Towards a Lock-Free, Fixed-Size and Persistent Hash Map Design

Miguel Areias and Ricardo Rocha
CRACS & INESC-TEC LA
University of Porto, Portugal

SBAC-PAD

Concurrent 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**.
  - If **K** is **unique** then **K uniquely identifies** each **content**.
  - Some of the most **usual operations** are:
    - **User-level** (externally activated by users) : **search**, **insert** and **remove**.
    - **Kernel-level** (internally activated by thresholds): **expansion** (and **compression**, which will not be discussed in this talk).

Concurrent 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**.
  - If **K** is **unique** then **K uniquely identifies** each **content**.
  - Some of the most **usual operations** are:
    - **User-level** (externally activated by users) : **search**, **insert** and **remove**.
    - **Kernel-level** (internally activated by thresholds): **expansion** (and **compression**, which will not be discussed in this talk).

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

Our Motivation

- There are **several hash map designs** that already support efficiently **multithreading**: Concurrent Hash Maps (**CH**), Concurrent Skip Lists (**CS**), Non Blocking Hash Maps (**NB**) and Concurrent Tries (**CT**).

Our Motivation

- There are **several hash map designs** that already support efficiently **multithreading**: Concurrent Hash Maps (**CH**), Concurrent Skip Lists (**CS**), Non Blocking Hash Maps (**NB**) and Concurrent Tries (**CT**).
- However, to the best of our knowledge, **non** of the **existent designs** combine three **properties**: **Lock-Free Progress**, **Persistent Memory References** and **Fixed-Size Data Structures**.

Properties / Designs	CH	CS	NB	CT
Lock-Free Progress	✗	✗	✓	✓
Persistent Memory References	✗	✓	✓	✗
Fixed-Size Data Structures	✗	-	✗	✓

Our Motivation

- There are **several hash map designs** that already support efficiently **multithreading**: Concurrent Hash Maps (**CH**), Concurrent Skip Lists (**CS**), Non Blocking Hash Maps (**NB**) and Concurrent Tries (**CT**).
- However, to the best of our knowledge, **non** of the **existent designs** combine three **properties**: **Lock-Free Progress**, **Persistent Memory References** and **Fixed-Size Data Structures**.

Properties / Designs	CH	CS	NB	CT
Lock-Free Progress	✗	✗	✓	✓
Persistent Memory References	✗	✓	✓	✗
Fixed-Size Data Structures	✗	-	✗	✓

- In **this talk** we will:
  - explain why these **properties** are **important** in some **domains**.

Our Motivation

- There are **several hash map designs** that already support efficiently **multithreading**: Concurrent Hash Maps (**CH**), Concurrent Skip Lists (**CS**), Non Blocking Hash Maps (**NB**) and Concurrent Tries (**CT**).
- However, to the best of our knowledge, **non** of the **existent designs** combine three **properties**: **Lock-Free Progress**, **Persistent Memory References** and **Fixed-Size Data Structures**.

Properties / Designs	CH	CS	NB	CT	FP
Lock-Free Progress	✗	✗	✓	✓	✓
Persistent Memory References	✗	✓	✓	✗	✓
Fixed-Size Data Structures	✗	-	✗	✓	✓

- In **this talk** we will:
  - explain why these **properties** are **important** in some **domains**.
  - present a new design (**FP**) that **supports** those three **properties**.

Our Motivation

- There are **several hash map designs** that already support efficiently **multithreading**: Concurrent Hash Maps (**CH**), Concurrent Skip Lists (**CS**), Non Blocking Hash Maps (**NB**) and Concurrent Tries (**CT**).
- However, to the best of our knowledge, **non** of the **existent designs** combine three **properties**: **Lock-Free Progress**, **Persistent Memory References** and **Fixed-Size Data Structures**.

Properties / Designs	CH	CS	NB	CT	FP
Lock-Free Progress	✗	✗	✓	✓	✓
Persistent Memory References	✗	✓	✓	✗	✓
Fixed-Size Data Structures	✗	-	✗	✓	✓

- In **this talk** we will:
  - explain why these **properties** are **important** in some **domains**.
  - present a new design (**FP**) that **supports** those three **properties**.
  - show a **performance** analysis **comparison** between **all designs**.

Property 1: Lock-Free Progress

- **Lock-Free linearizable objects** permit a **greater concurrency** since **semantically consistent** (non-interfering) operations **may execute in parallel**.
- **Lock-Free** techniques **do not use** traditional **locking** mechanisms.
  - **Avoid problems** such as **deadlocks** (threads **delaying** each other **perpetually**) and **convoying** (a thread holding a lock is **descheduled** by an **interrupt**).

Property 1: Lock-Free Progress

- **Lock-Free linearizable objects** permit a **greater concurrency** since **semantically consistent** (non-interfering) operations **may execute in parallel**.
- **Lock-Free** techniques **do not use** traditional **locking** mechanisms.
  - **Avoid problems** such as **deadlocks** (threads **delaying** each other **perpetually**) and **convoying** (a thread holding a lock is **descheduled** by an **interrupt**).
- 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**).

Property 1: Lock-Free Progress

- **Lock-Free linearizable objects** permit a **greater concurrency** since **semantically consistent** (non-interfering) operations **may execute in parallel**.
- **Lock-Free** techniques **do not use** traditional **locking** mechanisms.
  - **Avoid problems** such as **deadlocks** (threads **delaying** each other **perpetually**) and **convoying** (a thread holding a lock is **descheduled** by an **interrupt**).
- 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.

Property 2: Persistent Memory References

- The **expand** operation is crucial for a **hash map** to maintain an efficient access to **pairs**.
  - Occurs whenever the **internal hashing** data structures become **saturated** due to multiple **key collisions**.

Property 2: Persistent Memory References

- The **expand** operation is crucial for a **hash map** to maintain an efficient access to **pairs**.
  - Occurs whenever the **internal hashing** data structures become **saturated** due to multiple **key collisions**.
  - Some **hash map** designs use **disposable memory references**.
    - Up on expansion, some pairs are **copied** from the **old** memory references to **new** memory references.
    - Pairs are then **rehashed** using the **new** memory references.
    - When the **expansion** is complete, the **old** references are **discarded**.

Property 2: Persistent Memory References

- The **expand** operation is crucial for a **hash map** to maintain an efficient access to **pairs**.
  - Occurs whenever the **internal hashing** data structures become **saturated** due to multiple **key collisions**.
  - Some **hash map** designs use **disposable memory references**.
    - Up on expansion, some pairs are **copied** from the **old** memory references to **new** memory references.
    - Pairs are then **rehashed** using the **new** memory references.
    - When the **expansion** is complete, the **old** references are **discarded**.

- The **Persistent Memory References** property consists in not **copying pairs** and not using **disposable memory references**.
  - A **pair** remains **always** in the **same** memory references.
  - The memory references **persist** with the **pair** until it is either **removed** or the **hash map** dies.

Property 3: Fixed-Size Data Structures

- On the **expand** operation, some **hash map** designs **duplicate** the size of their **internal hashing** data structures.
- The **Fixed-Size Data Structures** property consists in using **always** data structures of the **same size**.

Property 3: Fixed-Size Data Structures

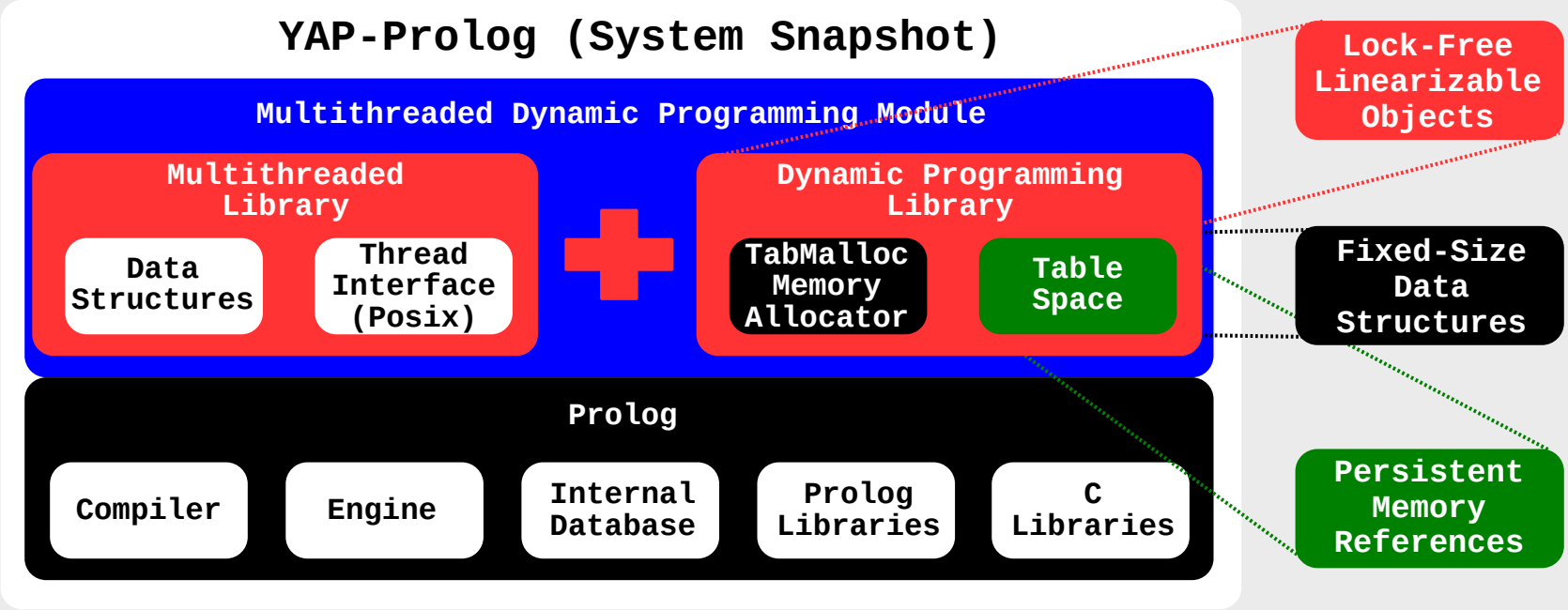
- On the **expand** operation, some **hash map** designs **duplicate** the size of their **internal hashing** data structures.
- The **Fixed-Size Data Structures** property consists in using **always** data structures of the **same size**.
- In **ICPADS'12** we presented the **TabMalloc**, a **user-level page-based concurrent memory allocator**.
  - Data structures of the same type (and consequently of the same size) are **pre-allocated** within pages.

Property 3: Fixed-Size Data Structures

- On the **expand** operation, some **hash map** designs **duplicate** the size of their **internal hashing** data structures.
- The **Fixed-Size Data Structures** property consists in using **always** data structures of the **same size**.
- In **ICPADS'12** we presented the **TabMalloc**, a **user-level page-based concurrent memory allocator**.
  - Data structures of the same type (and consequently of the same size) are **pre-allocated** within pages.
- The **Fixed-Size Data Structures** property allows an **efficient** usage of **TabMalloc** (**page-based**), because it **knows beforehand** the type of data structures that the **hash map** will use.

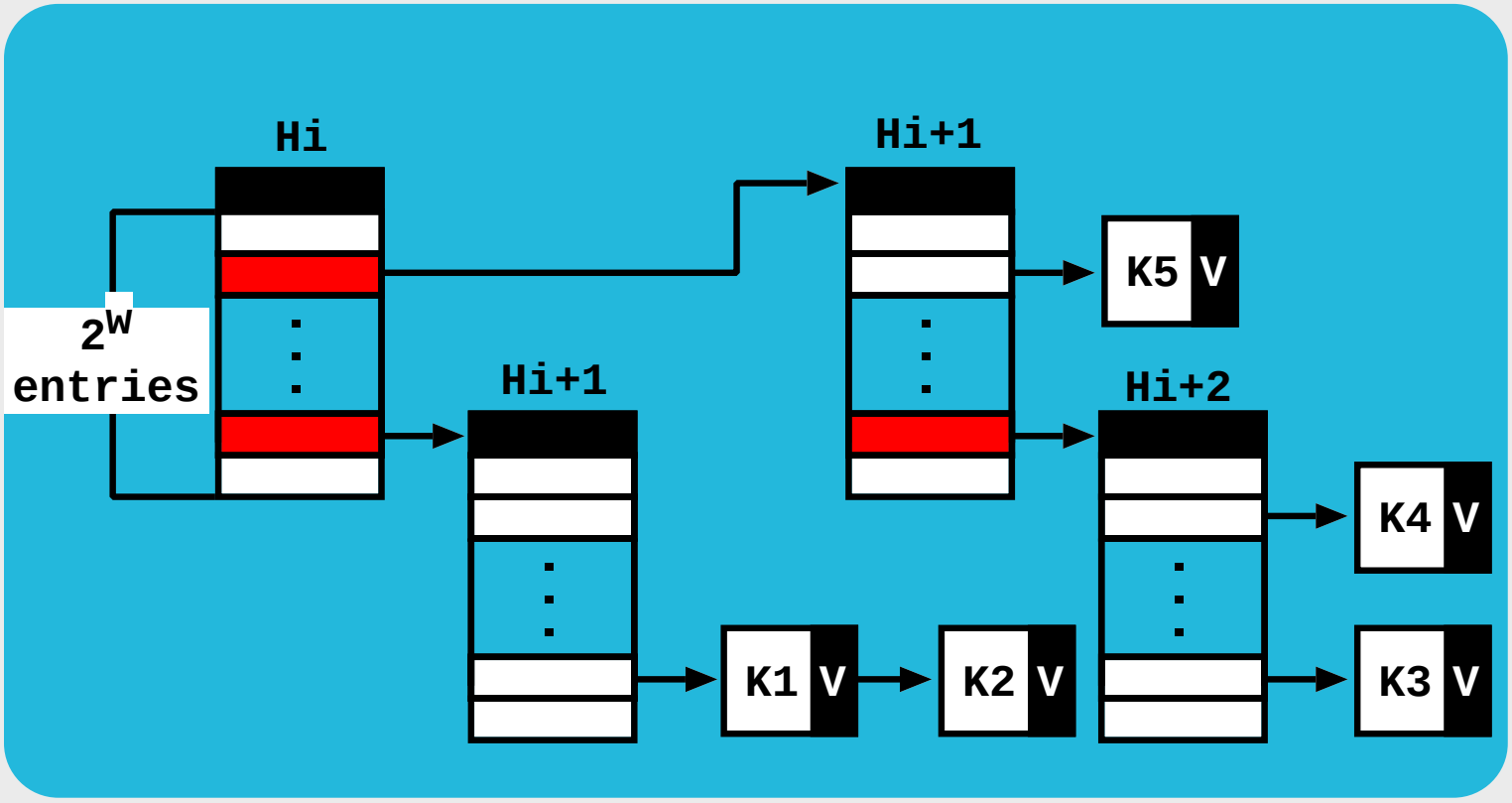
Properties 1, 2 and 3: Real-World Example

- These properties are **useful** when the **internals** of a **hash map** are directly **accessed** by an external system.
- For example a **Prolog** system with a **dynamic programming** library that **stores** uniquely identifiable **computations** and their **answers** (**hash maps** are used within the **Table Space** component).

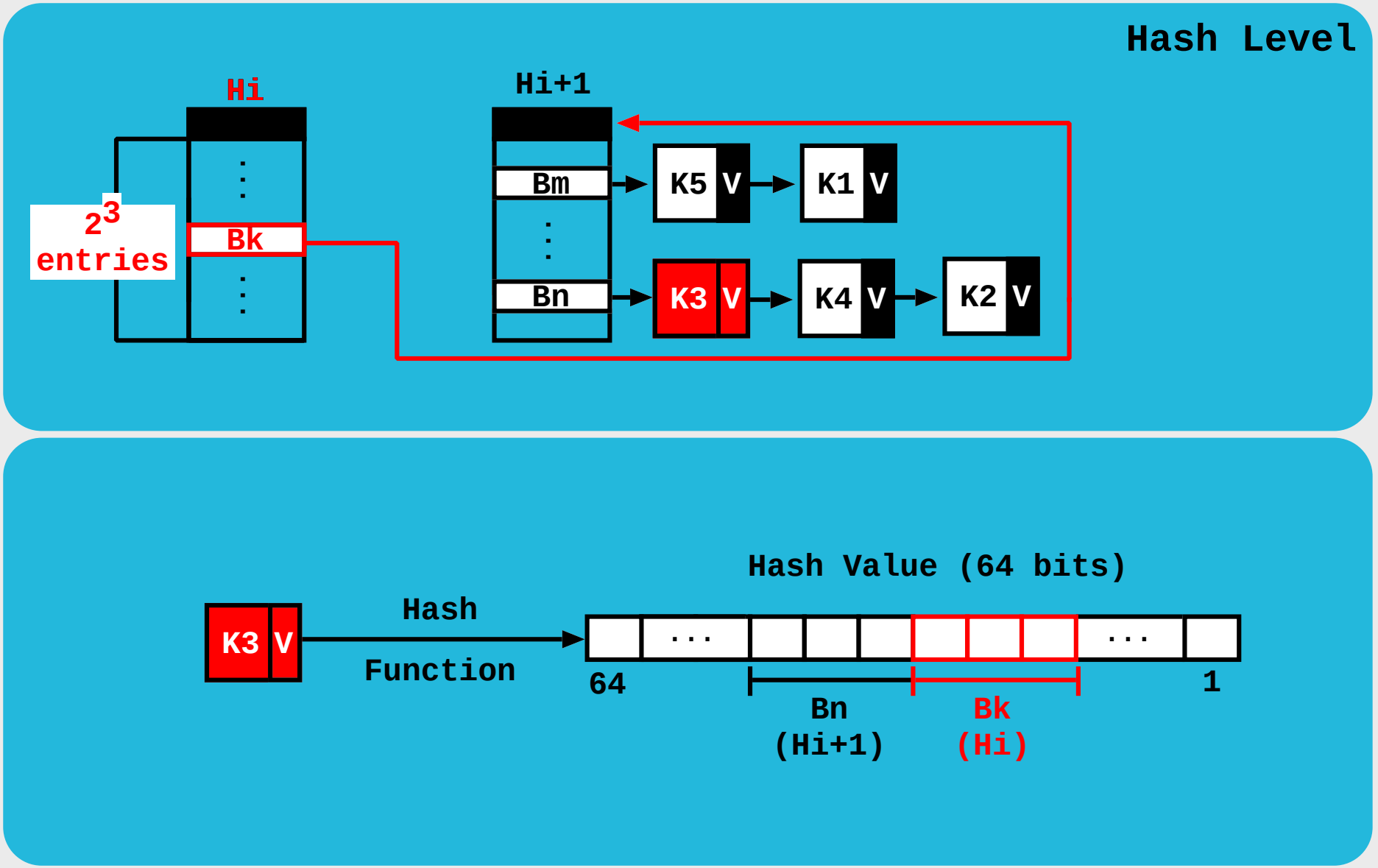


The FP Design - Hash Trie Structure

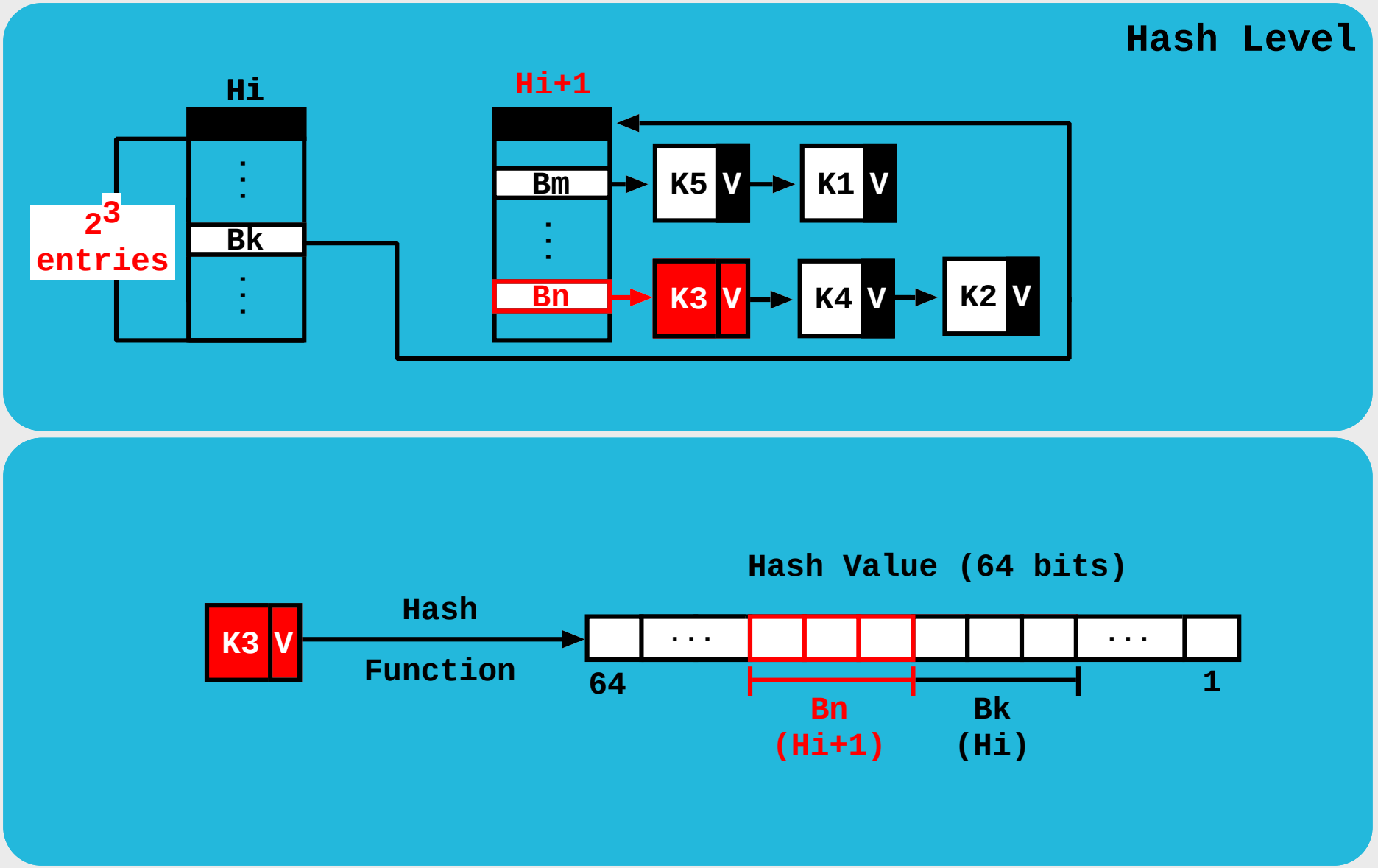
- **Hash buckets** refer to a **chaining mechanism** that supports **key collisions**.
- **Chain nodes** store **pairs (Key, Content, (Next_On_Chain, State))**. For the **sake of simplicity** we will present only **(Key, (Next_On_Chain, State))**. **State** can be **valid (V)** or **invalid (I)**.



The FP Design - Searching for K3



The FP Design - Searching for K3

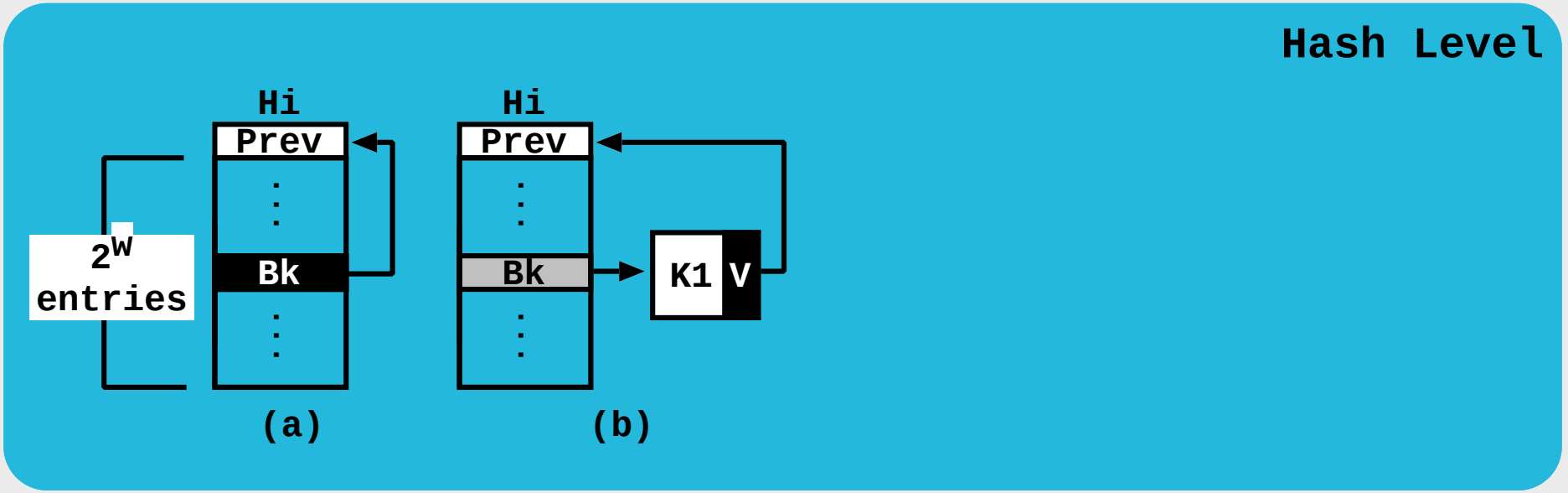


The FP Design - Internals

- 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.

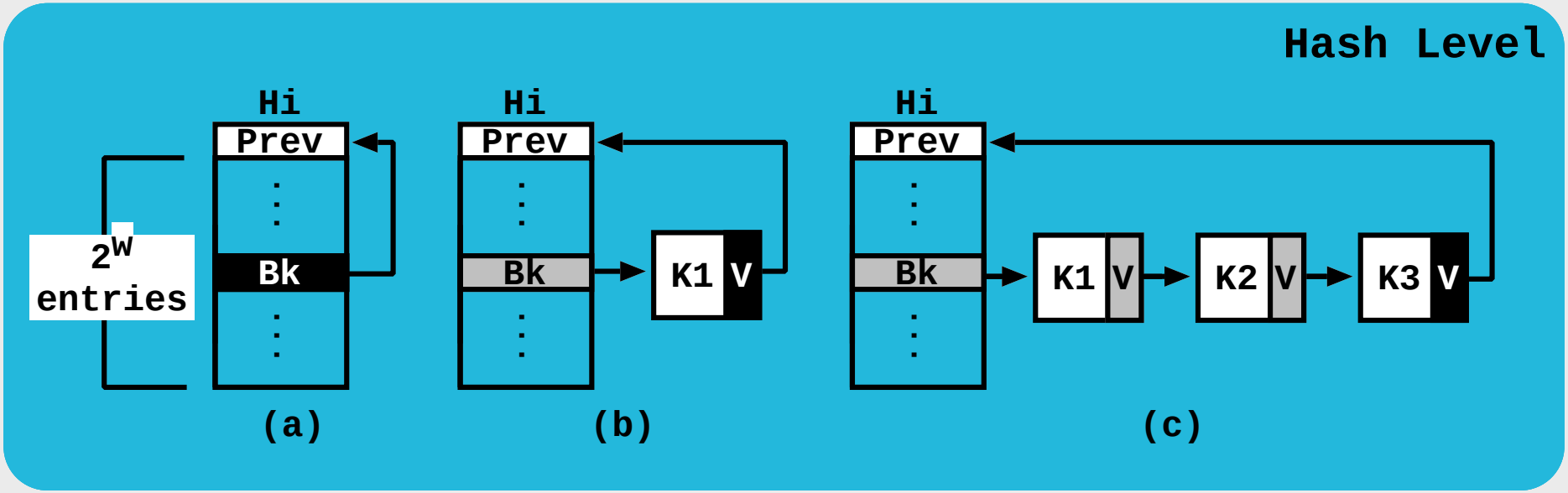
The FP Design - Internals

- 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.

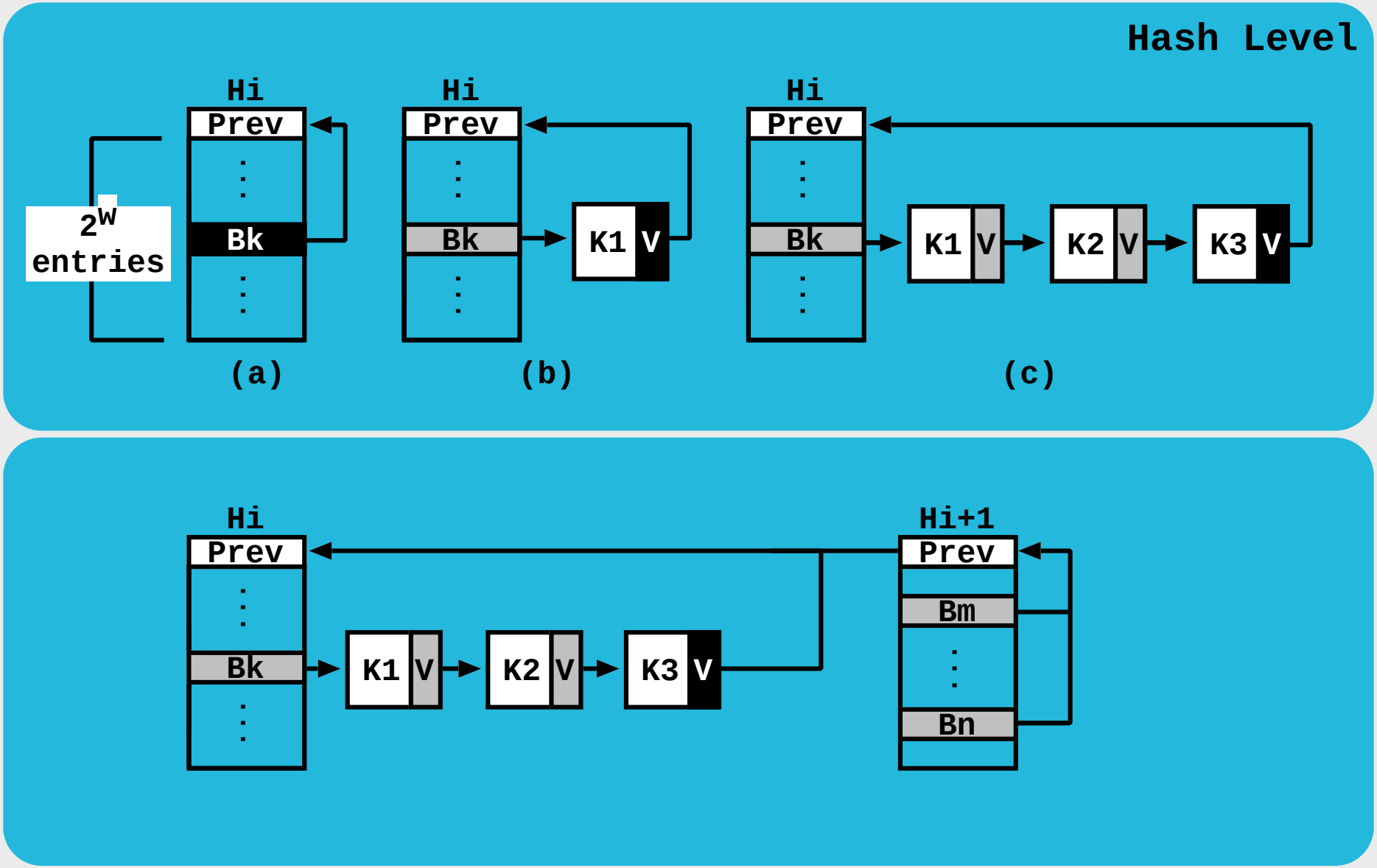


The FP Design - Internals

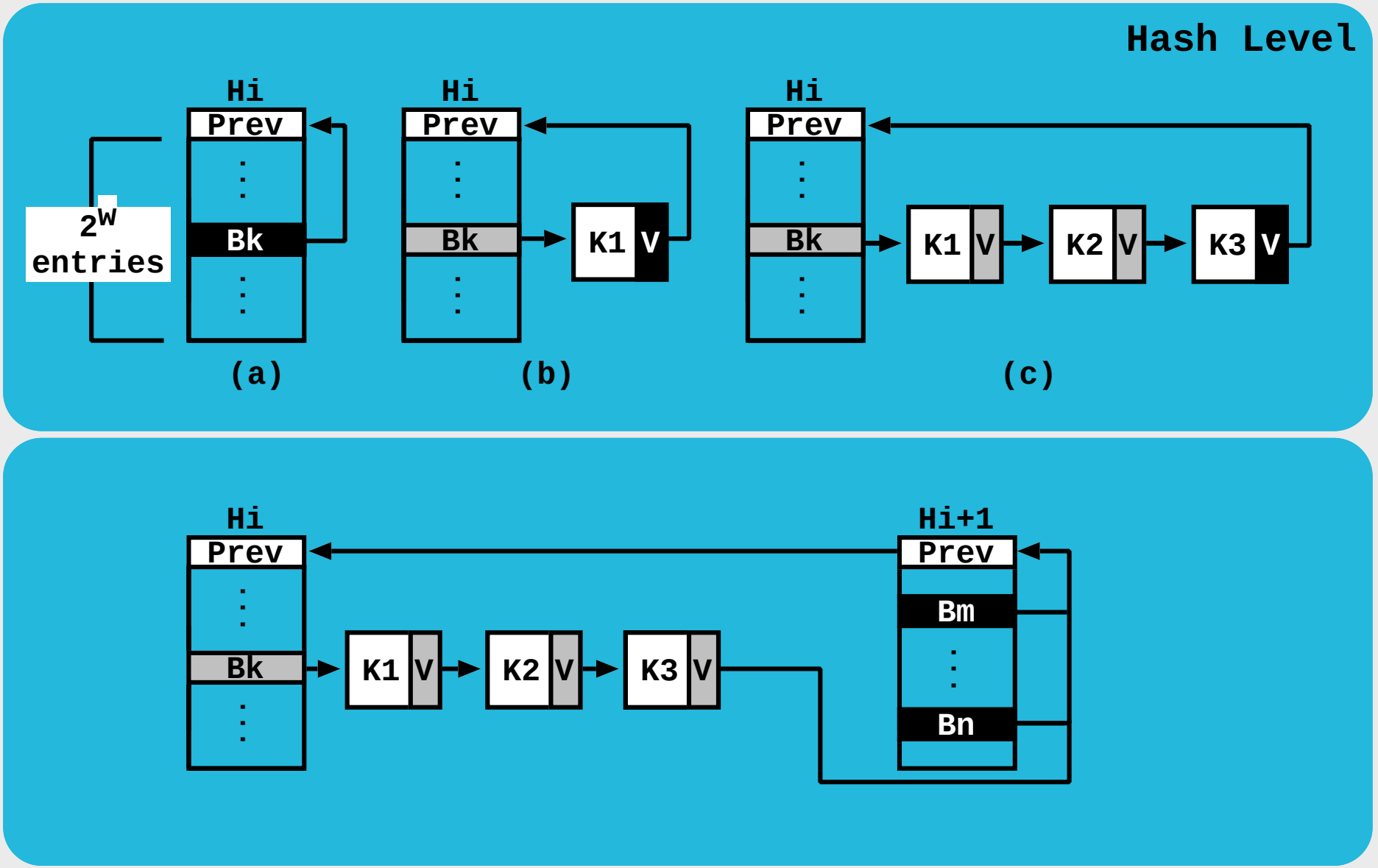
- To support **multithreading**, our design allows **threads** to:
  - **Recover from preemption**, by using a **Prev** field to traverse the **hash buckets** backwards.
  - **Identify chains**, by using a **back-reference** on the end of each chain.
  - **Maintain consistency**, by using **CAS** on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



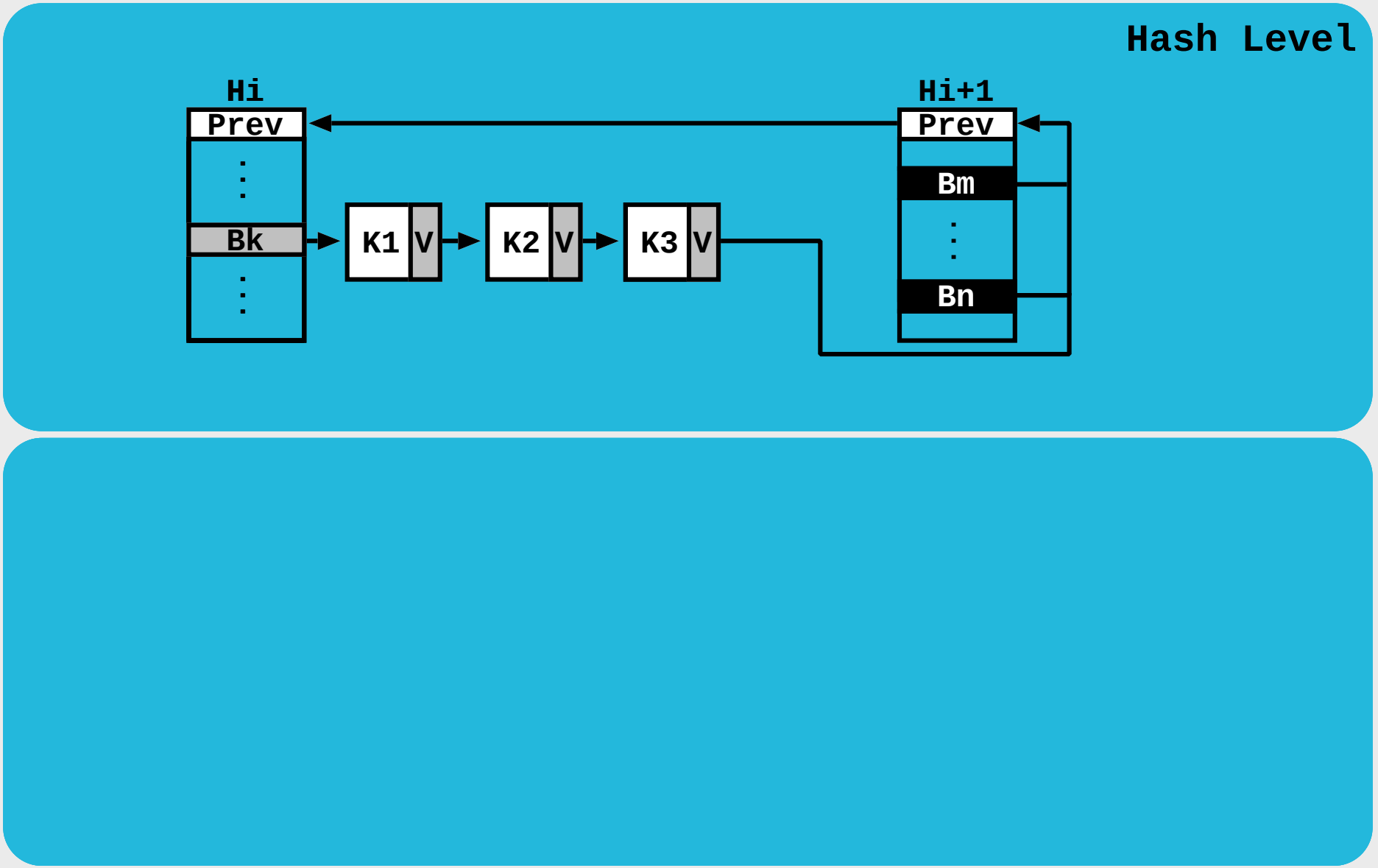
The FP Design - Expand (Valid Nodes Only)



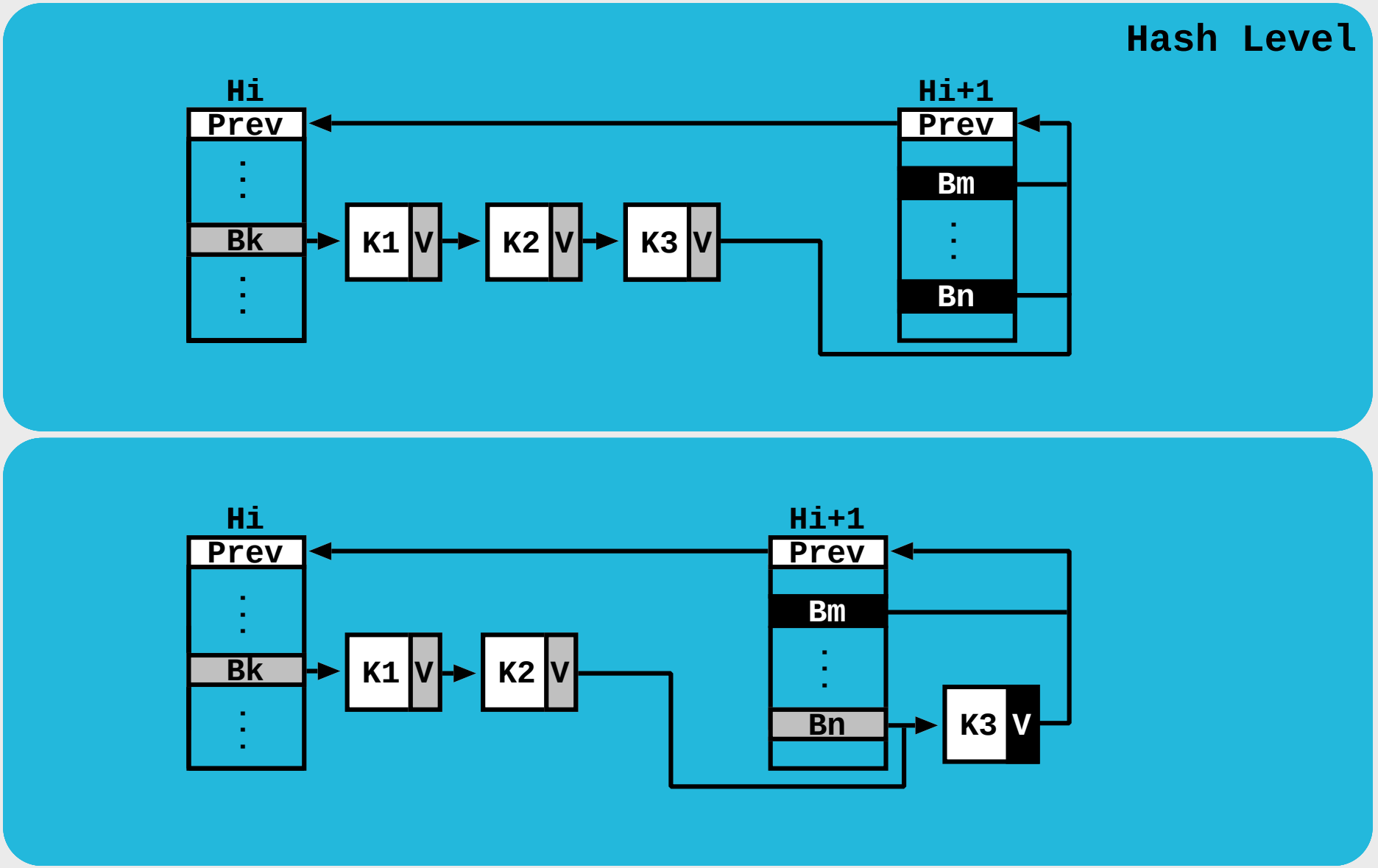
The FP Design - Expand (Valid Nodes Only)



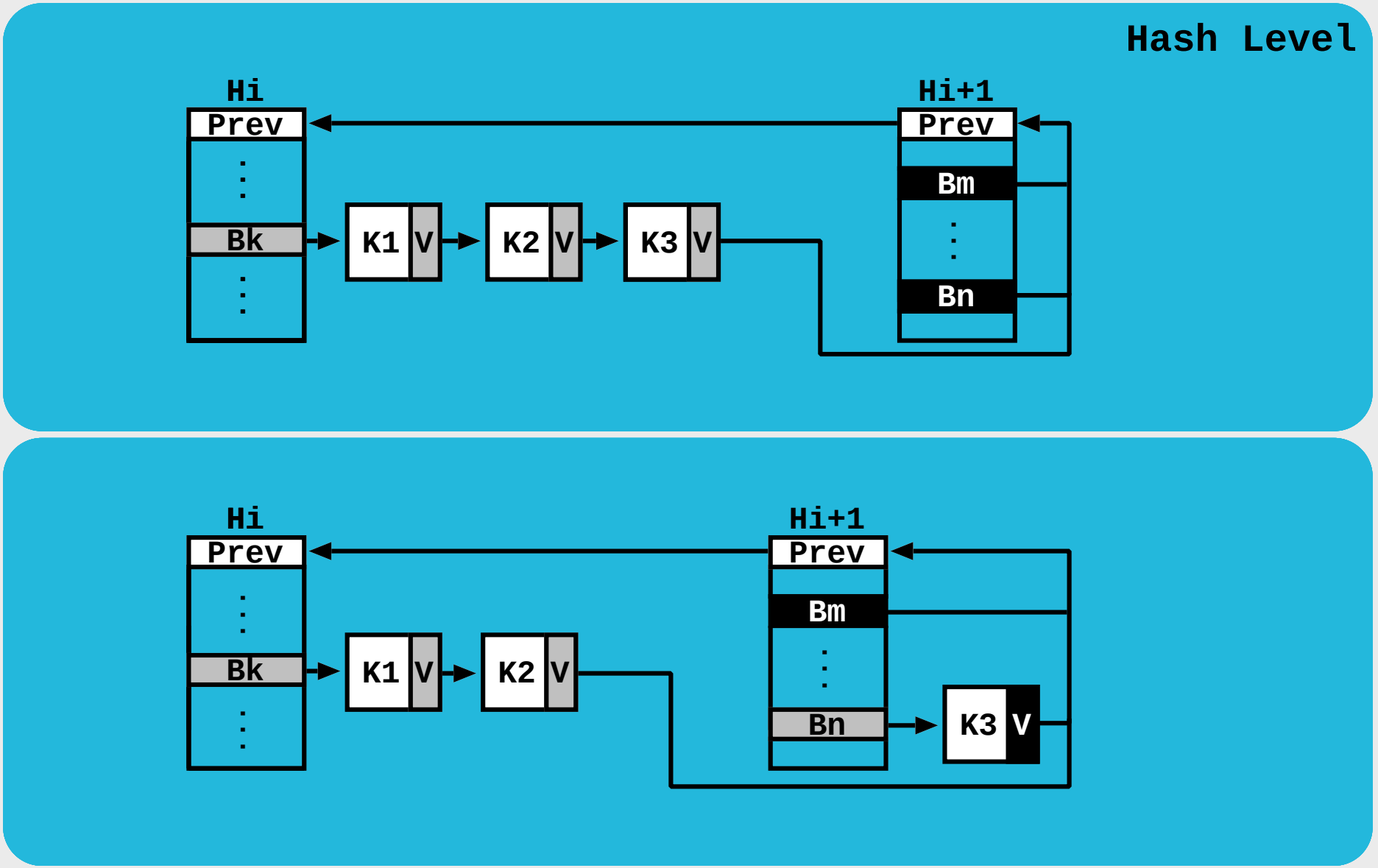
The FP Design - Expand (Valid Nodes Only)



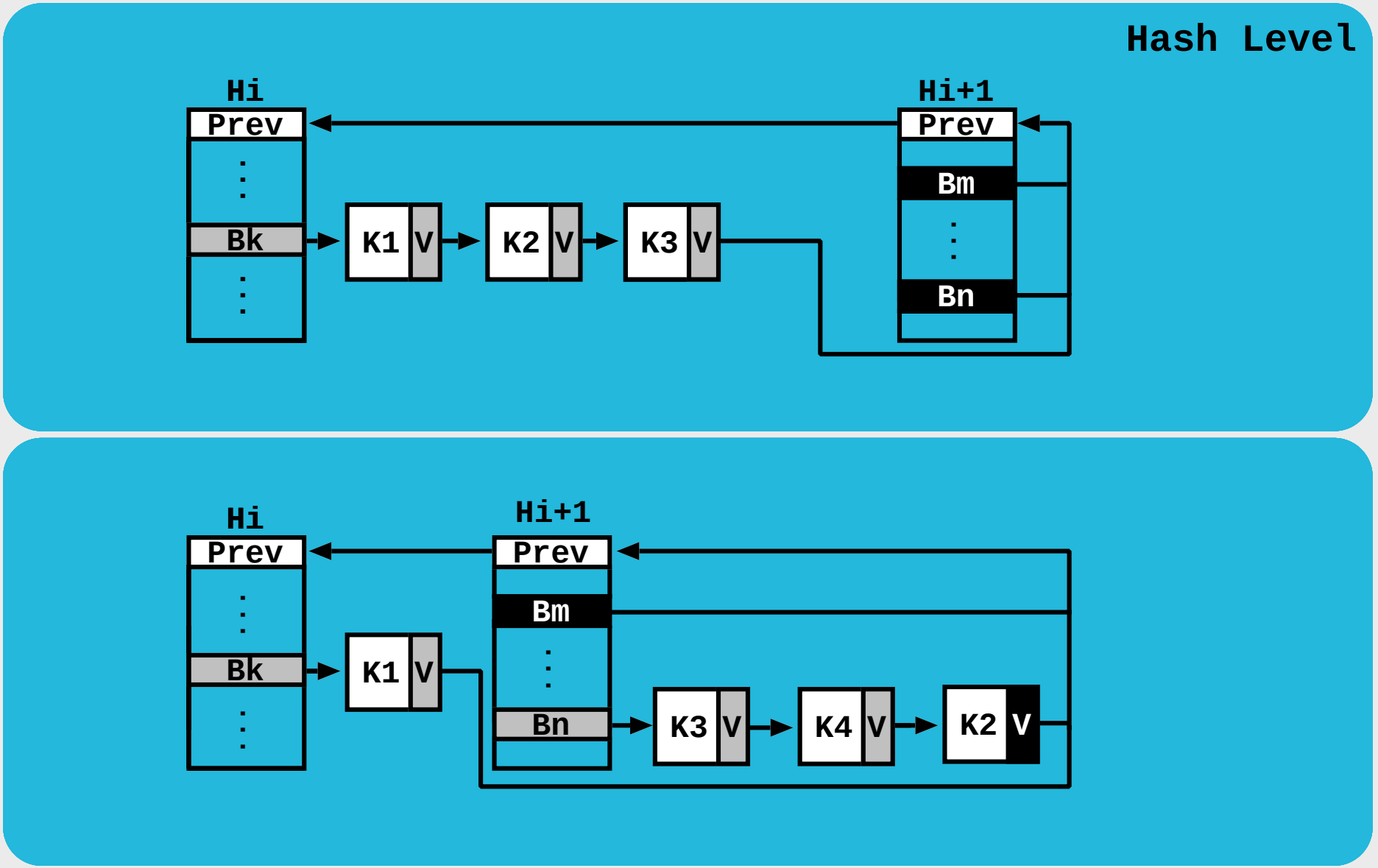
The FP Design - Expand (Valid Nodes Only)



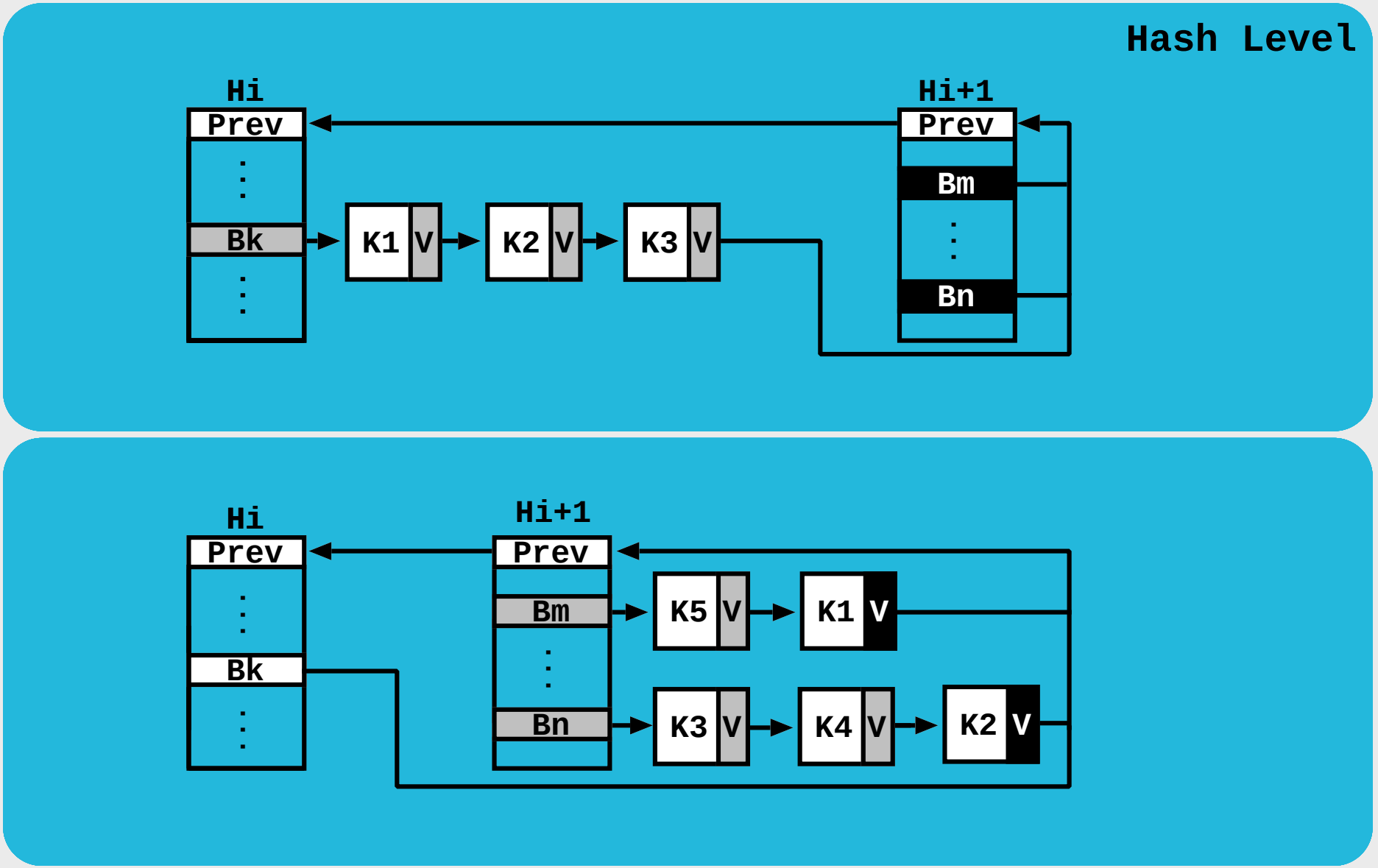
The FP Design - Expand (Valid Nodes Only)



The FP Design - Expand (Valid Nodes Only)

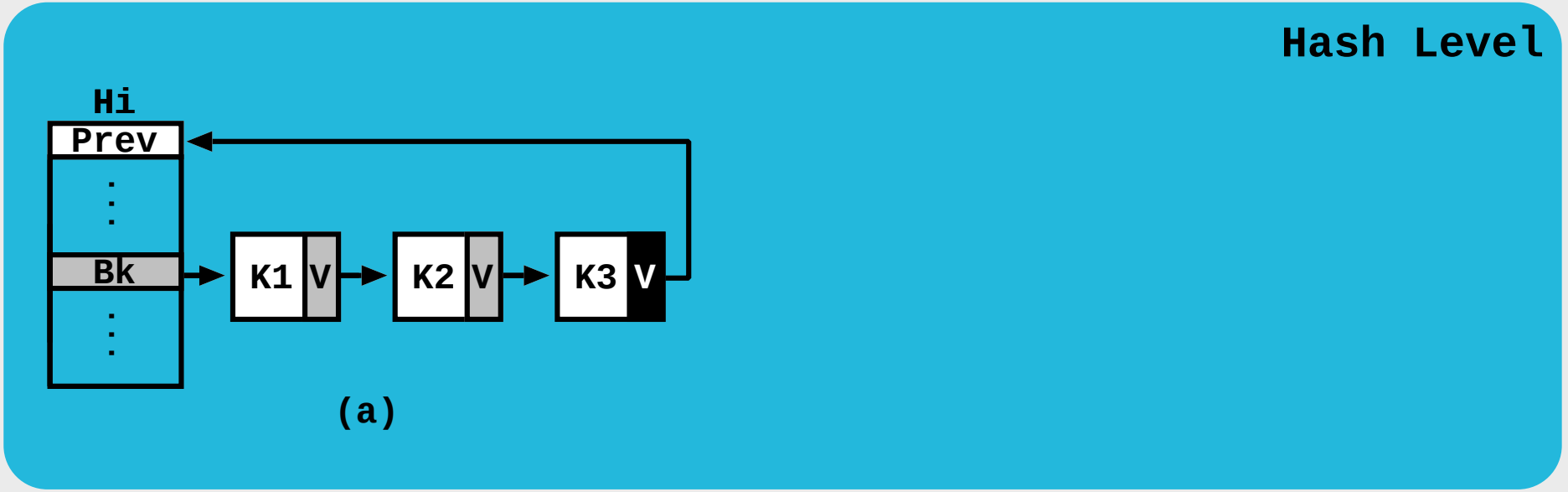


The FP Design - Expand (Valid Nodes Only)



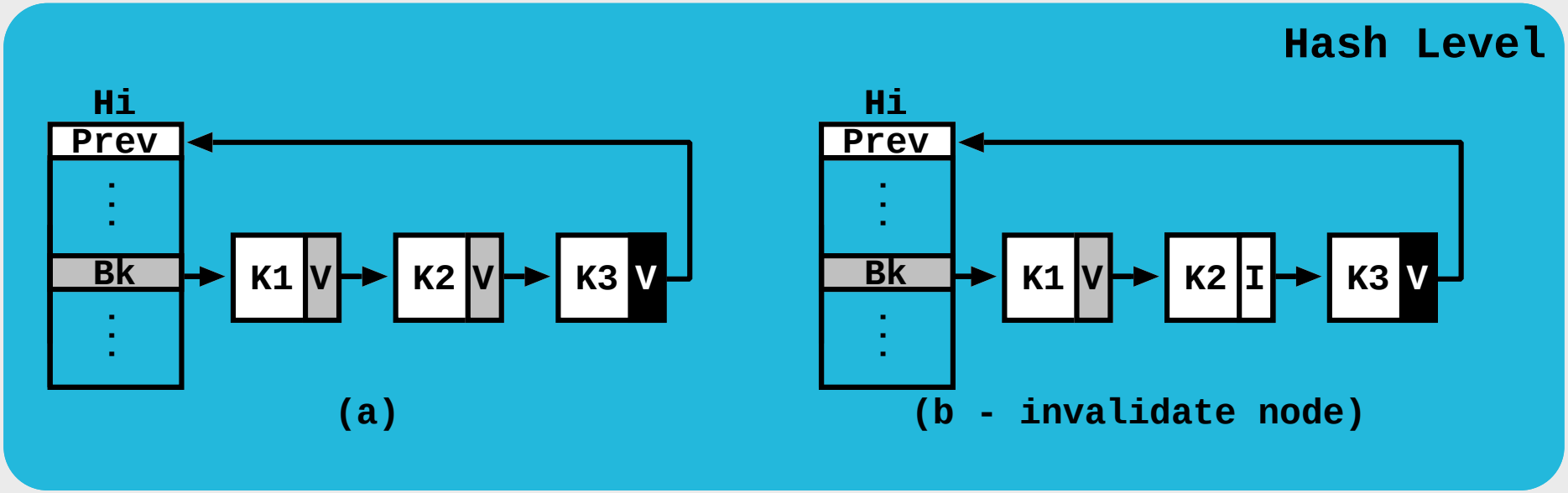
The FP Design - Remove

- 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.



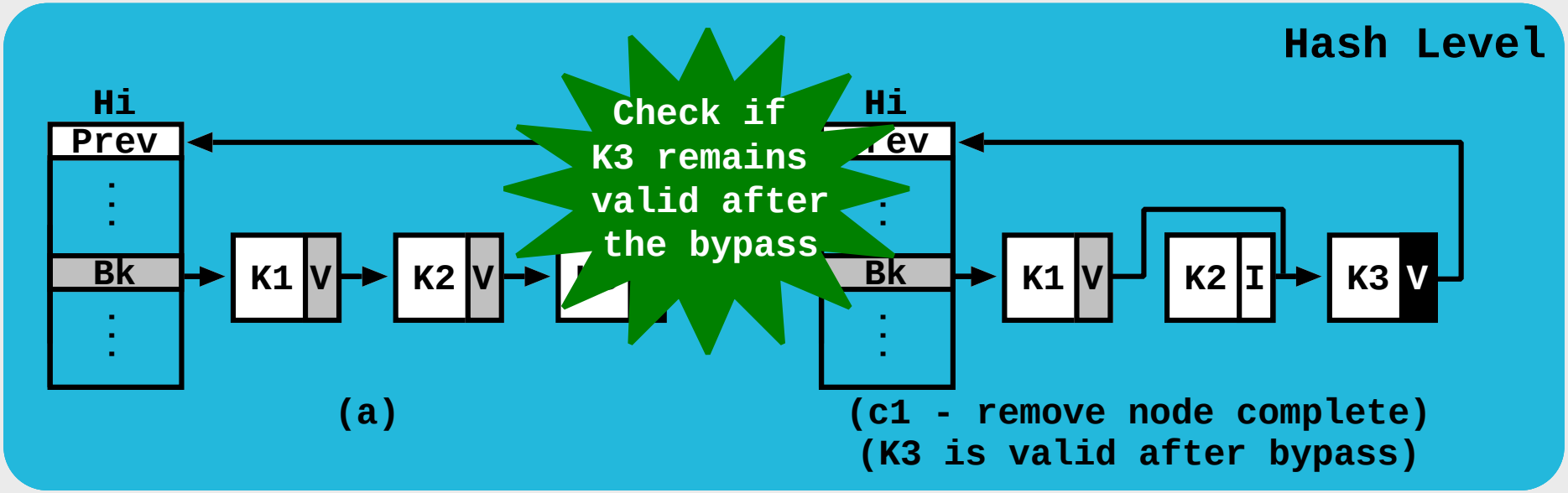
The FP Design - Remove

- 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.



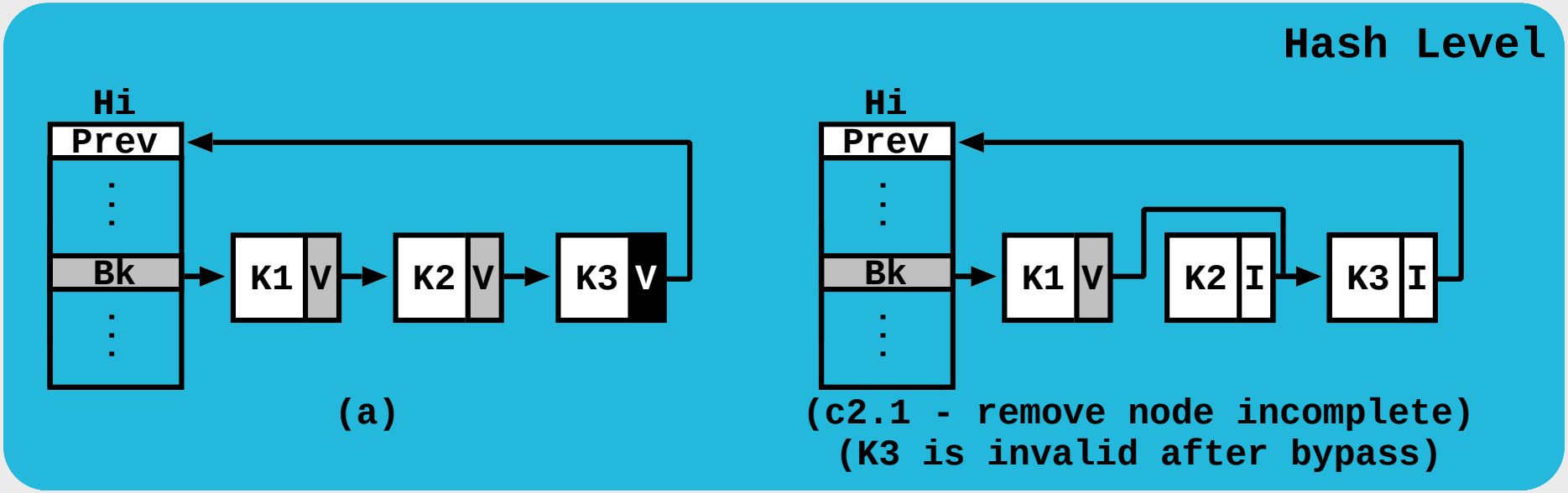
The FP Design - Remove

- 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.



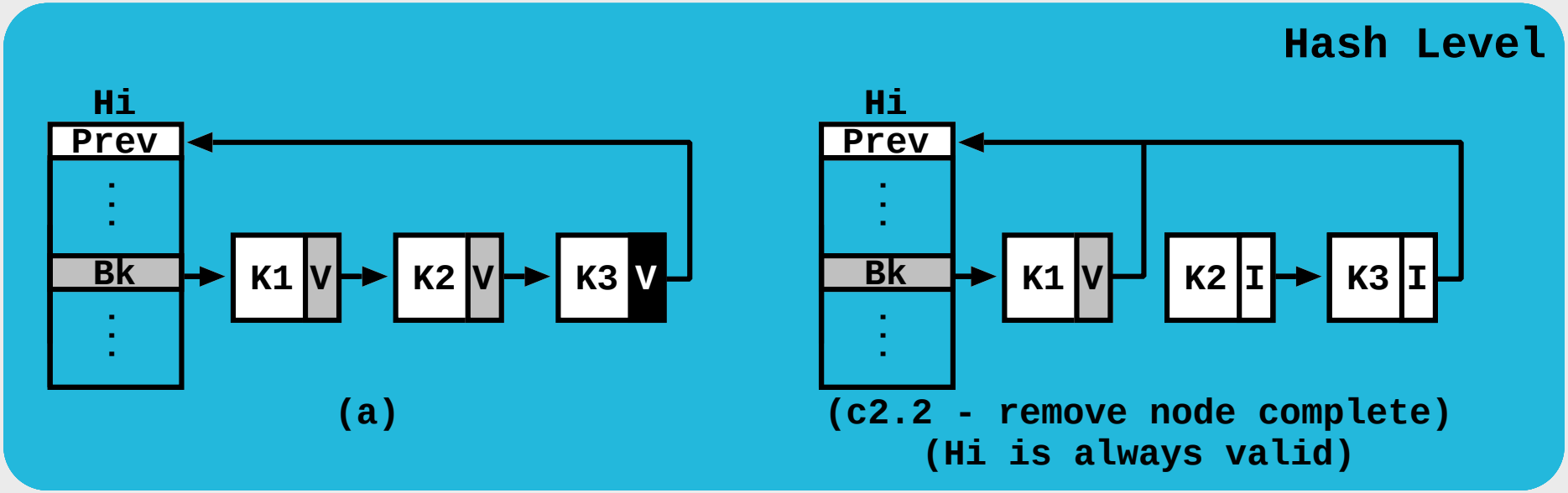
The FP Design - Remove

- 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.



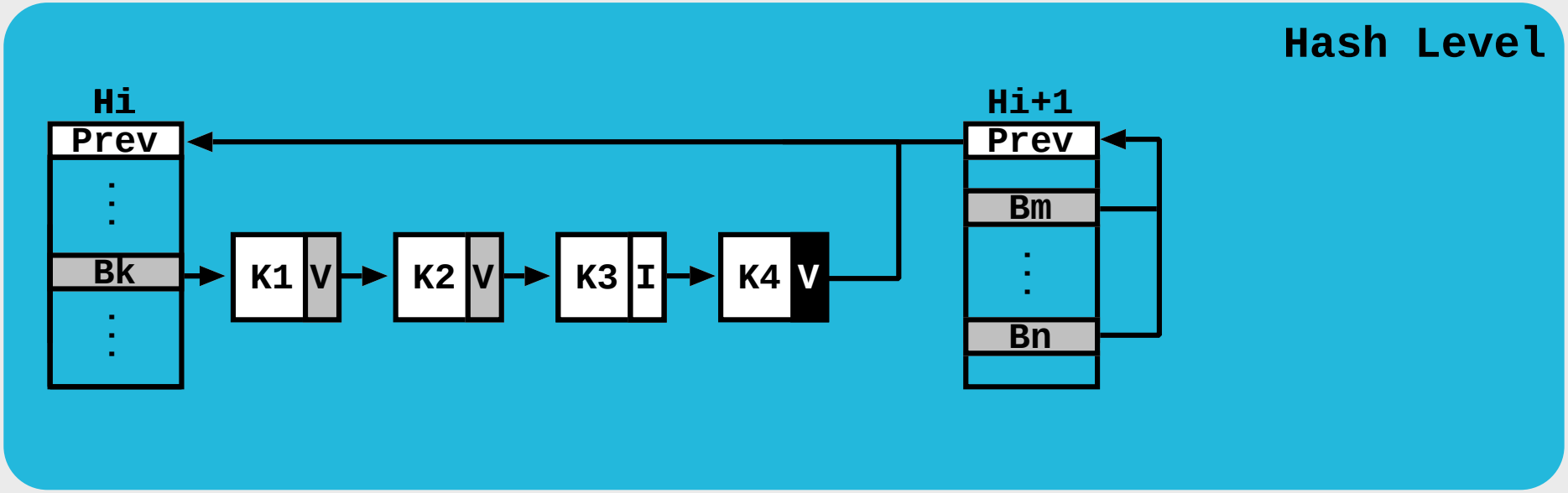
The FP Design - Remove

- 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.



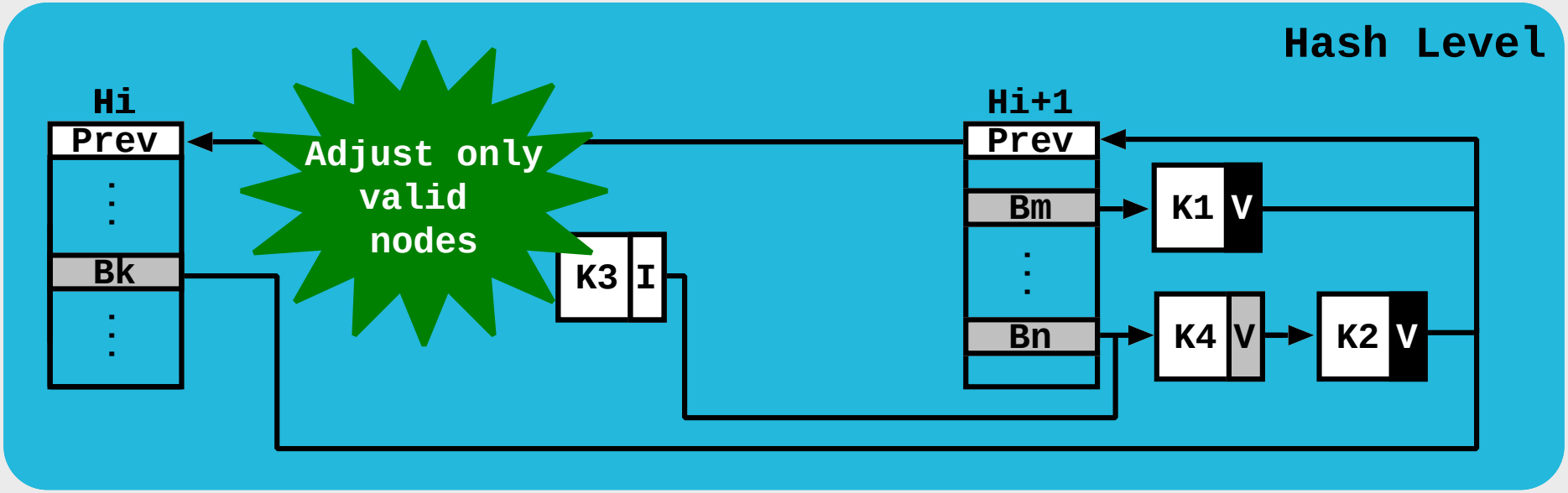
The FP Design - Expand (Valid and Invalid Nodes)

- 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).



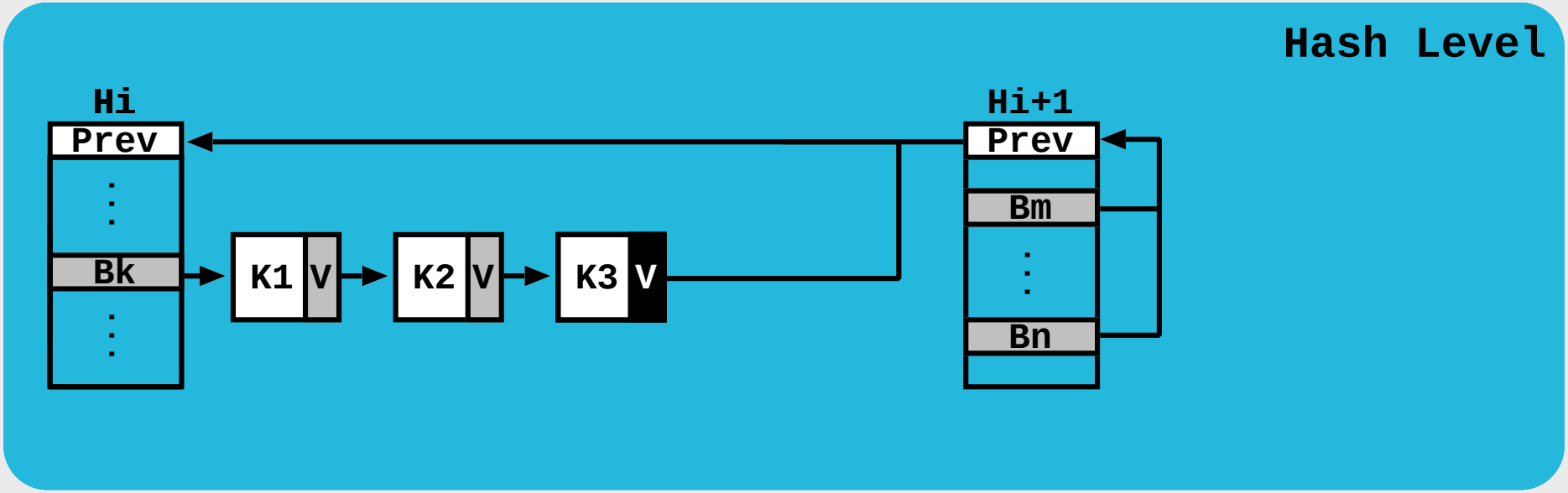
The FP Design - Expand (Valid and Invalid Nodes)

- 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).



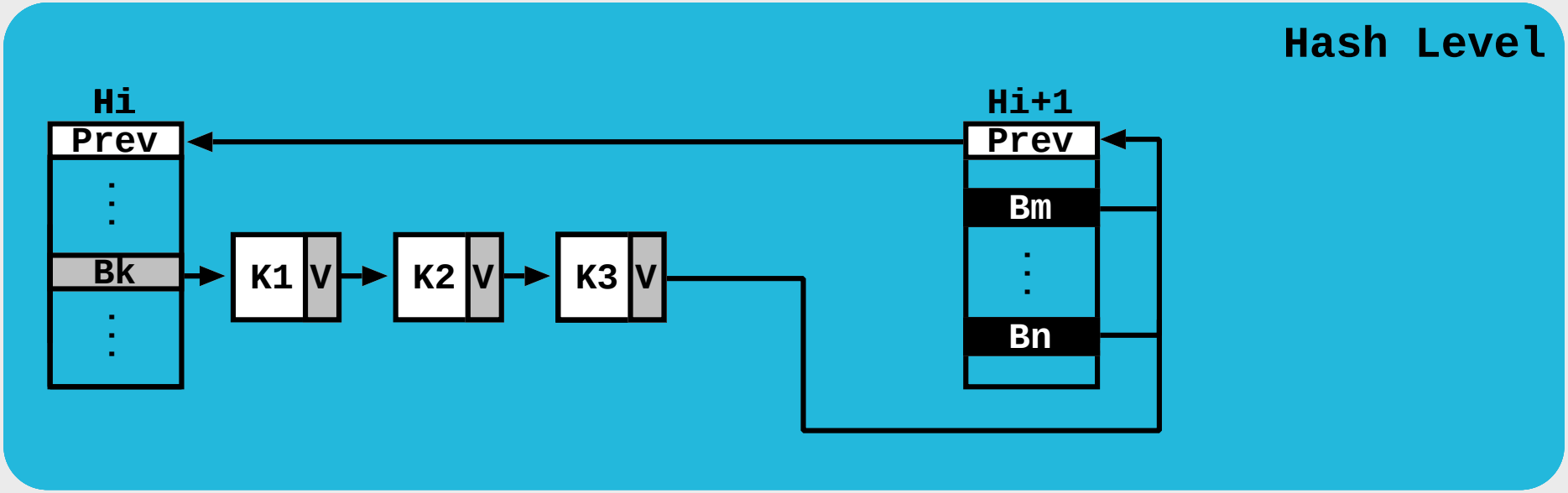
The FP Design - Expand (Valid and Invalid Nodes)

- 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).



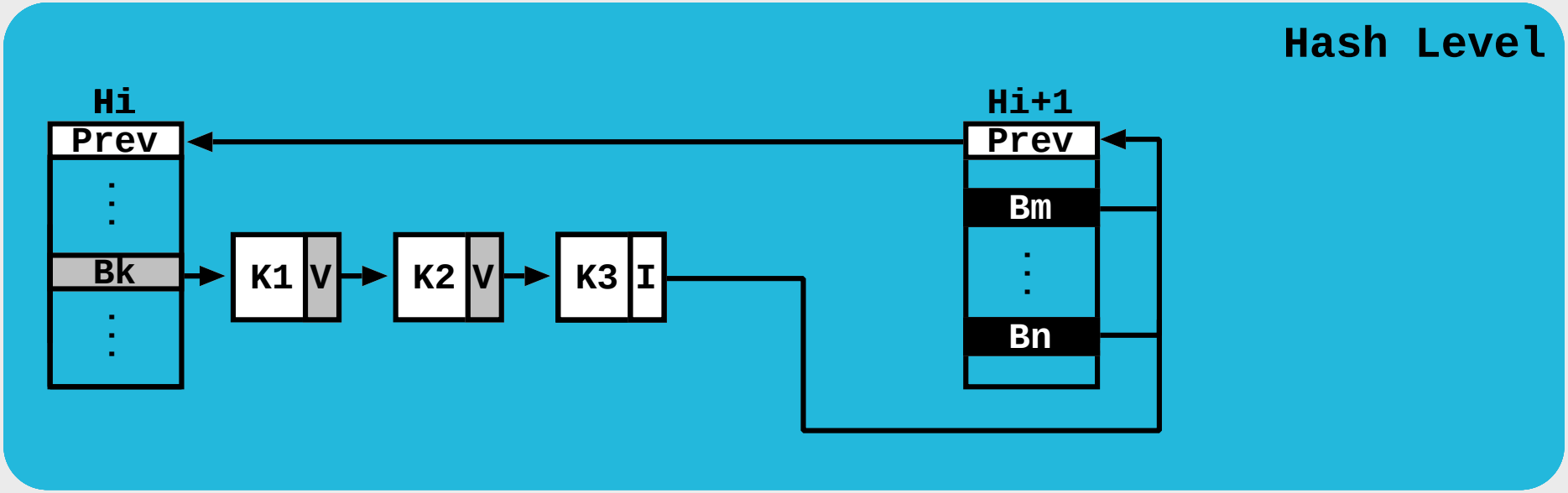
The FP Design - Expand (Valid and Invalid Nodes)

- 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).



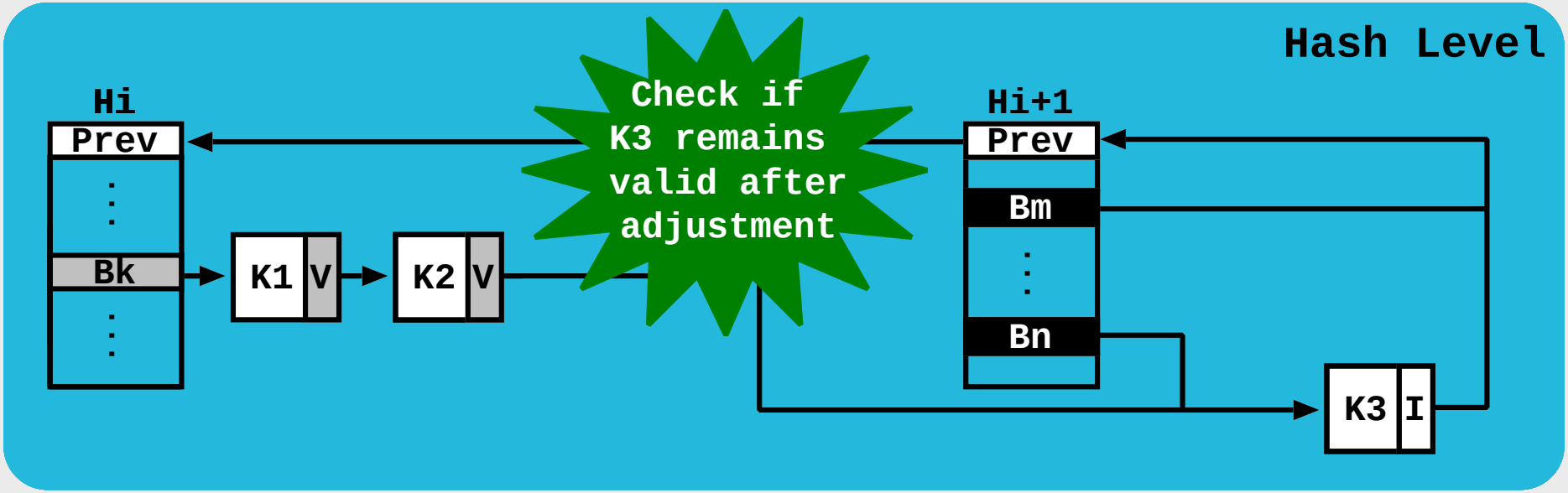
The FP Design - Expand (Valid and Invalid Nodes)

- 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).



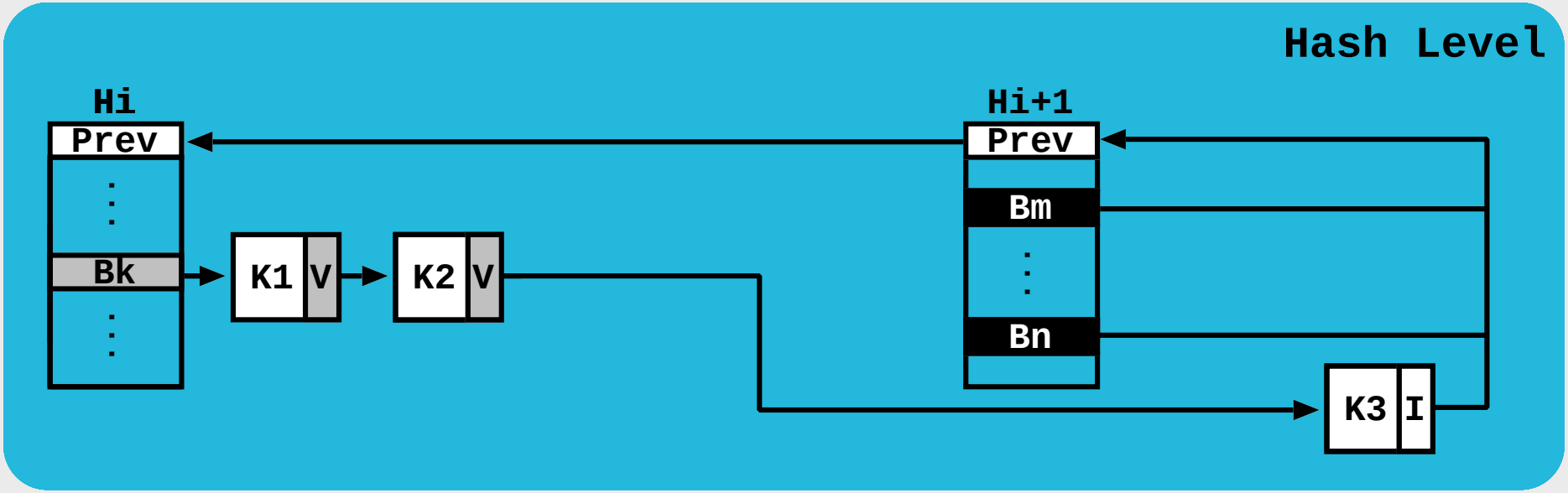
The FP Design - Expand (Valid and Invalid Nodes)

- 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).



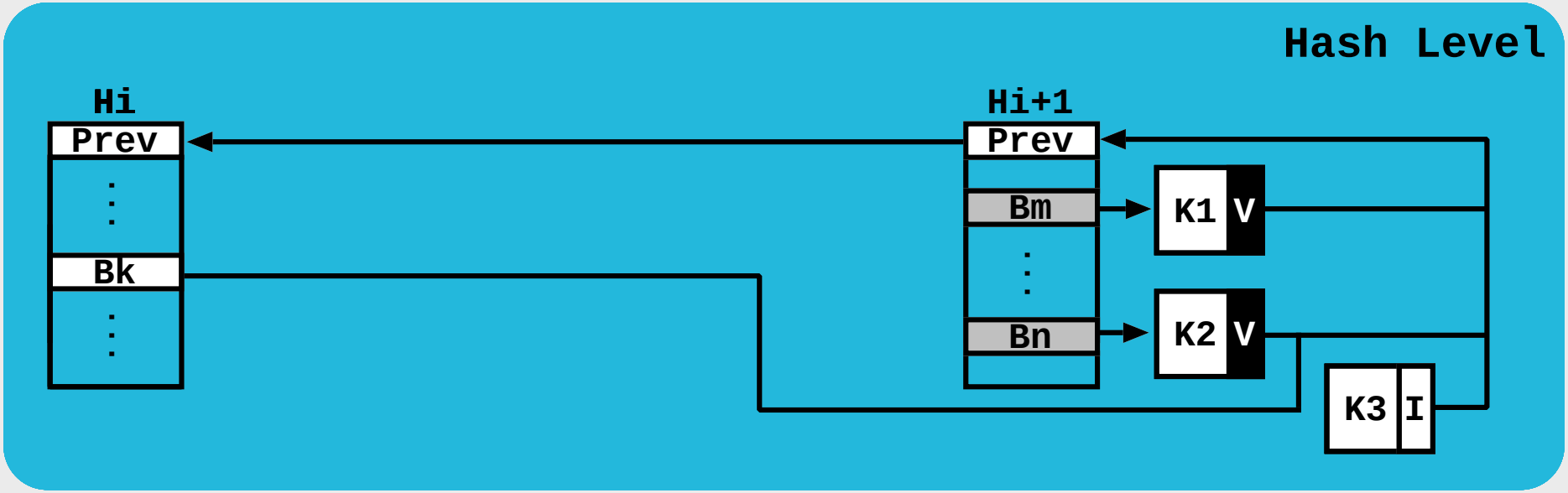
The FP Design - Expand (Valid and Invalid Nodes)

- 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 operations** (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).
- **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})						Speedup Ratio (E_{T_1}/E_{T_p})					
	CH	CS	NB	CT	FP_8	FP_{32}	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 (T_p)	Execution Time (E_{T_p})						Speedup Ratio (E_{T_1}/E_{T_p})					
	CH	CS	NB	CT	FP_8	FP_{32}	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

Conclusions and Further Work

- In this work, we have presented the **FP** hash map design:
  - Supports **concurrent search**, **insert**, **remove** and **expand** operations.
  - Combines three **properties**.

Properties / Designs	CH	CS	NB	CT	FP
Lock-Free Progress	✗	✗	✓	✓	✓
Persistent Memory References	✗	✓	✓	✗	✓
Fixed-Size Data Structures	✗	-	✗	✓	✓

Conclusions and Further Work

- In this work, we have presented the **FP** hash map design:
  - Supports **concurrent search**, **insert**, **remove** and **expand** operations.
  - Combines three **properties**.

Properties / Designs	CH	CS	NB	CT	FP
Lock-Free Progress	✗	✗	✓	✓	✓
Persistent Memory References	✗	✓	✓	✗	✓
Fixed-Size Data Structures	✗	-	✗	✓	✓

- **Experimental results** show that the design:
  - Is quite **competitive** when compared against other **state-of-the-art** designs implemented in Java.
  - Allows **different** types of **configurations** aimed for different **performances** in memory usage and execution time.

Conclusions and Further Work

- In this work, we have presented the **FP** hash map design:
  - Supports **concurrent search**, **insert**, **remove** and **expand** operations.
  - Combines three **properties**.

Properties / Designs	CH	CS	NB	CT	FP
Lock-Free Progress	✗	✗	✓	✓	✓
Persistent Memory References	✗	✓	✓	✗	✓
Fixed-Size Data Structures	✗	-	✗	✓	✓

- **Experimental results** show that the design:
  - Is quite **competitive** when compared against other **state-of-the-art** designs implemented in Java.
  - Allows **different** types of **configurations** aimed for different **performances** in memory usage and execution time.

- **Further work** will include the implementation of the design as a library that can be **easily included** in big systems (**Yap-Prolog**).

Thank You !!!

Miguel Areias and Ricardo Rocha
miguel-areias@dcc.fc.up.pt *ricroc@dcc.fc.up.pt*

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