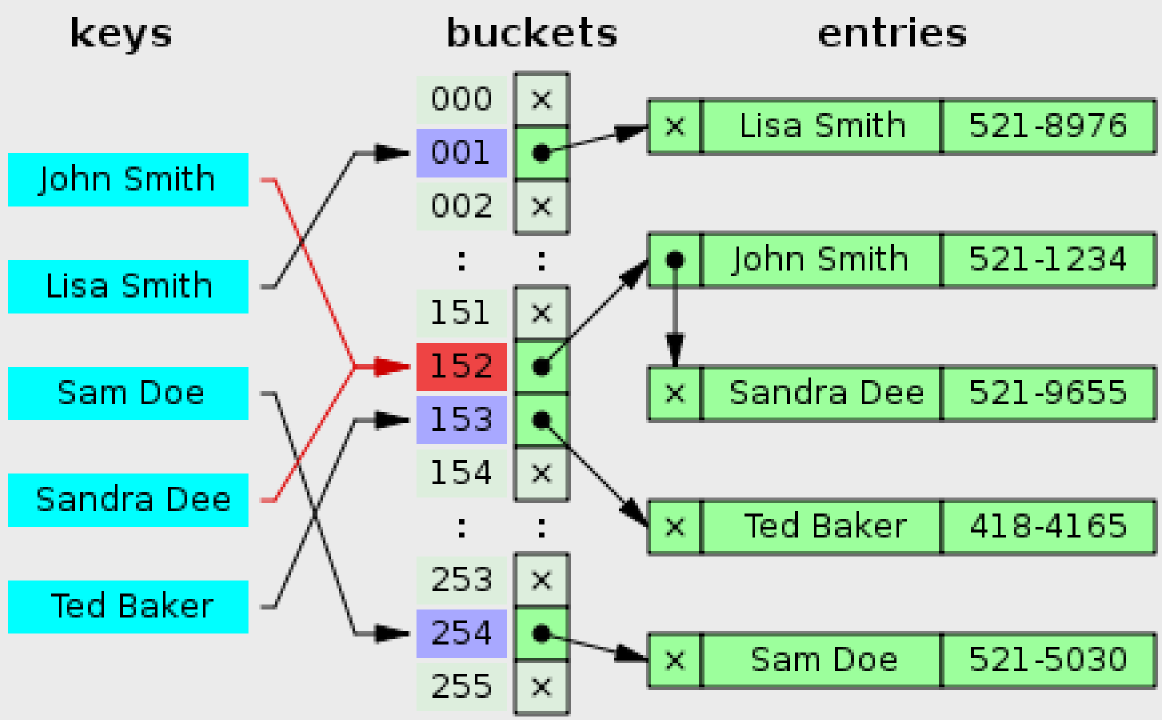
- **Hash maps** are **useful** to **store information** that can be organized as **pairs (K, C)**, where **K** is an identifier (or a **key**) and **C** is the **associated content**.



A small phone book as a hash map [Wikipedia].

Hash Maps

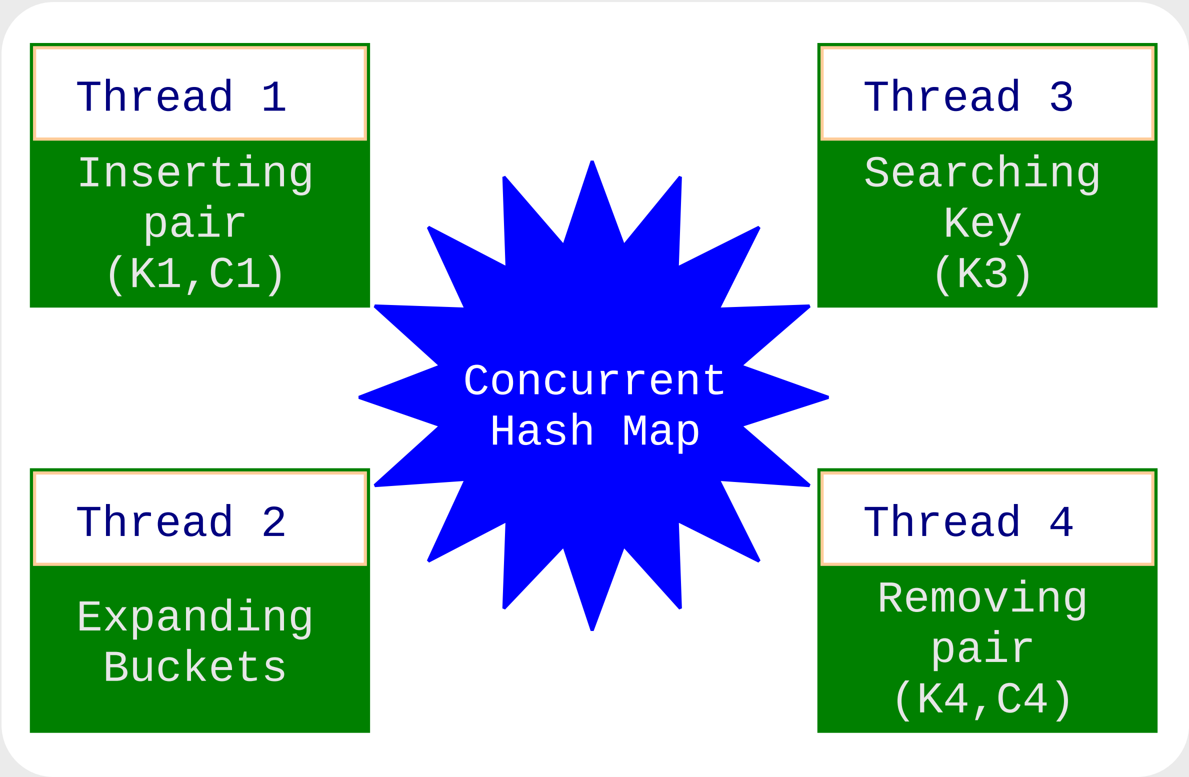
- Some of the most **usual methods** are:
  - **User-level** (externally activated by users) : **search**, **insert** and **remove**.
  - **Kernel-level** (internally activated by thresholds): **expansion** (key collision) (and **compression**, which will not be discussed in this talk).



Key collisions resolved using a separate chaining mechanism [Wikipedia].

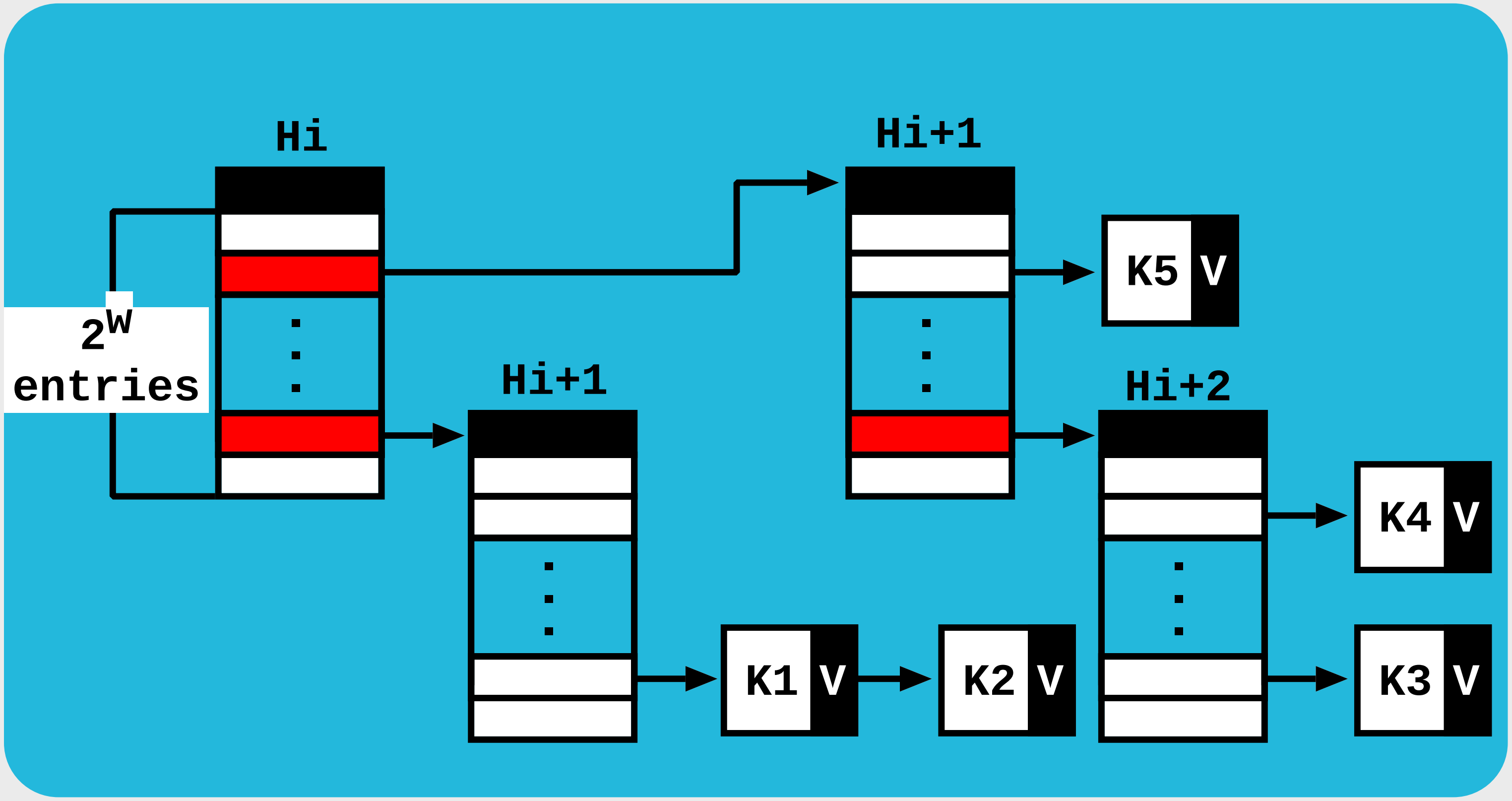
Concurrent Hash Maps

- **Multithreaded hash maps** allow the **concurrent execution** of multiple **methods**.
  - **Each operation** runs **independently**, but at the engine level, **all methods share** the underlying **data structures** that support the **hash map**.

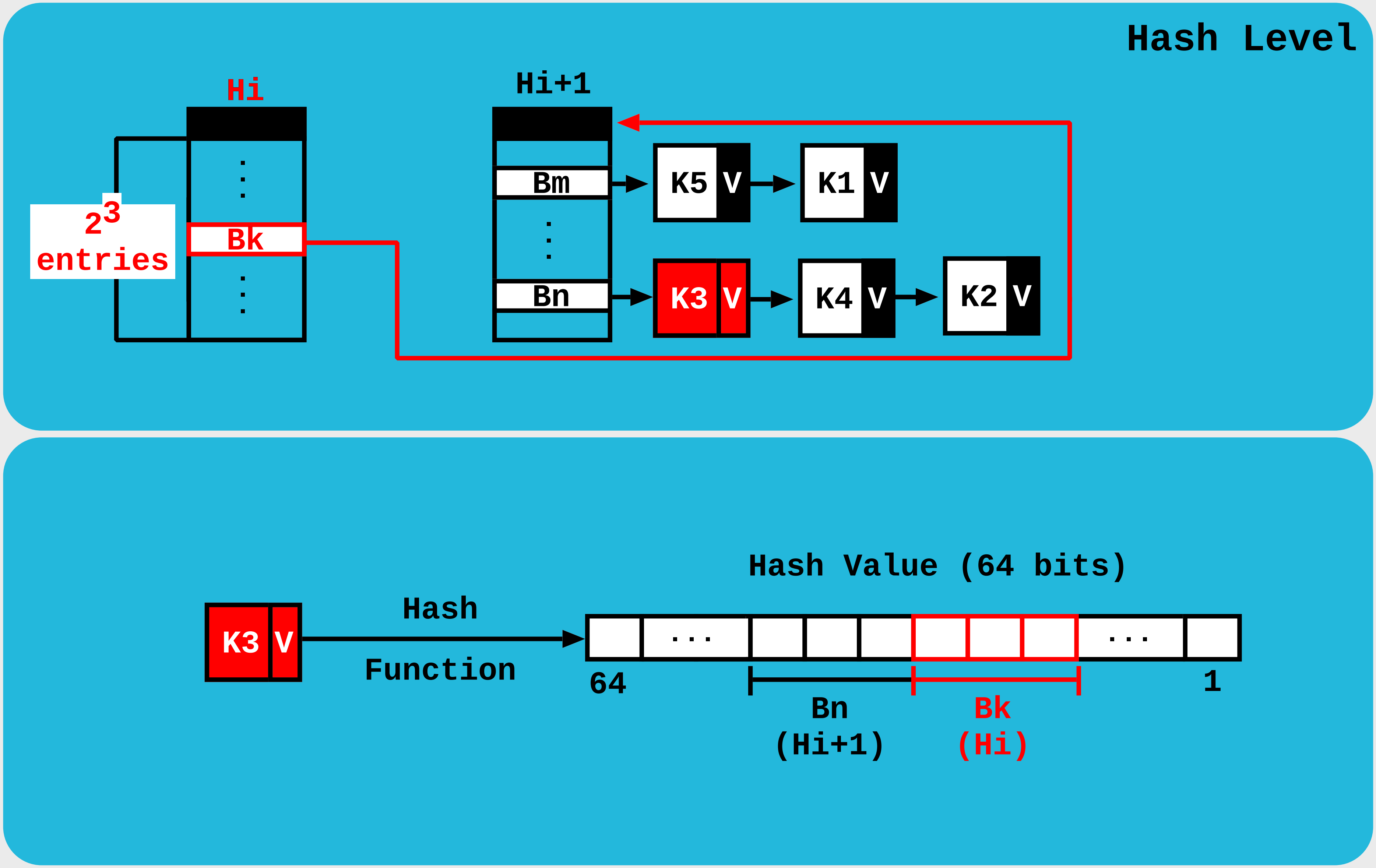


The FP Design - Hash Trie Structure

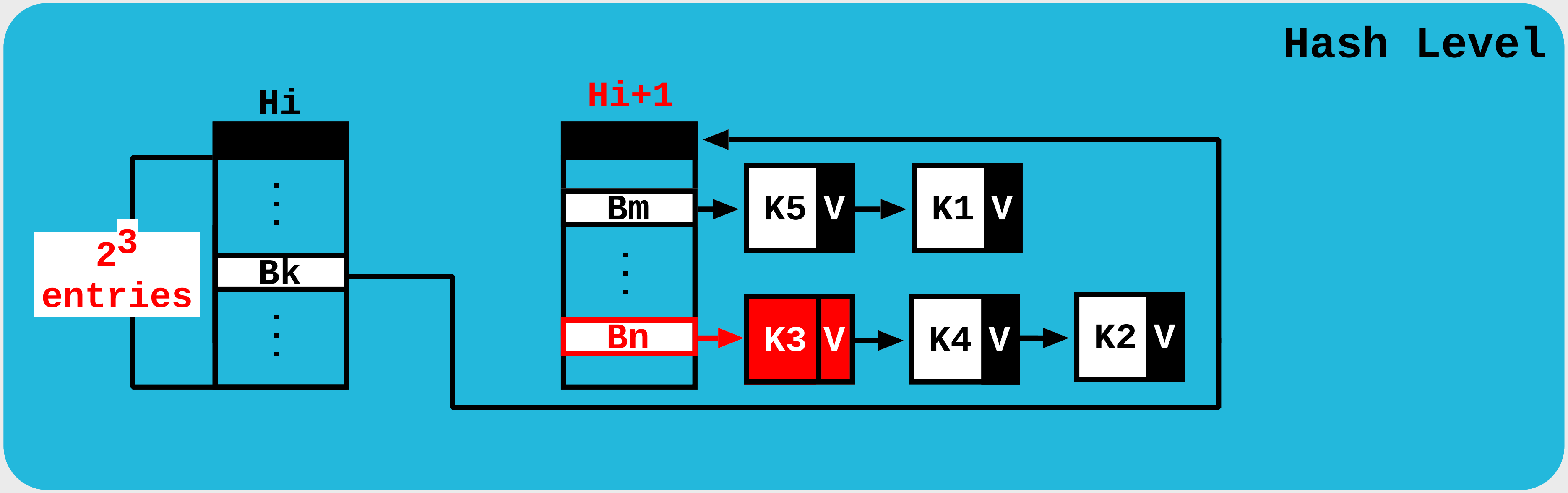
- **Hash buckets** refer to a **chaining mechanism** that supports **key collisions**.
- **Chain nodes** store **pairs (Key, Content, (Next_On_Chain, State))**. For the **sake of simplicity** we will present only **(Key, (Next_On_Chain, State))**. **State** can be **valid (V)** or **invalid (I)**.

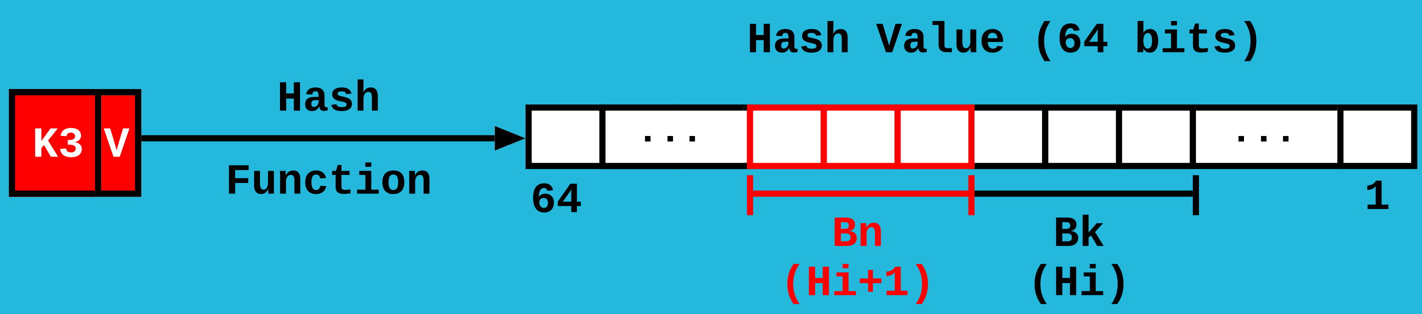


The FP Design - Searching for K3



The FP Design - Searching for K3



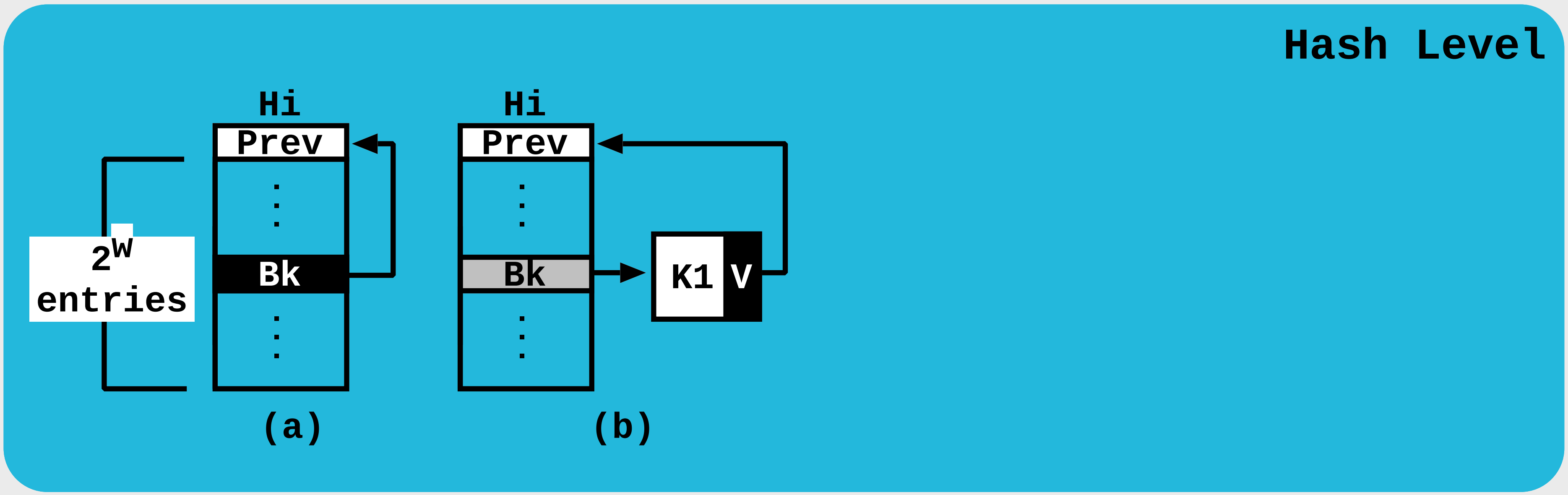


The FP Design - Internals

- To support **multithreading**, our design allows **threads** to:
  - **Recover from preemption**, by using a **Prev** field to traverse the **hash buckets** backwards.
  - **Identify chains**, by using a **back-reference** on the end of each chain.
  - **Maintain consistency**, by using **CAS** on write operations.

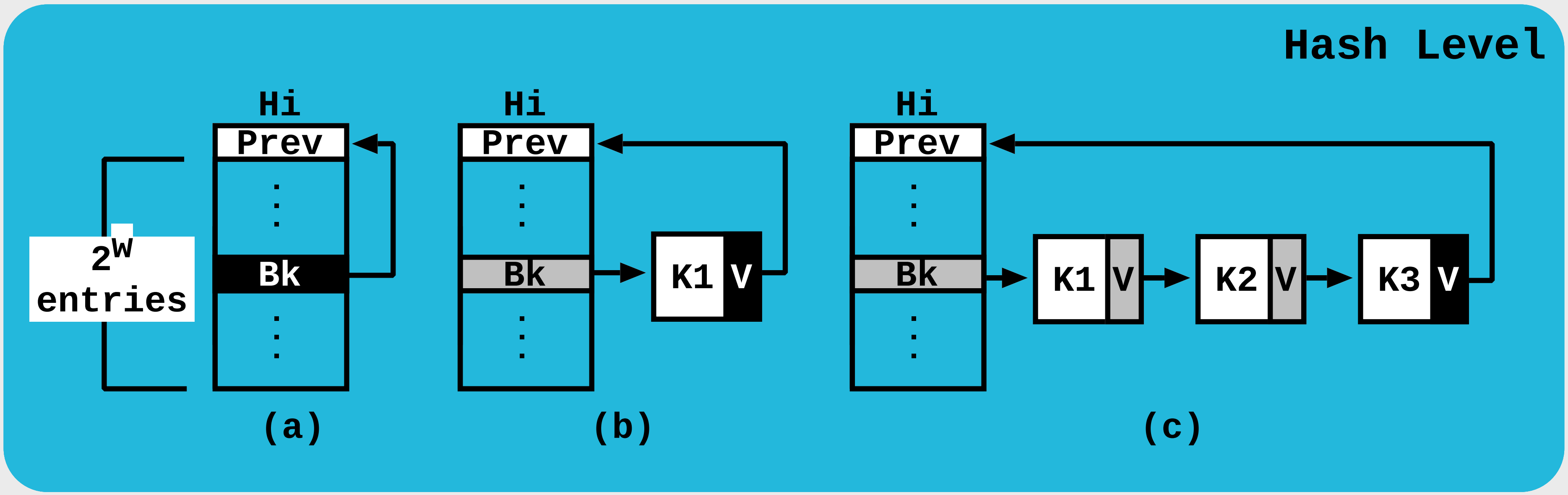
The FP Design - Internals

- To support **multithreading**, our design allows **threads** to:
  - **Recover from preemption**, by using a **Prev** field to traverse the **hash buckets** backwards.
  - **Identify chains**, by using a **back-reference** on the end of each chain.
  - **Maintain consistency**, by using **CAS** on write operations.

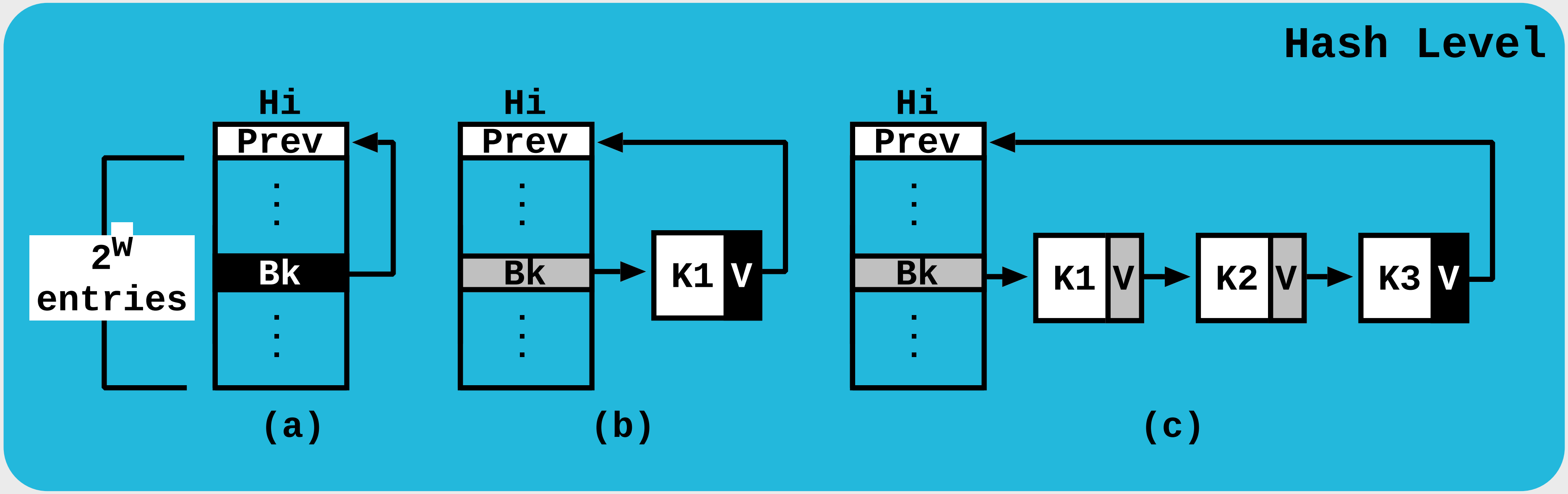


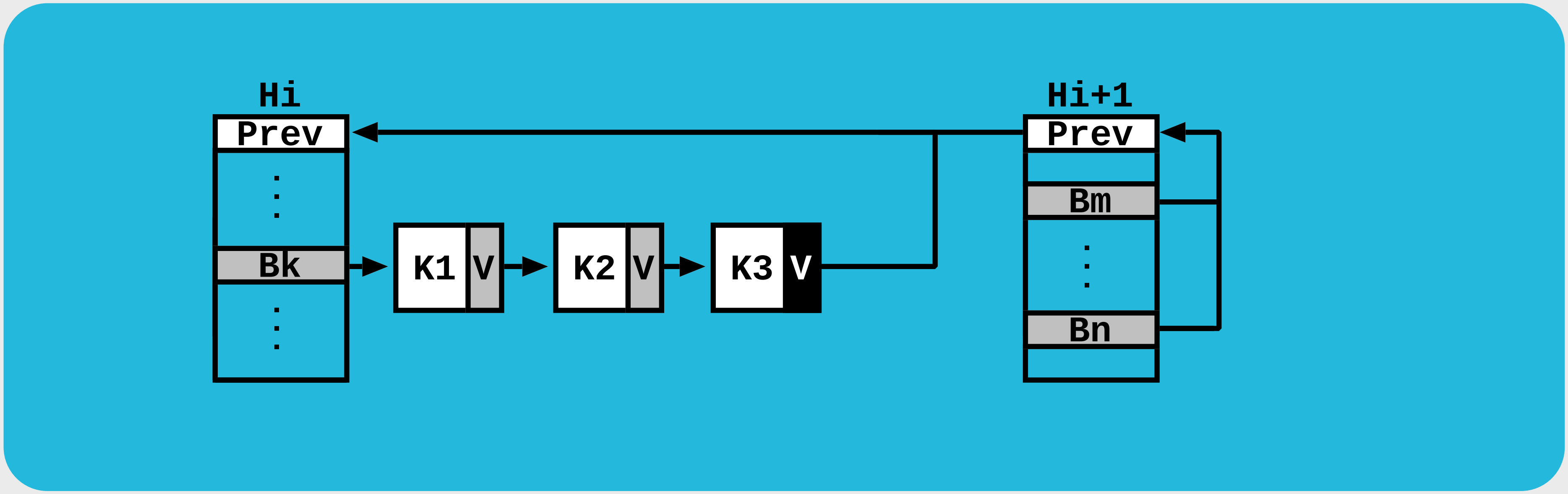
The FP Design - Internals

- To support **multithreading**, our design allows **threads** to:
  - **Recover from preemption**, by using a **Prev** field to traverse the **hash buckets** backwards.
  - **Identify chains**, by using a **back-reference** on the end of each chain.
  - **Maintain consistency**, by using **CAS** on write operations.

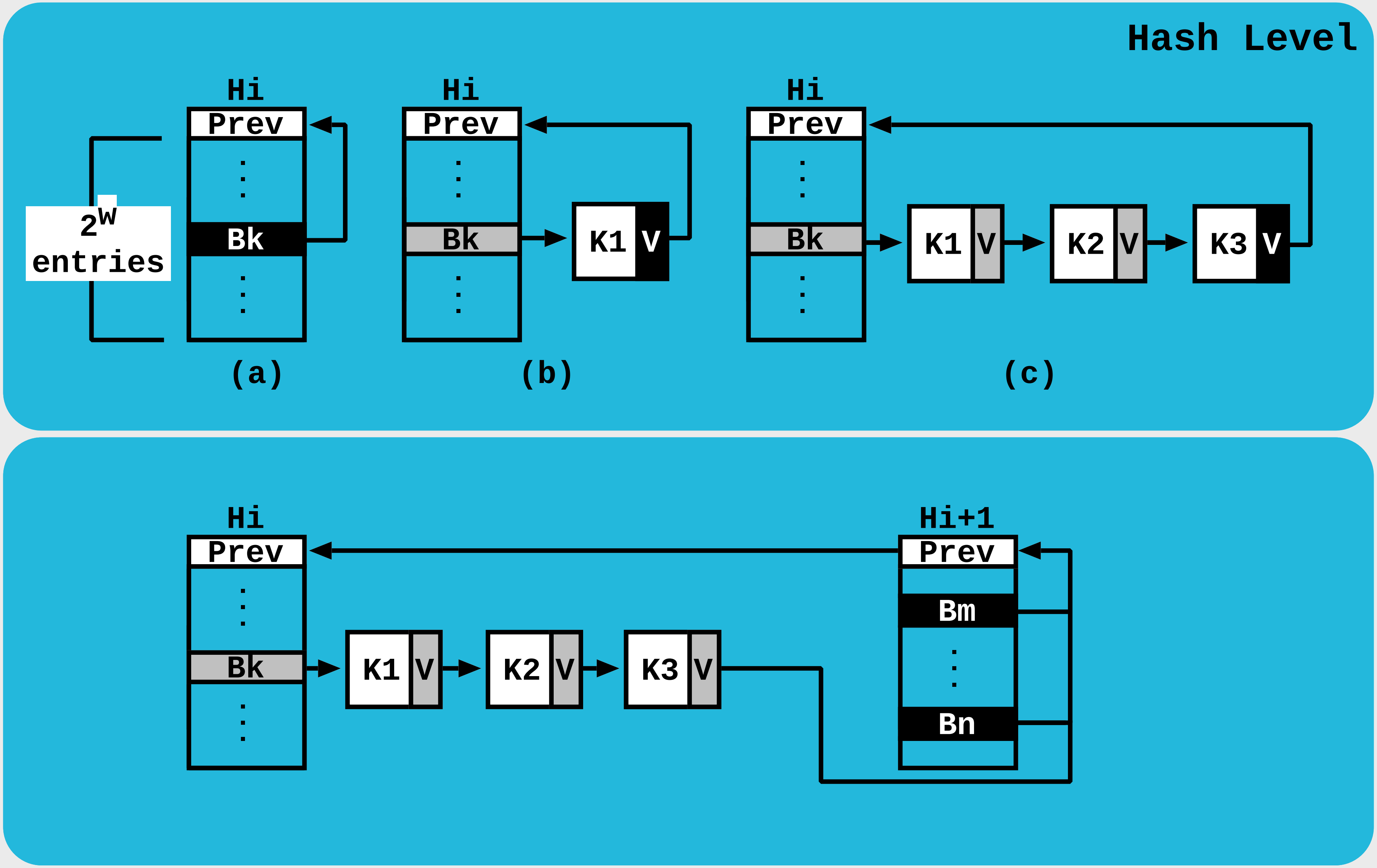


The FP Design - Expand (Valid Nodes Only)

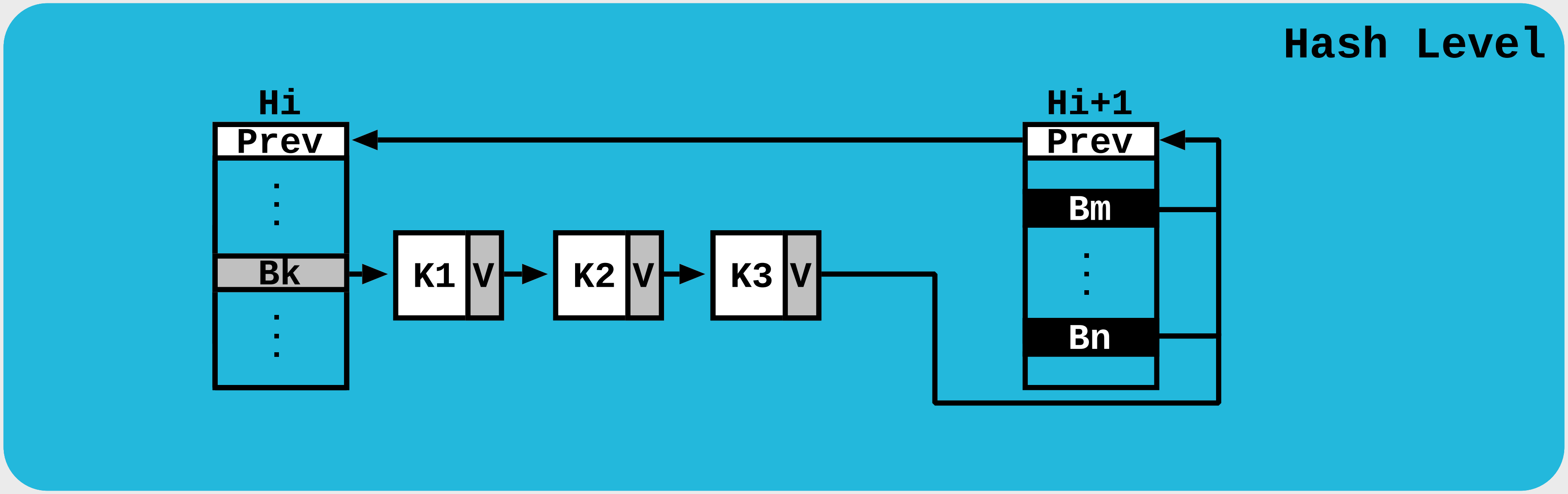




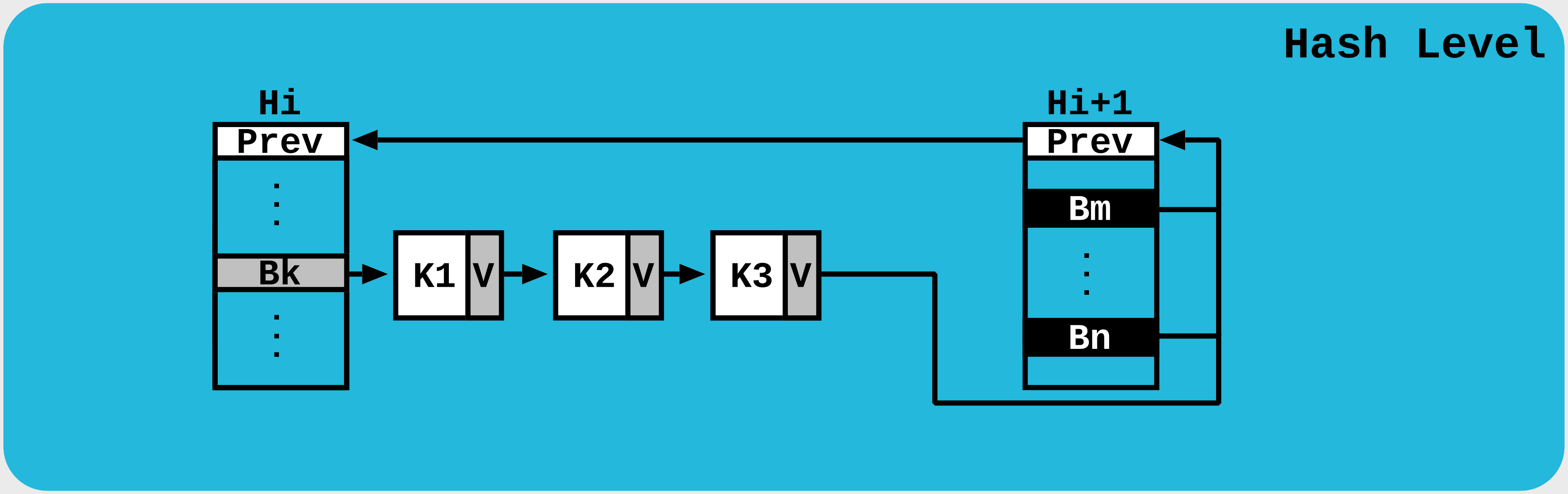
The FP Design - Expand (Valid Nodes Only)

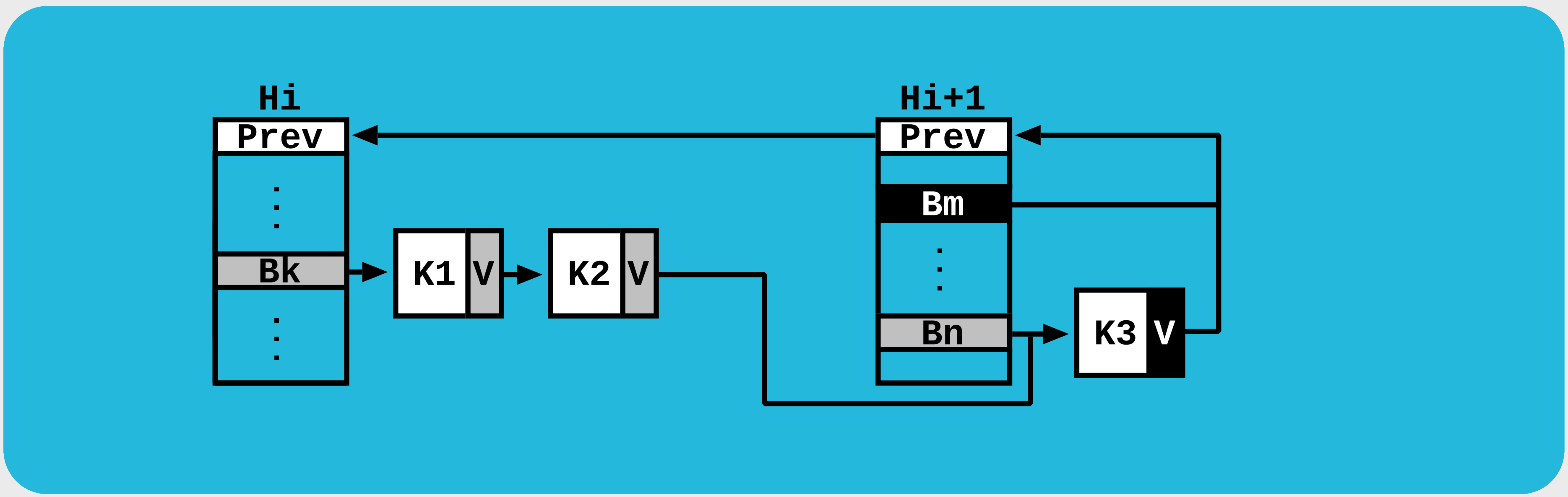


The FP Design - Expand (Valid Nodes Only)

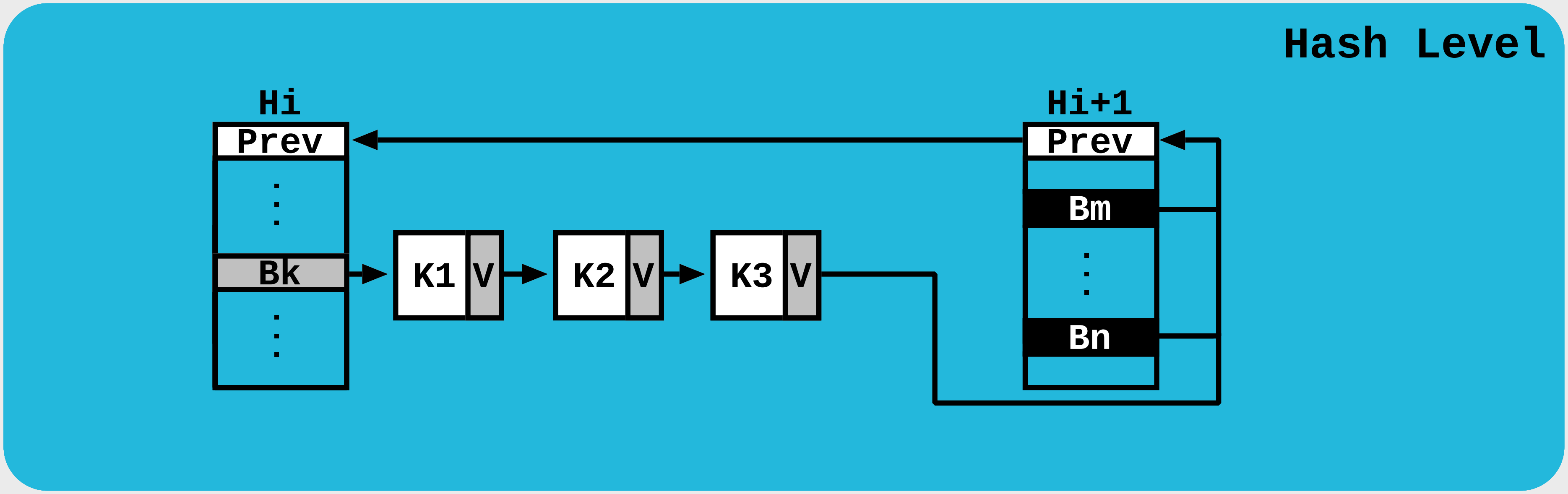


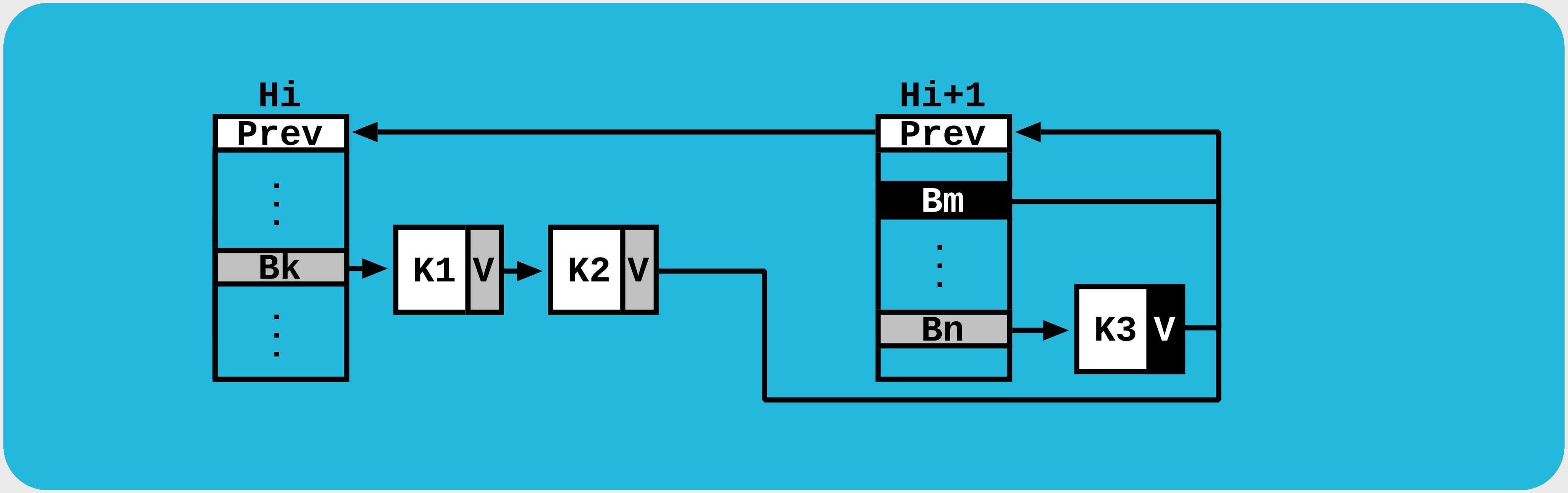
The FP Design - Expand (Valid Nodes Only)



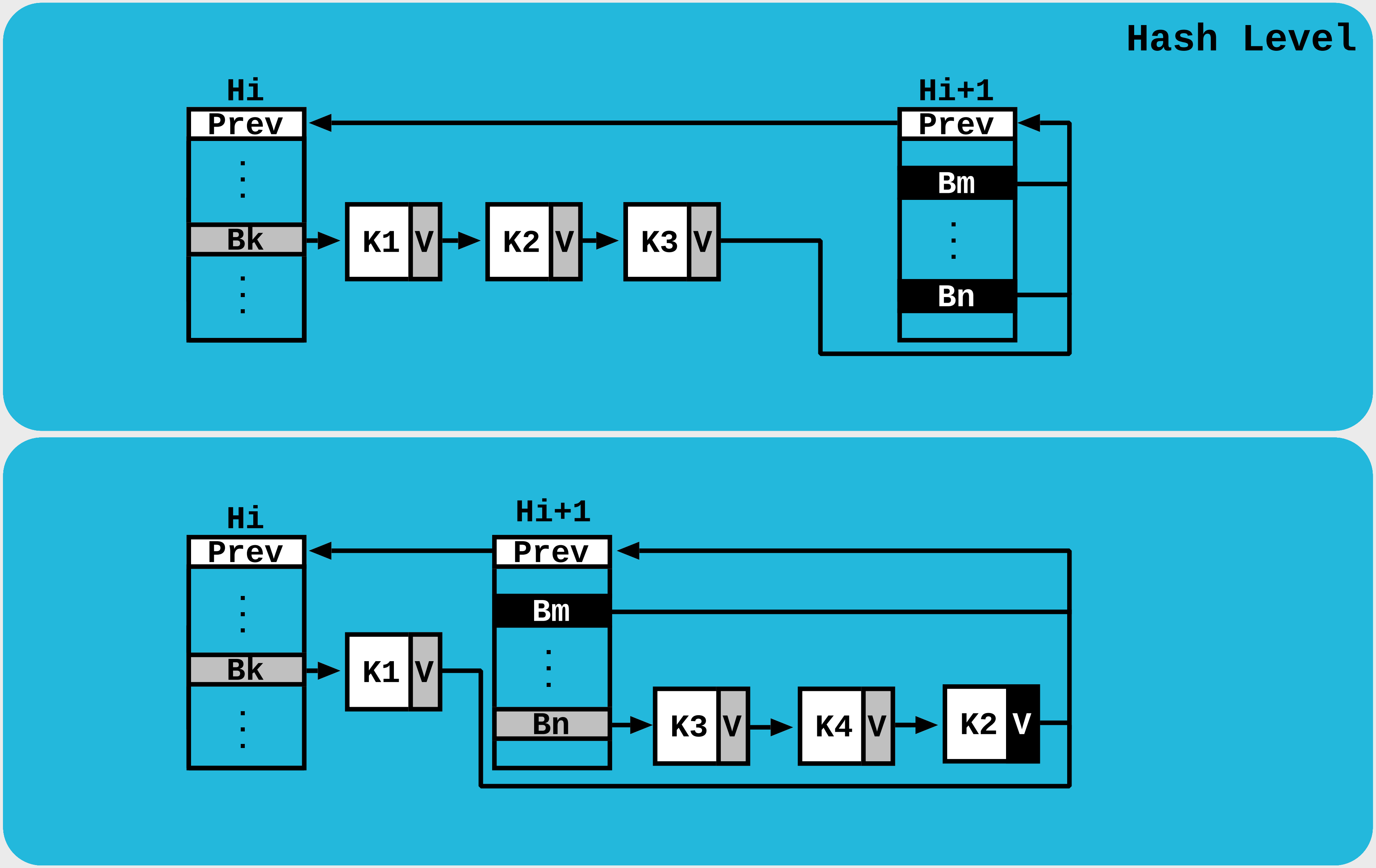


The FP Design - Expand (Valid Nodes Only)

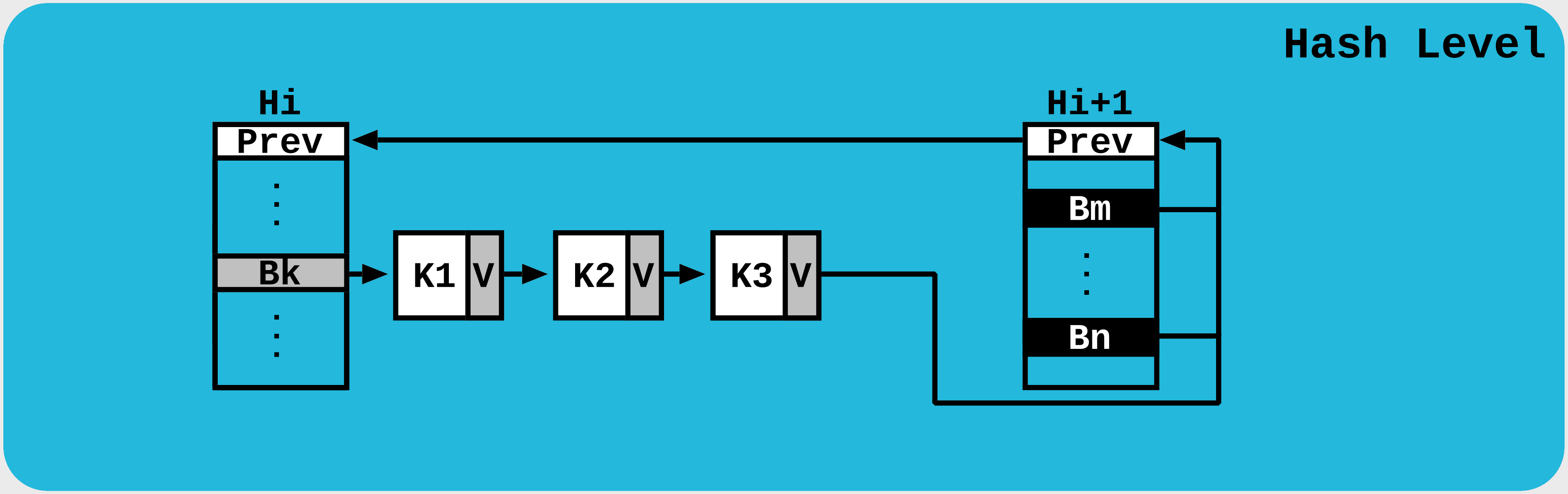


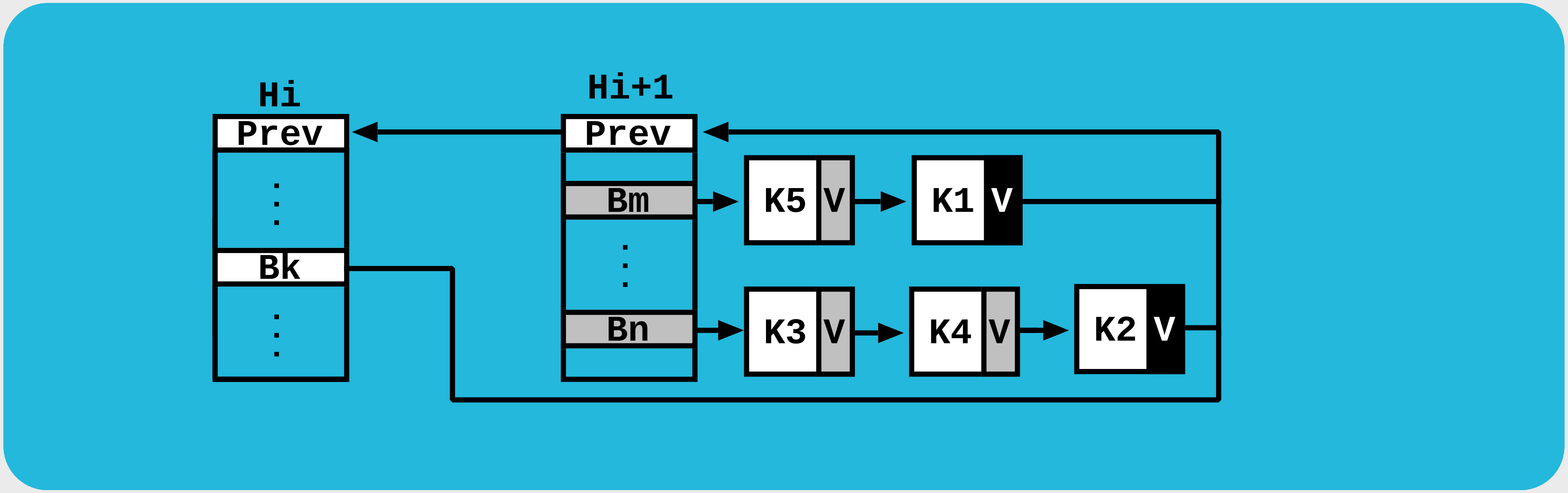


The FP Design - Expand (Valid Nodes Only)



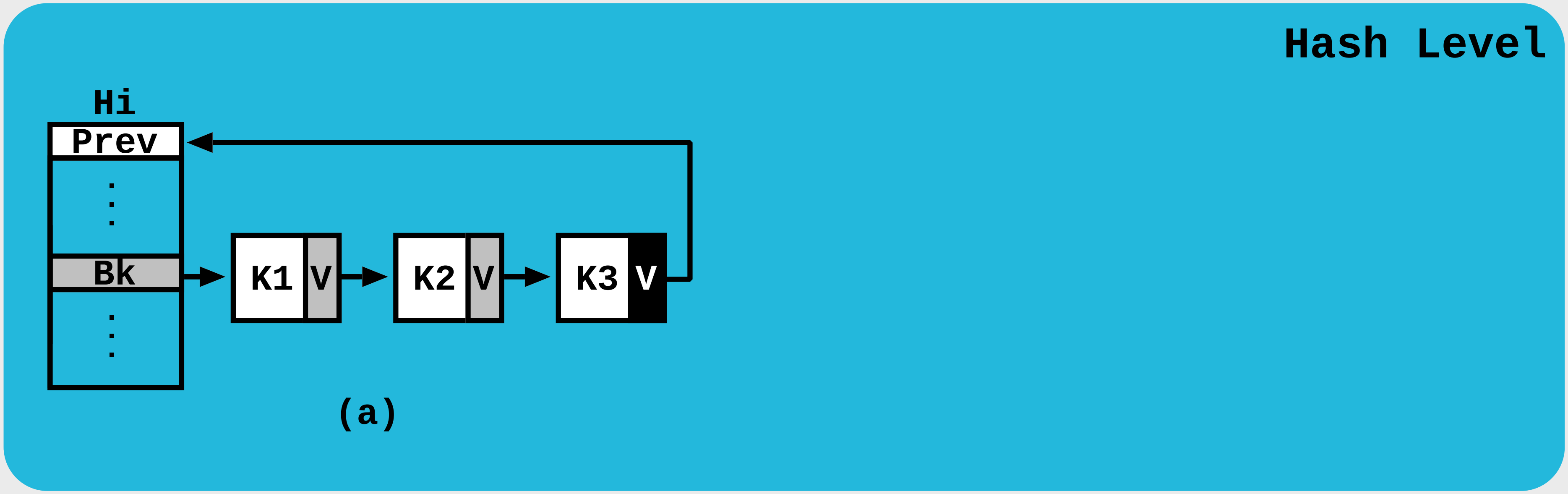
The FP Design - Expand (Valid Nodes Only)





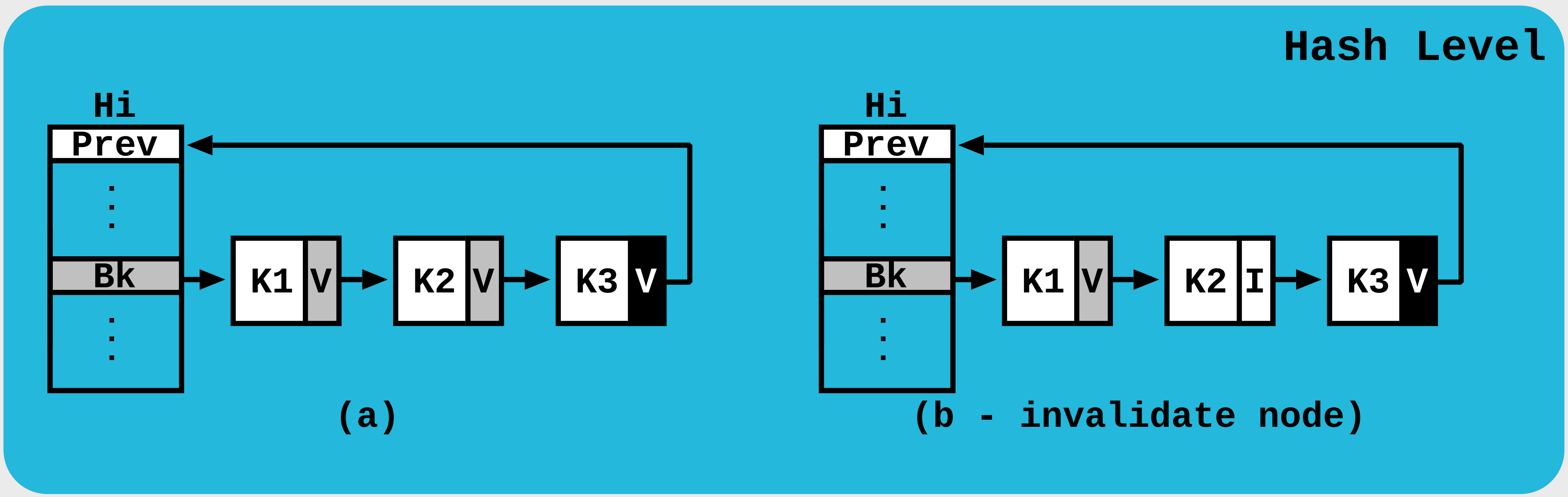
The FP Design - Remove

- The **remove operation** has two steps:
  - **Invalidate node** by **changing** its **state** from **valid** to **invalid**.
  - **Turn** the node **invisible** to all threads. Find **two valid** data structures (previous and next) and **bypass** the **invalid** node.



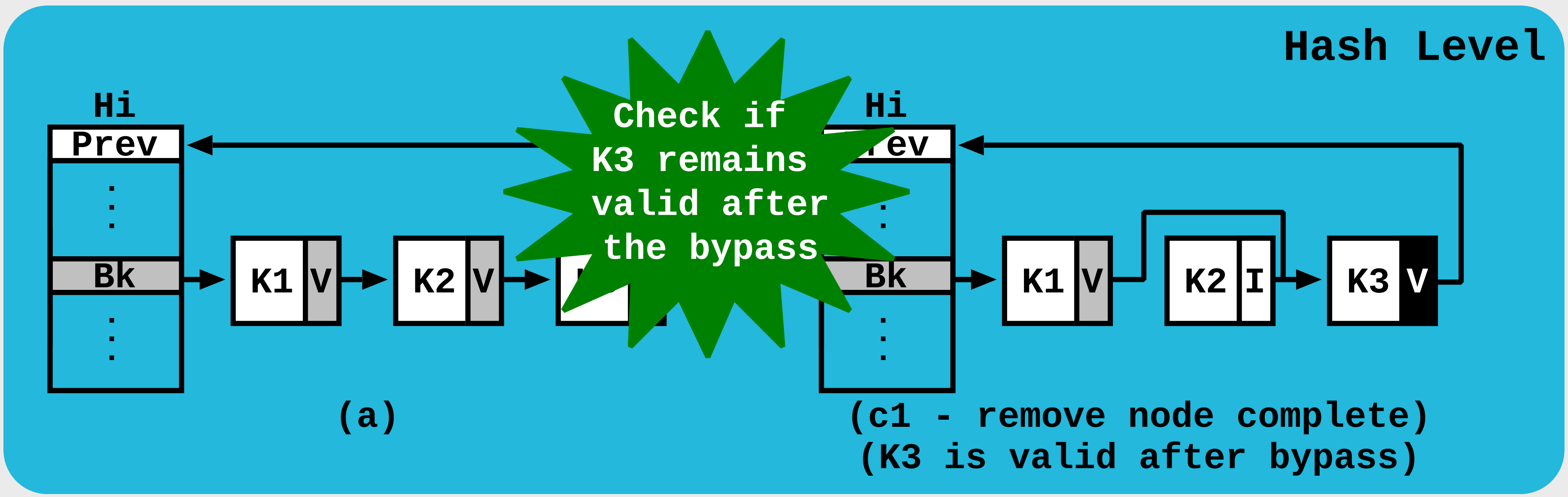
The FP Design - Remove

- The **remove operation** has two steps:
  - **Invalidate node** by **changing** its **state** from **valid** to **invalid**.
  - **Turn** the node **invisible** to all threads. Find **two valid** data structures (previous and next) and **bypass** the **invalid** node.



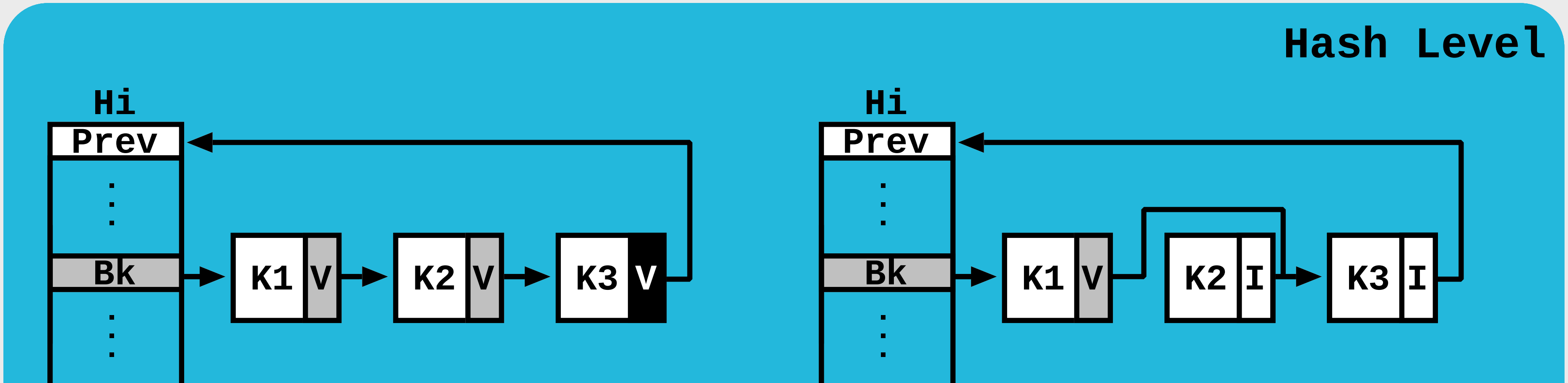
The FP Design - Remove

- The **remove operation** has two steps:
  - **Invalidate node** by **changing** its **state** from **valid** to **invalid**.
  - **Turn** the node **invisible** to all threads. Find **two valid** data structures (previous and next) and **bypass** the **invalid** node.



The FP Design - Remove

- The **remove operation** has two steps:
  - **Invalidate node** by **changing** its **state** from **valid** to **invalid**.
  - **Turn** the node **invisible** to all threads. Find **two valid** data structures (previous and next) and **bypass** the **invalid** node.

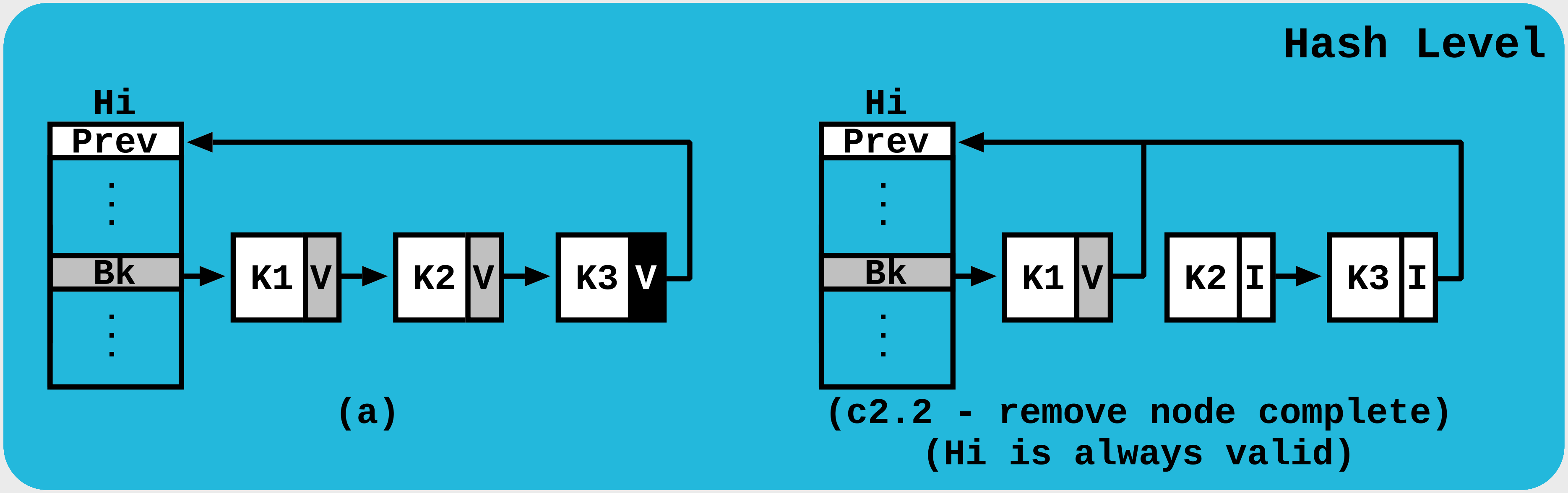


(a)

(c2.1 - remove node incomplete)
(K3 is invalid after bypass)

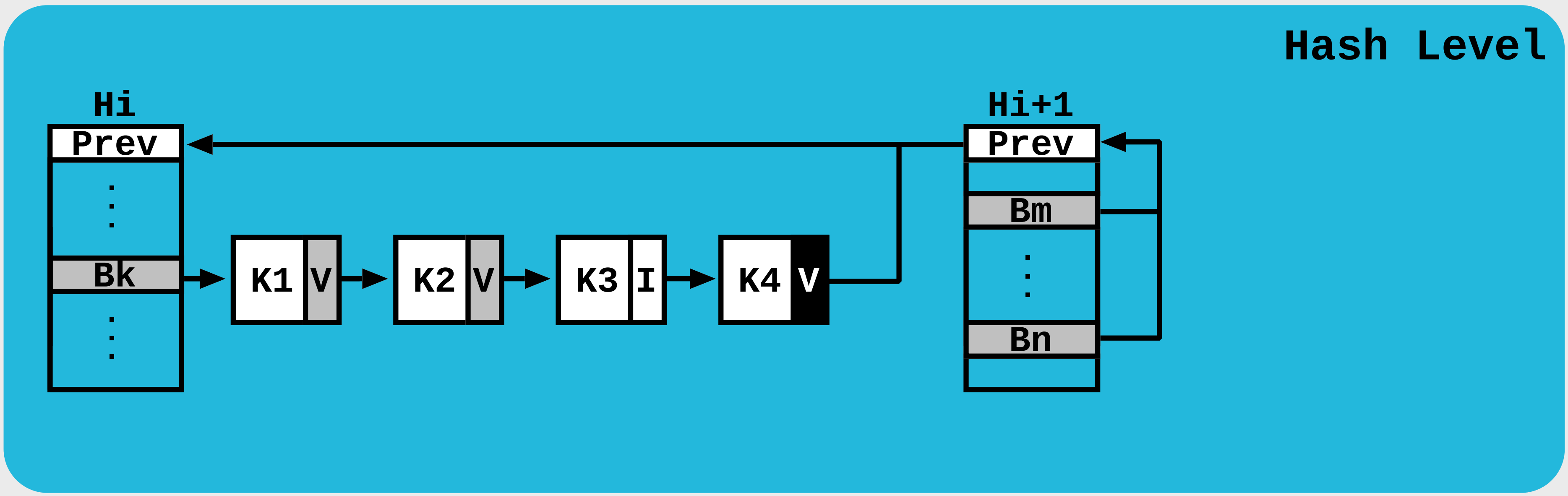
The FP Design - Remove

- The **remove operation** has two steps:
  - **Invalidate node** by **changing** its **state** from **valid** to **invalid**.
  - **Turn** the node **invisible** to all threads. Find **two valid** data structures (previous and next) and **bypass** the **invalid** node.



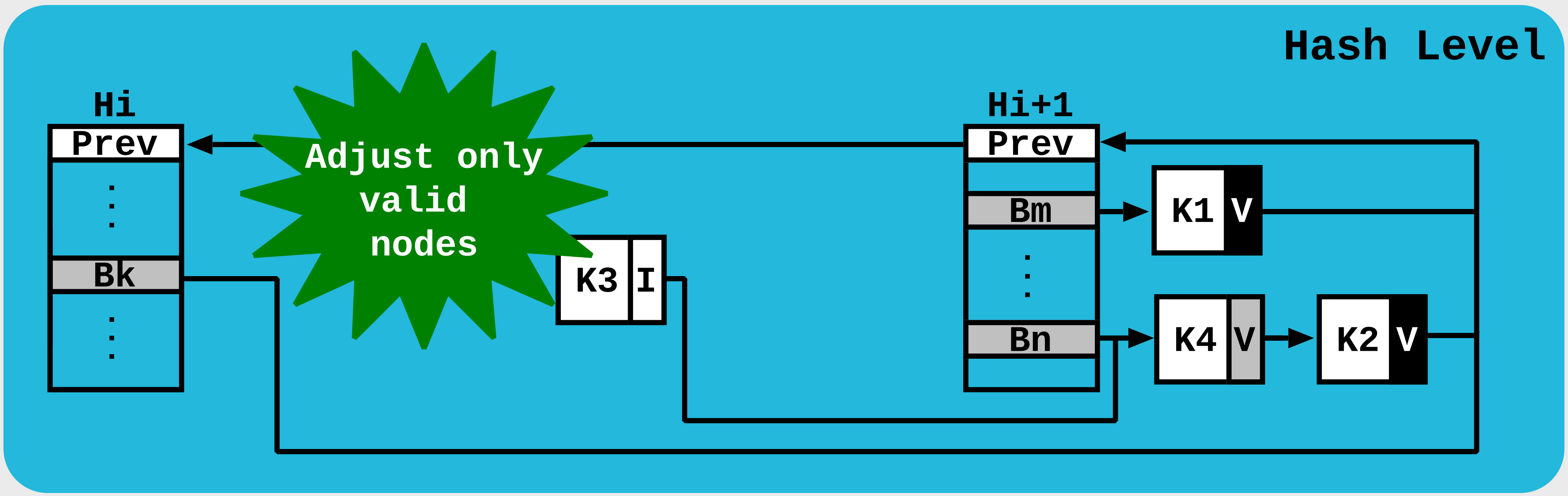
The FP Design - Expand (Valid and Invalid Nodes)

- The **expand operation** has two steps:
  - **Find** and **begin** the **expansion** in the **right-most (or deepest) valid node**.
  - **Adjust** only **valid nodes** on the **new hash** level. **Leave** the **invalid nodes** unchanged (it **allows** threads to **recover** to **valid** data structures).



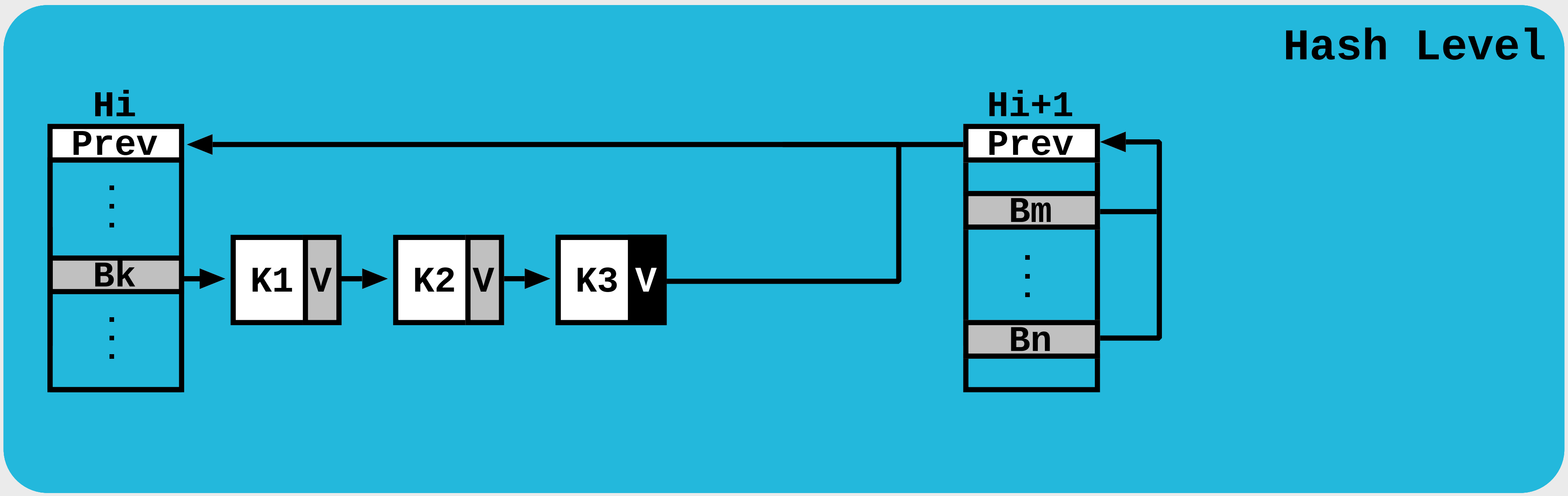
The FP Design - Expand (Valid and Invalid Nodes)

- The **expand operation** has two steps:
  - **Find** and **begin** the **expansion** in the **right-most (or deepest) valid node**.
  - **Adjust** only **valid nodes** on the **new hash** level. **Leave** the **invalid nodes** unchanged (it **allows** threads to **recover** to **valid** data structures).



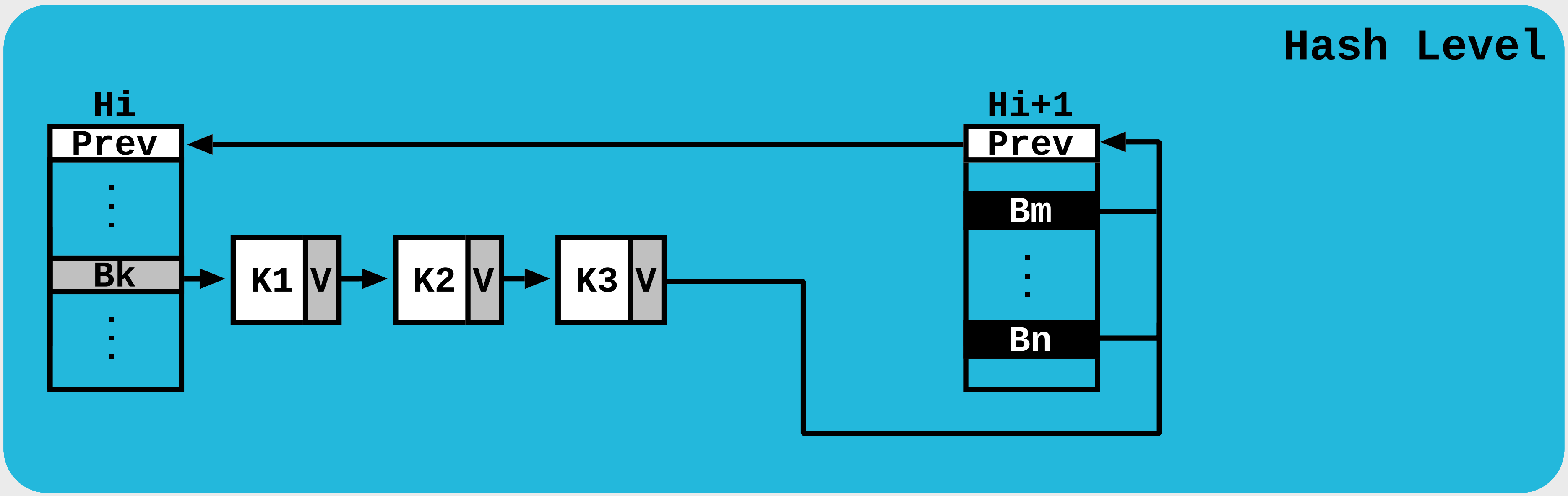
The FP Design - Expand (Valid and Invalid Nodes)

- The **expand operation** has two steps:
  - **Find** and **begin** the **expansion** in the **right-most (or deepest) valid node**.
  - **Adjust** only **valid nodes** on the **new hash** level. **Leave** the **invalid nodes** unchanged (it **allows** threads to **recover** to **valid** data structures).



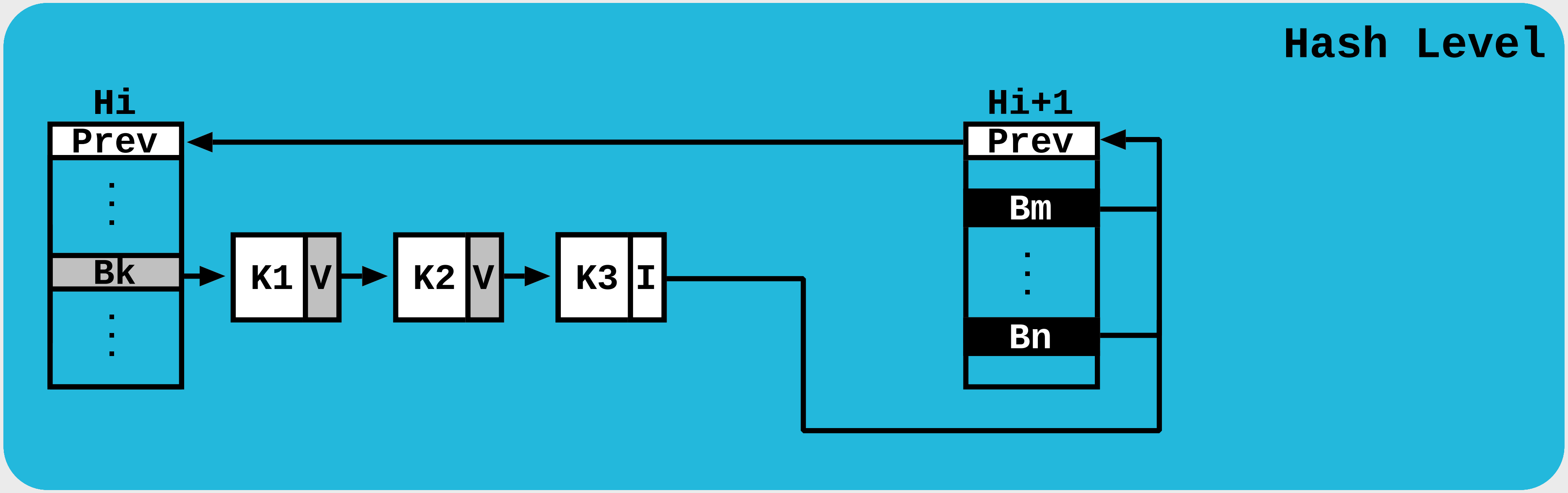
The FP Design - Expand (Valid and Invalid Nodes)

- The **expand operation** has two steps:
  - **Find** and **begin** the **expansion** in the **right-most (or deepest) valid node**.
  - **Adjust** only **valid nodes** on the **new hash** level. **Leave** the **invalid nodes** unchanged (it **allows** threads to **recover** to **valid** data structures).



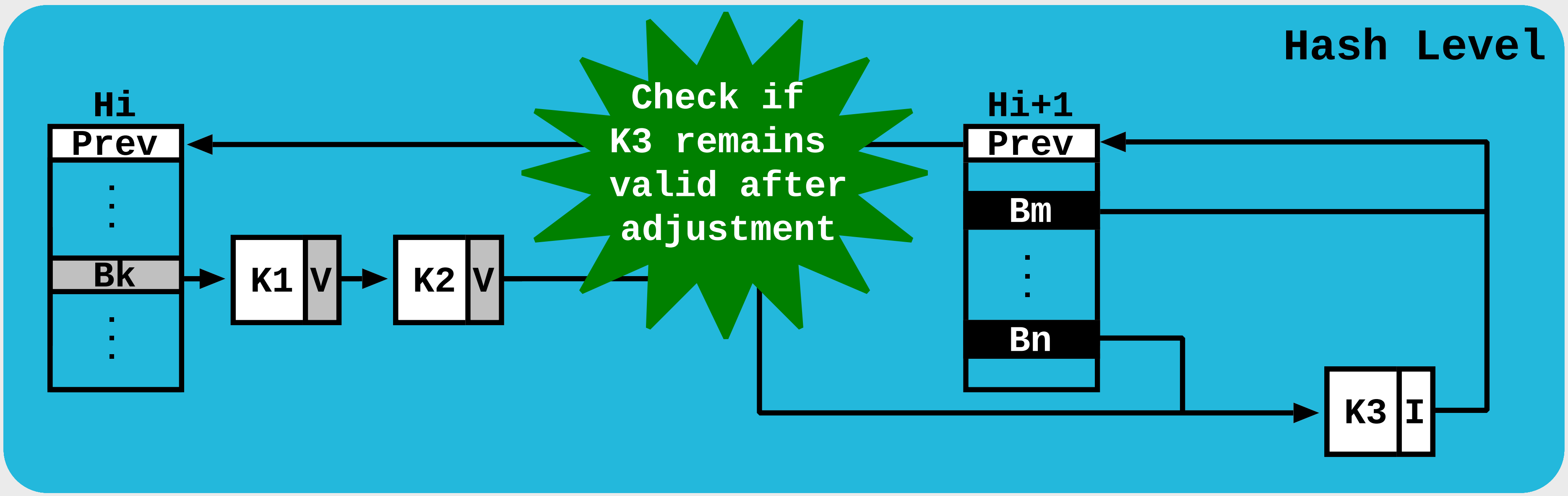
The FP Design - Expand (Valid and Invalid Nodes)

- The **expand operation** has two steps:
  - **Find** and **begin** the **expansion** in the **right-most (or deepest) valid node**.
  - **Adjust** only **valid nodes** on the **new hash** level. **Leave** the **invalid nodes** unchanged (it **allows** threads to **recover** to **valid** data structures).



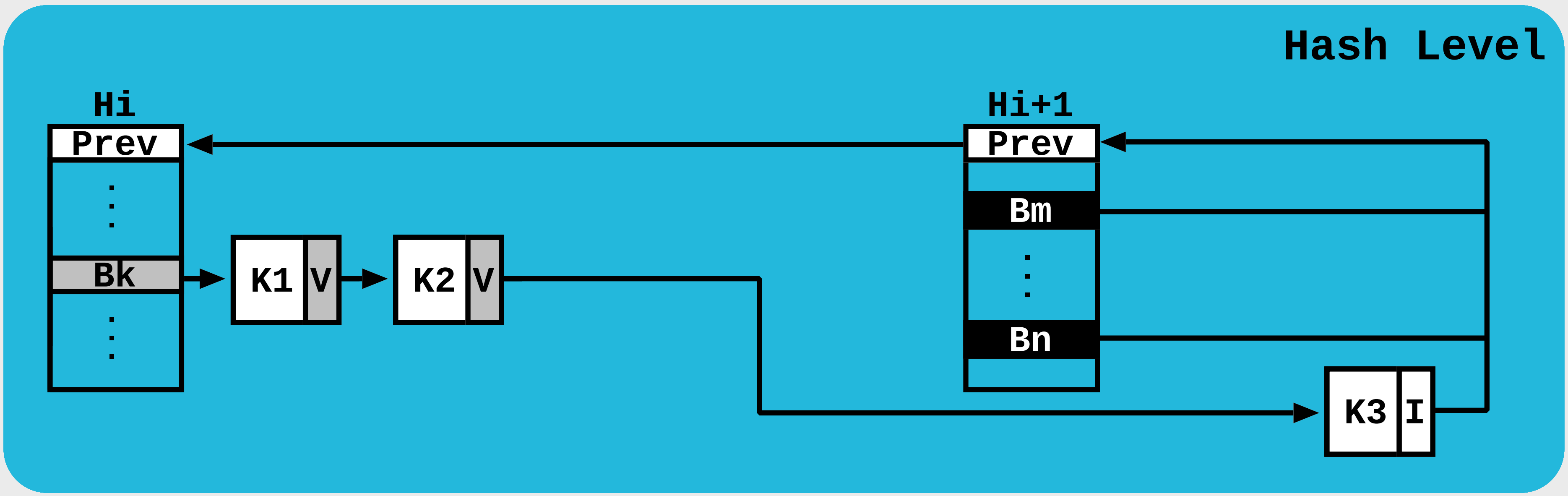
The FP Design - Expand (Valid and Invalid Nodes)

- The **expand operation** has two steps:
  - **Find** and **begin** the **expansion** in the **right-most (or deepest) valid node**.
  - **Adjust** only **valid nodes** on the **new hash** level. **Leave** the **invalid nodes** unchanged (it **allows** threads to **recover** to **valid** data structures).



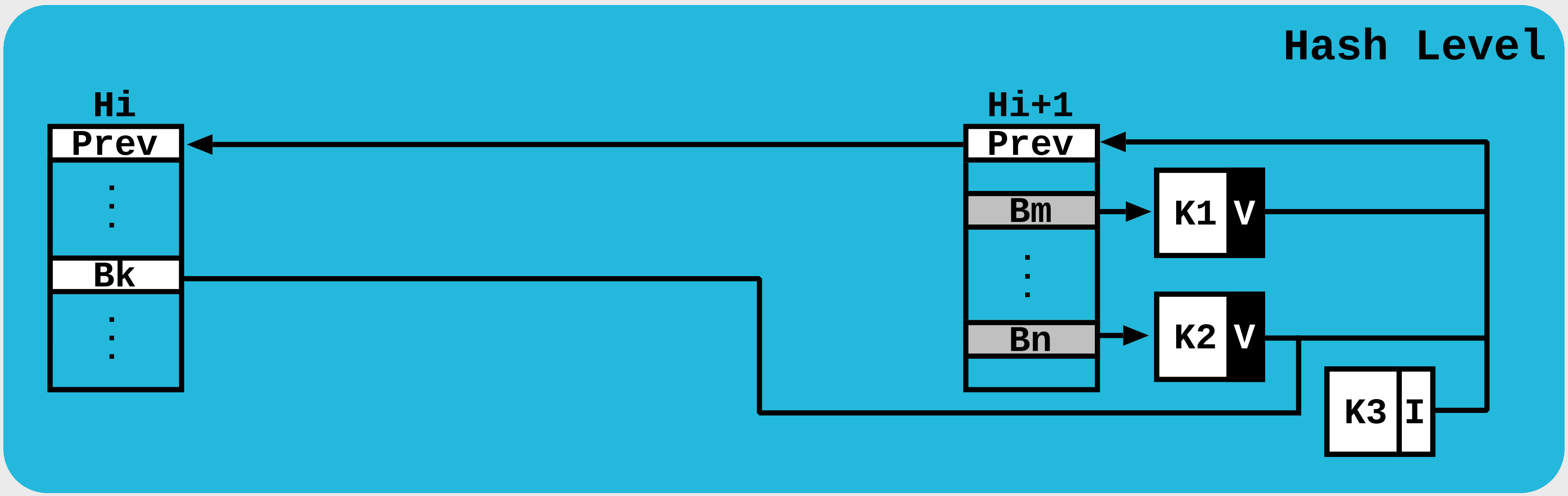
The FP Design - Expand (Valid and Invalid Nodes)

- The **expand operation** has two steps:
  - **Find** and **begin** the **expansion** in the **right-most (or deepest) valid node**.
  - **Adjust** only **valid nodes** on the **new hash** level. **Leave** the **invalid nodes** unchanged (it **allows** threads to **recover** to **valid** data structures).



The FP Design - Expand (Valid and Invalid Nodes)

- The **expand operation** has two steps:
  - **Find** and **begin** the **expansion** in the **right-most (or deepest) valid node**.
  - **Adjust** only **valid nodes** on the **new hash** level. **Leave** the **invalid nodes** unchanged (it **allows** threads to **recover** to **valid** data structures).



Performance Analysis

- **Hardware**: 32 (2 x 16) core **AMD** with 32 GB of main memory.
- **Software**: Linux **Fedora** 20 with **Oracle's Java Development Kit** 1.8.
- **Benchmarks**: Sets of 10^6 **randomized keys** with **insert**, **search** and **remove methods** (each **benchmark** had **5 warm up** runs and **20 standard** runs).
- **FP** design: **Expanded** with 6 **valid** nodes and had **two configurations** (**8 and 32 hash bucket** levels).

U.PORTO FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Performance Analysis

- In the next slides, we will be comparing the **FP** design against other state-of-the-art **hash map** designs that support efficiently **multithreading**: Concurrent Hash Maps (**CH**), Concurrent Skip Lists (**CS**), Non Blocking Hash Maps (**NB**) and Concurrent Tries (**CT**).
- **Podium** colors: **first place**, **second place** and **third place**.

Performance Analysis

➤ **Execution time** (**lower** is **better**) **Speedup Ratio** (**higher** is **better**).

# Threads (T_p)	Execution Time (E_{T_p}) CH	CS	NB	CT	FP_8	FP_{32}	Speedup Ratio (E_{T_1}/E_{T_p}) CH	CS	NB	CT	FP_8	FP_{32}
1st – Insert: 100% Search: 0% Remove: 0%												
1	**663**	3,238	12,968	**919**	946	**542**						
8	294	550	2,933	**207**	**174**	**176**	2.26	**5.89**	4.42	**4.44**	**5.44**	3.08
16	199	332	2,031	**118**	**117**	**124**	3.33	**9.75**	6.39	**7.79**	**8.09**	4.37
24	201	276	1,717	**107**	**96**	**153**	3.30	**11.73**	7.55	**8.59**	**9.85**	3.54
32	212	270	1,576	**97**	**89**	**74**	3.13	**11.99**	8.23	**9.47**	**10.63**	7.32
2nd – Insert: 0% Search: 100% Remove: 0%												
1	**155**	3,753	**225**	773	720	**379**						
8	**38**	535	**34**	120	118	**76**	4.08	**7.01**	**6.62**	**6.44**	6.10	4.99
16	**27**	327	**25**	78	76	**53**	5.74	**11.48**	9.00	**9.91**	**9.47**	7.15
24	**30**	309	**22**	70	64	**53**	5.17	**12.15**	10.23	**11.04**	**11.25**	7.15
32	**32**	315	**26**	78	69	**54**	4.84	**11.91**	8.65	**9.91**	**10.43**	7.02
3rd – Insert: 0% Search: 0% Remove: 100%												
1	**314**	4,144	**451**	1,585	872	**582**						
8	**105**	595	**122**	226	172	**137**	2.99	**6.96**	3.70	**7.01**	**5.07**	4.25
16	**62**	341	**77**	156	108	**89**	5.06	**12.15**	5.86	**10.16**	**8.07**	6.54
24	**55**	303	**66**	132	**94**	130	5.71	**13.68**	6.83	**12.01**	**9.28**	4.48
32	**54**	306	**64**	124	**101**	102	5.81	**13.54**	7.05	**12.78**	**8.63**	5.71

Performance Analysis

➤ **Execution time** (**lower** is **better**) **Speedup Ratio** (**higher** is **better**).

# Threads ($\mathbf{T_p}$)	Execution Time (E_{T_p}) CH	CS	NB	CT	FP_8	FP_{32}	Speedup Ratio (E_{T_1}/E_{T_p}) CH	CS	NB	CT	FP_8	FP_{32}
4th – Insert: 60%		**Search: 30%**		**Remove: 10%**								
1	721	2,510	15,342	1,027	873	618						
8	150	413	4,030	174	148	142	4.81	6.08	3.81	5.90	5.90	4.35
16	128	247	2,803	115	91	106	5.63	10.16	5.47	8.93	9.59	5.83
24	75	191	2,566	89	72	74	9.61	13.14	5.98	11.54	12.13	8.35
32	72	178	1,870	90	80	67	10.01	14.10	8.20	11.41	10.91	9.22
5th – Insert: 20%		**Search: 70%**		**Remove: 10%**								
1	282	1,890	12,370	764	757	395						
8	51	282	8,517	171	157	74	5.53	6.70	1.45	4.47	4.82	5.34
16	39	184	3,623	87	72	82	7.23	10.27	3.41	8.78	10.51	4.82
24	37	143	3,058	73	69	64	7.62	13.22	4.05	10.47	10.97	6.17
32	38	145	2,081	74	69	65	7.42	13.03	5.94	10.32	10.97	6.08
6th – Insert: 25%		**Search: 50%**		**Remove: 25%**								
1	279	2,059	12,181	1,087	808	440						
8	113	340	3,125	159	127	83	2.47	6.06	3.90	6.84	6.36	5.30
16	64	214	3,482	104	82	70	4.36	9.62	3.50	10.45	9.85	6.29
24	42	180	2,609	87	71	78	6.64	11.44	4.67	12.49	11.38	5.64
32	44	166	1,902	83	77	66	6.34	12.40	6.40	13.10	10.49	6.67

U.PORTO
FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Thank You !!!

Miguel Areias
miguel-areias@dcc.fc.up.pt

FP design : *https://github.com/miar/ffp*
FCT grant: *SFRH/BPD/108018/2015*

Parallel Computing