On Extending a Fixed Size, Persistent and Lock-Free Hash Map Design to Store Sorted Keys

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

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 four **properties**: **Lock-Free Progress**, **Persistent Memory References**, **Fixed-Size Data Structures** and **Store Sorted Keys**.

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

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 four **properties**: **Lock-Free Progress**, **Persistent Memory References**, **Fixed-Size Data Structures** and **Store Sorted Keys**.

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

- 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 four **properties**: **Lock-Free Progress**, **Persistent Memory References**, **Fixed-Size Data Structures** and **Store Sorted Keys**.

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

- In **this talk** we will:
  - explain why these **properties** are **important** in some **domains**.
  - present our novel design (**FP**) that **supports** those four **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 four **properties**: **Lock-Free Progress**, **Persistent Memory References**, **Fixed-Size Data Structures** and **Store Sorted Keys**.

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

- In **this talk** we will:
  - explain why these **properties** are **important** in some **domains**.
  - present our novel design (**FP**) that **supports** those four **properties**.
  - show a **performance** analysis **comparison** (we will skip **NB**).

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.

Property 4: Store Sorted Keys

- Is the process of **storing keys** in a **sequence** with is **ordered** according with **some criteria**.
  - A **dictionary** is an example of a resource that **stores sorted items** (keys/strings) according with their **lexicographical order**.

Property 4: Store Sorted Keys

- Is the process of **storing keys** in a **sequence** with is **ordered** according with **some criteria**.
  - A **dictionary** is an example of a resource that **stores sorted items** (keys/strings) according with their **lexicographical order**.
- According with **Donald Knuth** (The Art of Computer Programming [2Ed - Vol 3]), **"... the order** in which **items are stored** in **computer memory** often has a **profound influence** on the **speed and simplicity** of algorithms that manipulate **those items**.**"**

Property 4: Store Sorted Keys

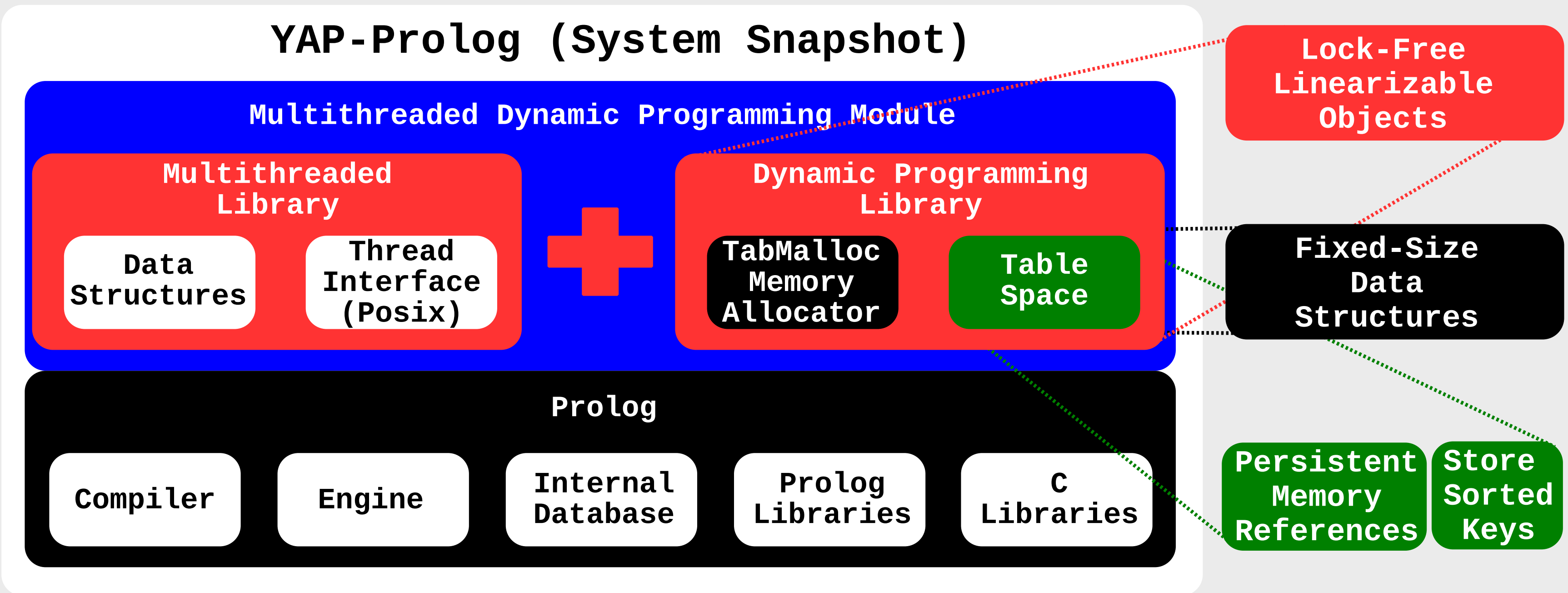
- Is the process of **storing keys** in a **sequence** with is **ordered** according with **some criteria**.
  - A **dictionary** is an example of a resource that **stores sorted items** (keys/strings) according with their **lexicographical order**.

- According with **Donald Knuth** (The Art of Computer Programming [2Ed - Vol 3]), **"**... **the order** in which **items are stored** in **computer memory** often has a **profound influence** on the **speed and simplicity** of algorithms that manipulate **those items**.**"**

- Some of the **advantages** of using **sorted keys** are:
  - **efficient searches**. Enables the **cut of the search space**, in tree-based data structures.
  - **processing keys in a defined order**. Suitable for **non-exact match searches**, such as finding all keys in an interval.

Properties 1, 2, 3 and 4: Real-World Example

- These properties are **useful** when the **internals** of a **hash map** are directly **accessed** by an external module.
- 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 - Key Ideas

- Combines a **hash trie structure** (index keys) with a **separate chaining** mechanism (deal with key collisions).
  - Keys are **sorted** in the **hash trie structure**, but **not sorted** in the **separate chaining** mechanism.

The FP Design - Key Ideas

- Combines a **hash trie structure** (index keys) with a **separate chaining** mechanism (deal with key collisions).
  - Keys are **sorted** in the **hash trie structure**, but **not sorted** in the **separate chaining** mechanism.

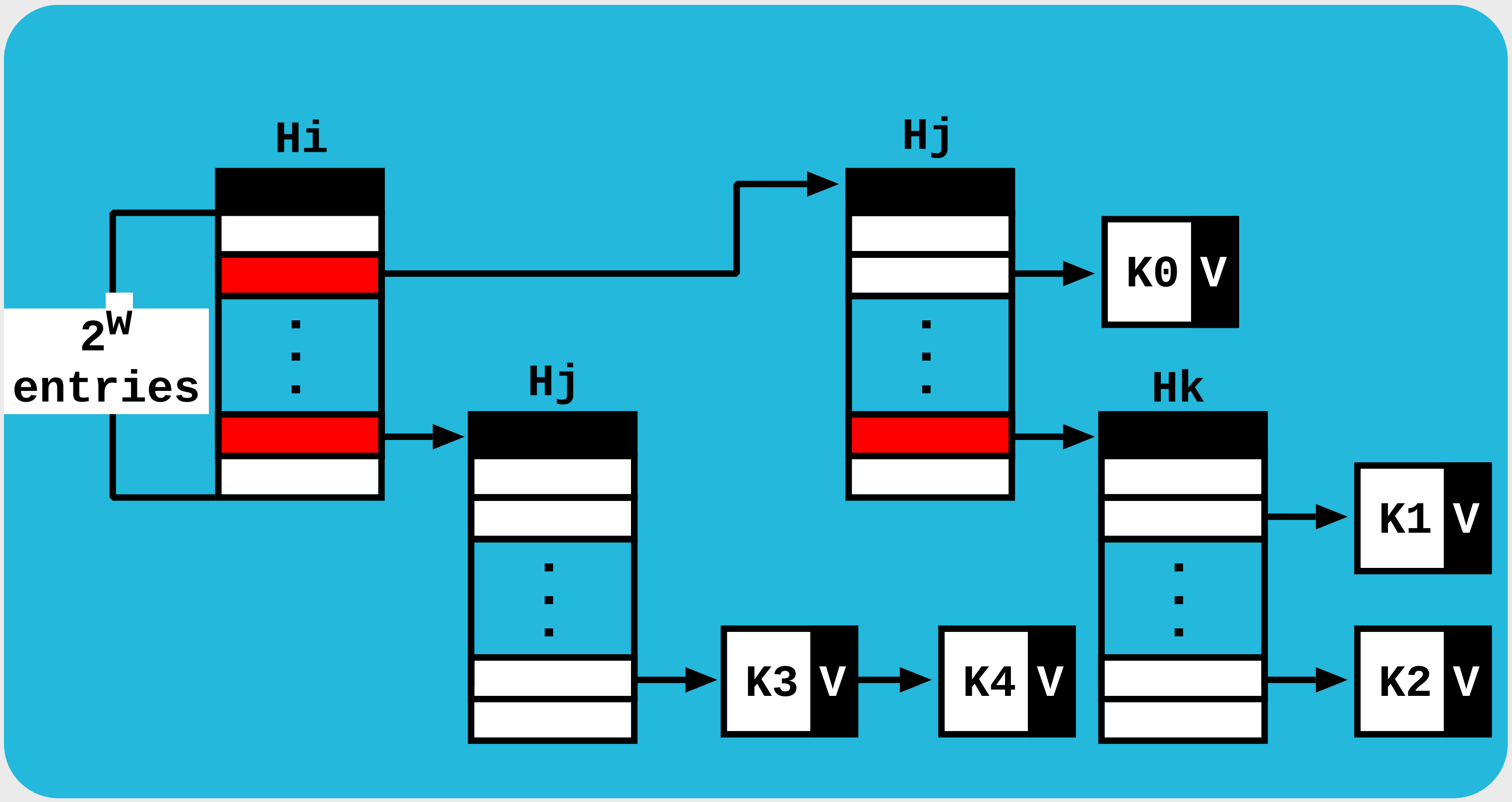
- On each **hash level**, a key is inserted in a **hash bucket** using its high-order bits (sorting upon insertion).
  - A **hash level** is **expanded** when a key must be inserted in a **hash bucket** chain that is full.

The FP Design - Key Ideas

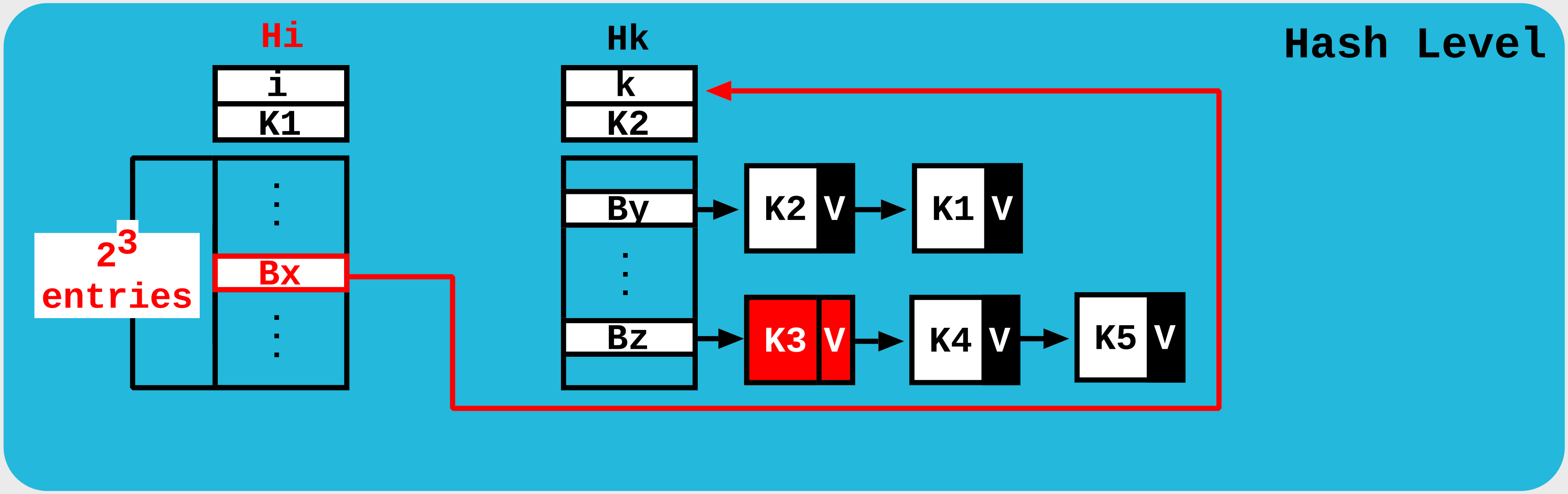
- Combines a **hash trie structure** (index keys) with a **separate chaining** mechanism (deal with key collisions).
  - Keys are **sorted** in the **hash trie structure**, but **not sorted** in the **separate chaining** mechanism.
- On each **hash level**, a key is inserted in a **hash bucket** using its high-order bits (sorting upon insertion).
  - A **hash level** is **expanded** when a key must be inserted in a **hash bucket** chain that is full.
- **Expansion** properties:
  - **XOR operations** detect the **high-order bits** where the keys differ.
  - Keys **remain sorted** using **back expansions** to **expand** keys in shallow levels and **front expansions** to **expand** keys in deep levels.

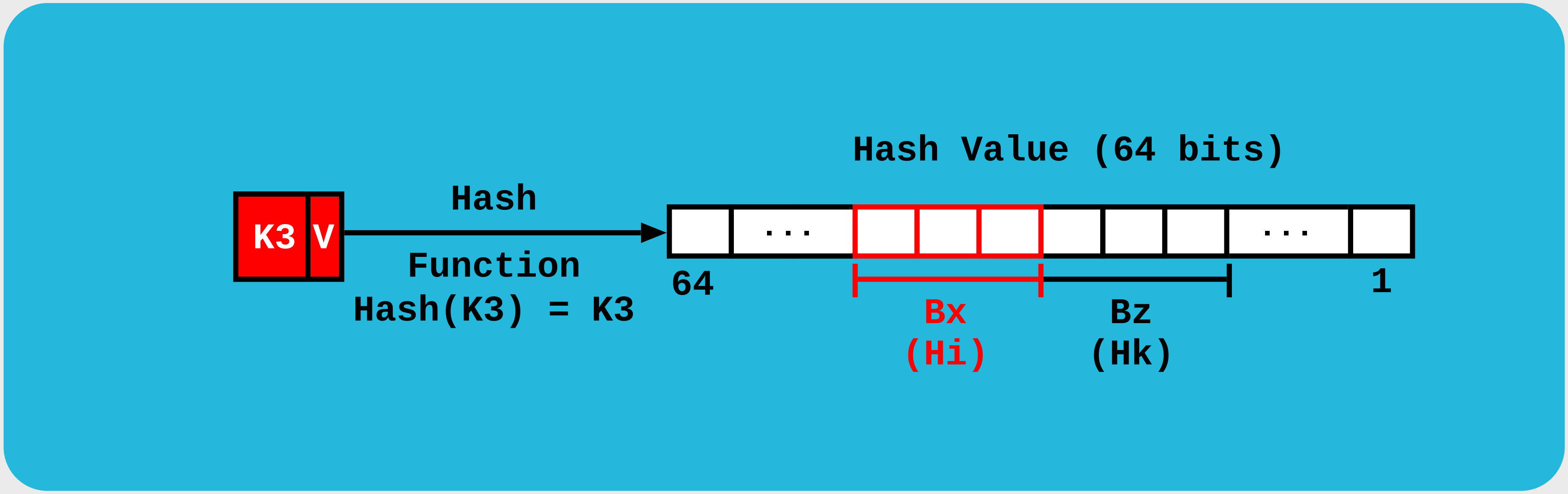
The FP Design - Hash Trie Structure

- **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)**.

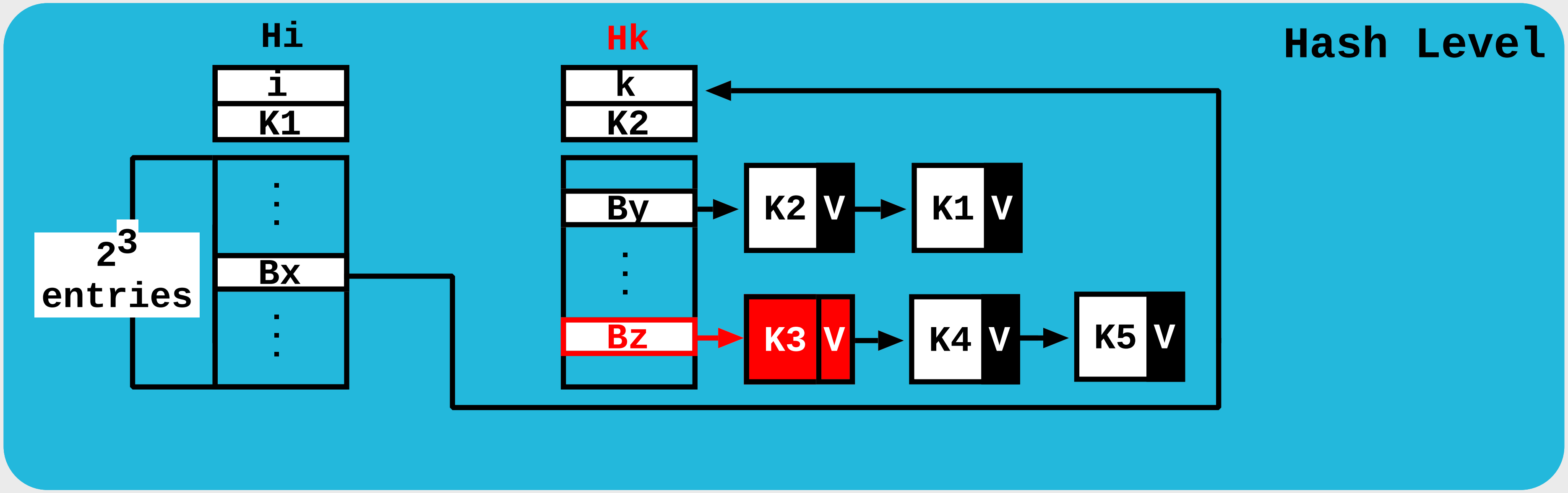


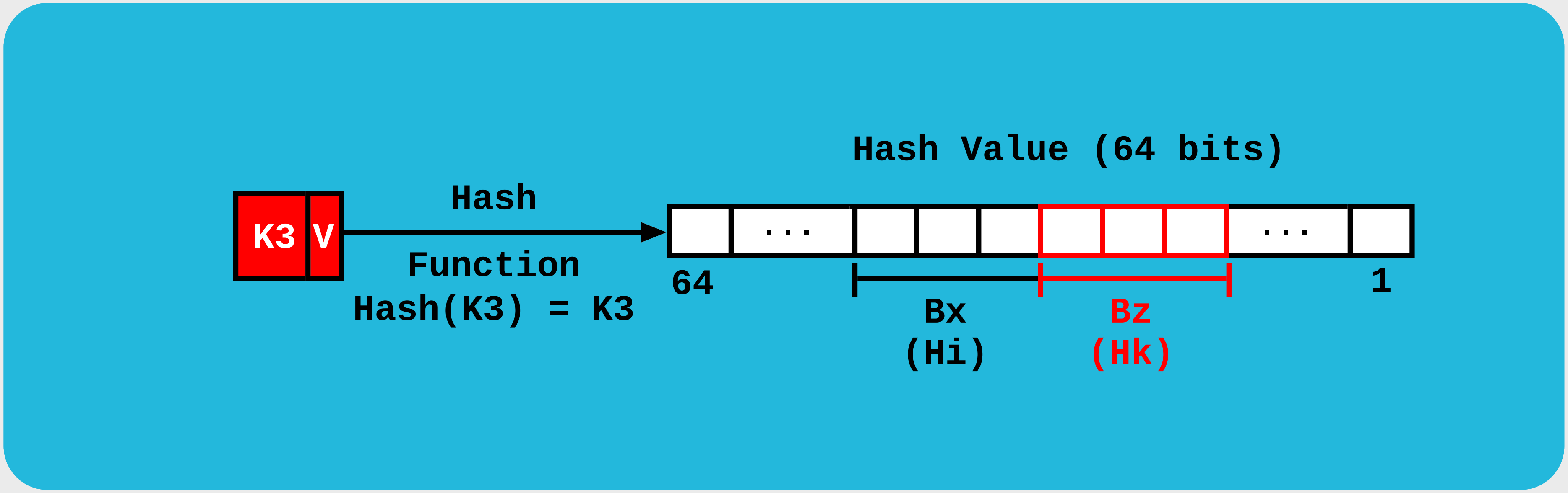
The FP Design - Searching for K3





The FP Design - Searching for K3



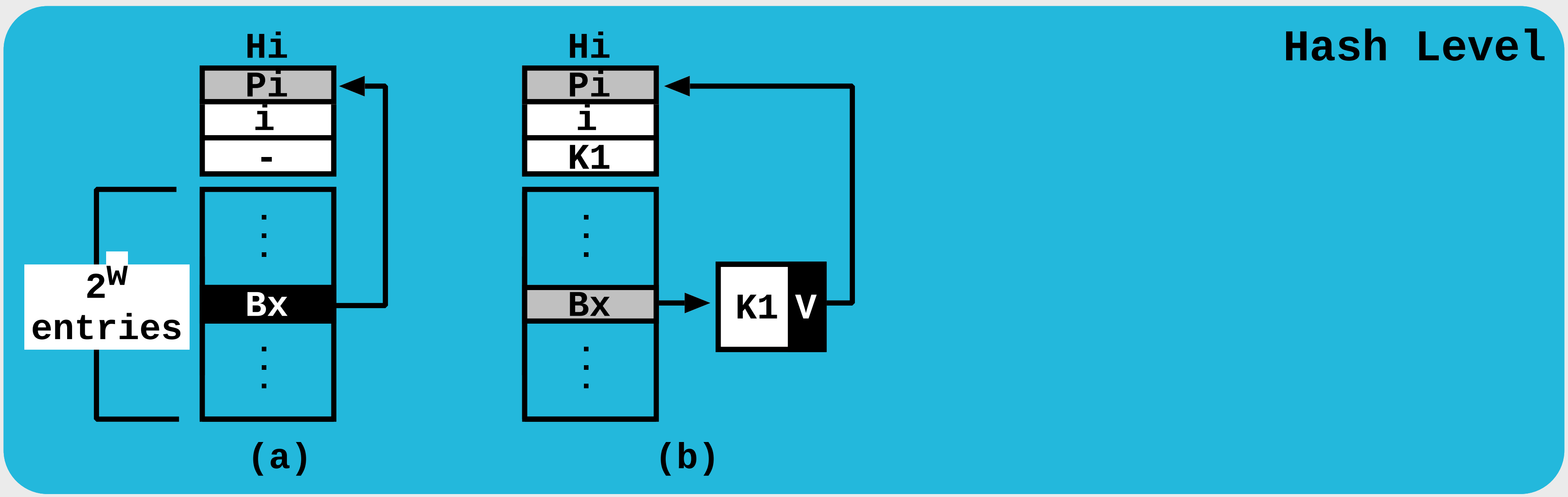


The FP Design - Internals

- To support **multithreading**, our design allows **threads** to:
  - **Recover from preemption**, by using a **previous field (Pi)** 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.

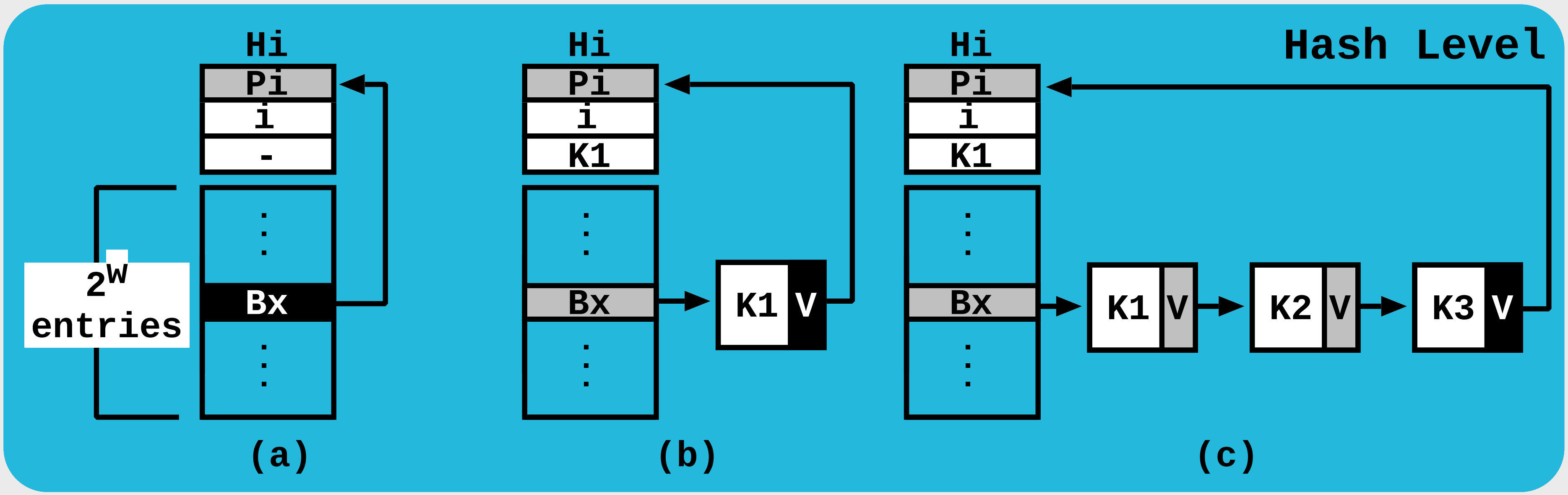
The FP Design - Internals

- To support **multithreading**, our design allows **threads** to:
  - **Recover from preemption**, by using a **previous field (Pi)** 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.

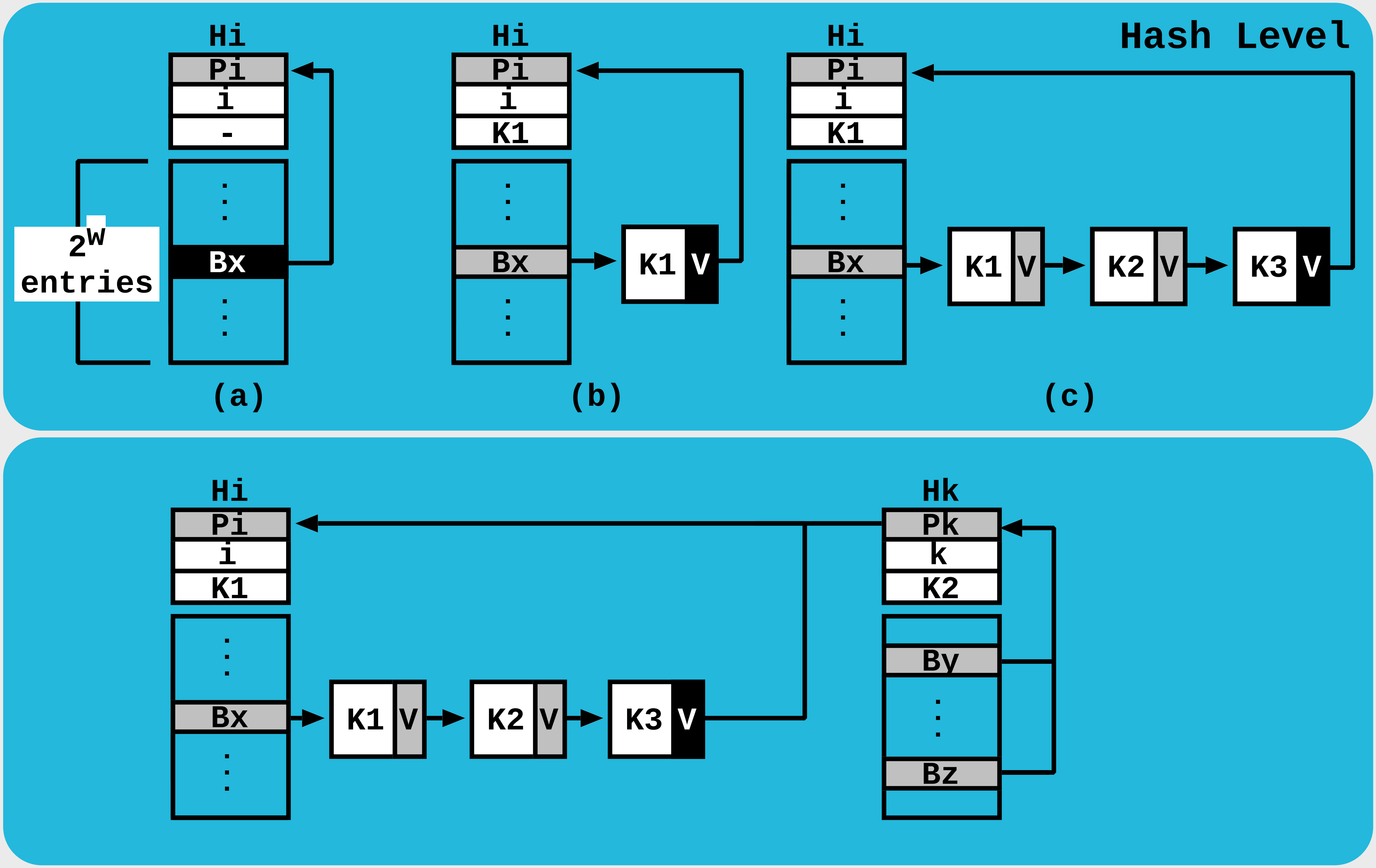


The FP Design - Internals

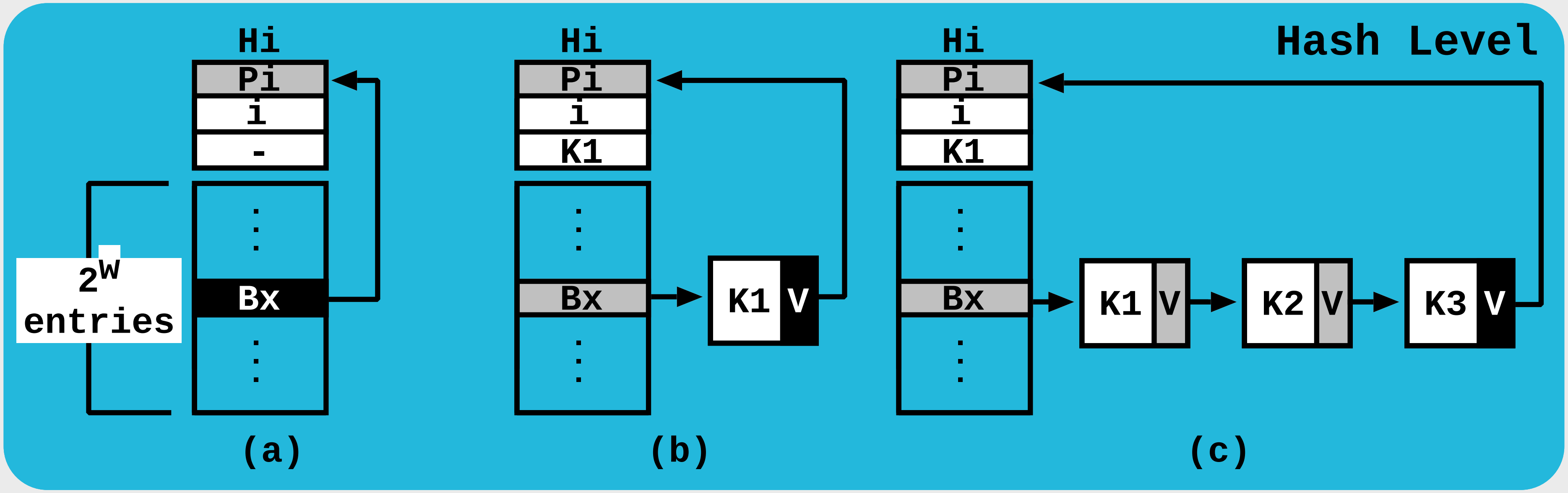
- To support **multithreading**, our design allows **threads** to:
  - **Recover from preemption**, by using a **previous field (Pi)** to traverse the **hash buckets** backwards.
  - **Identify chains**, by using a **back-reference** on the end of each chain.
  - **Maintain consistency**, by using **CAS**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

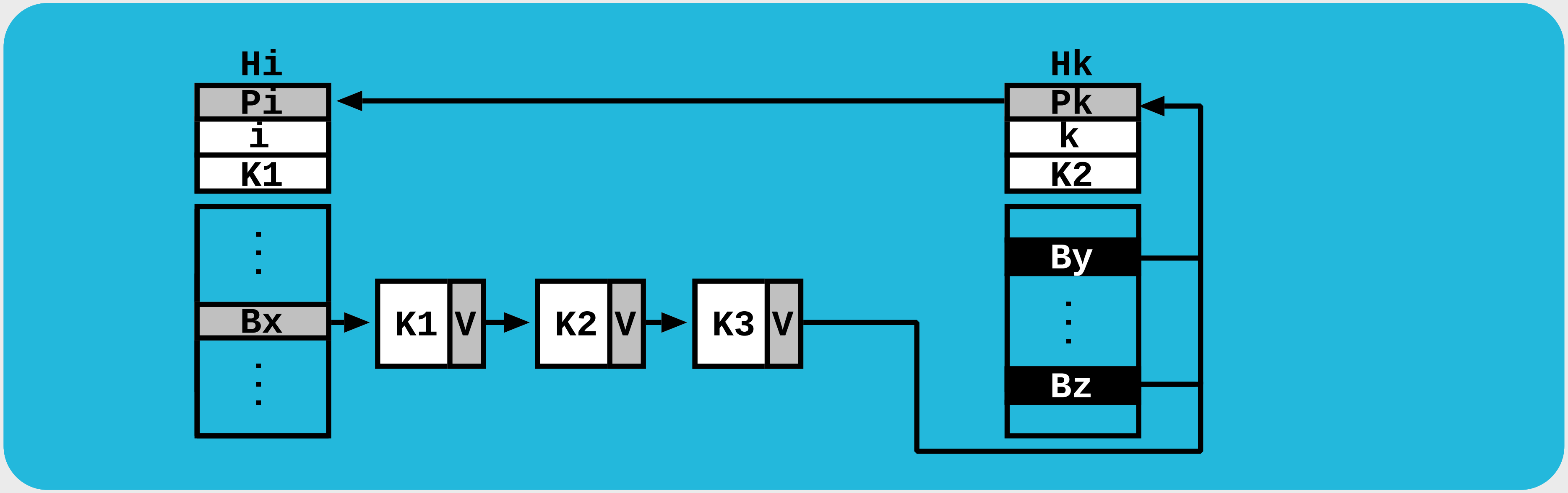


The FP Design - Front Expansion

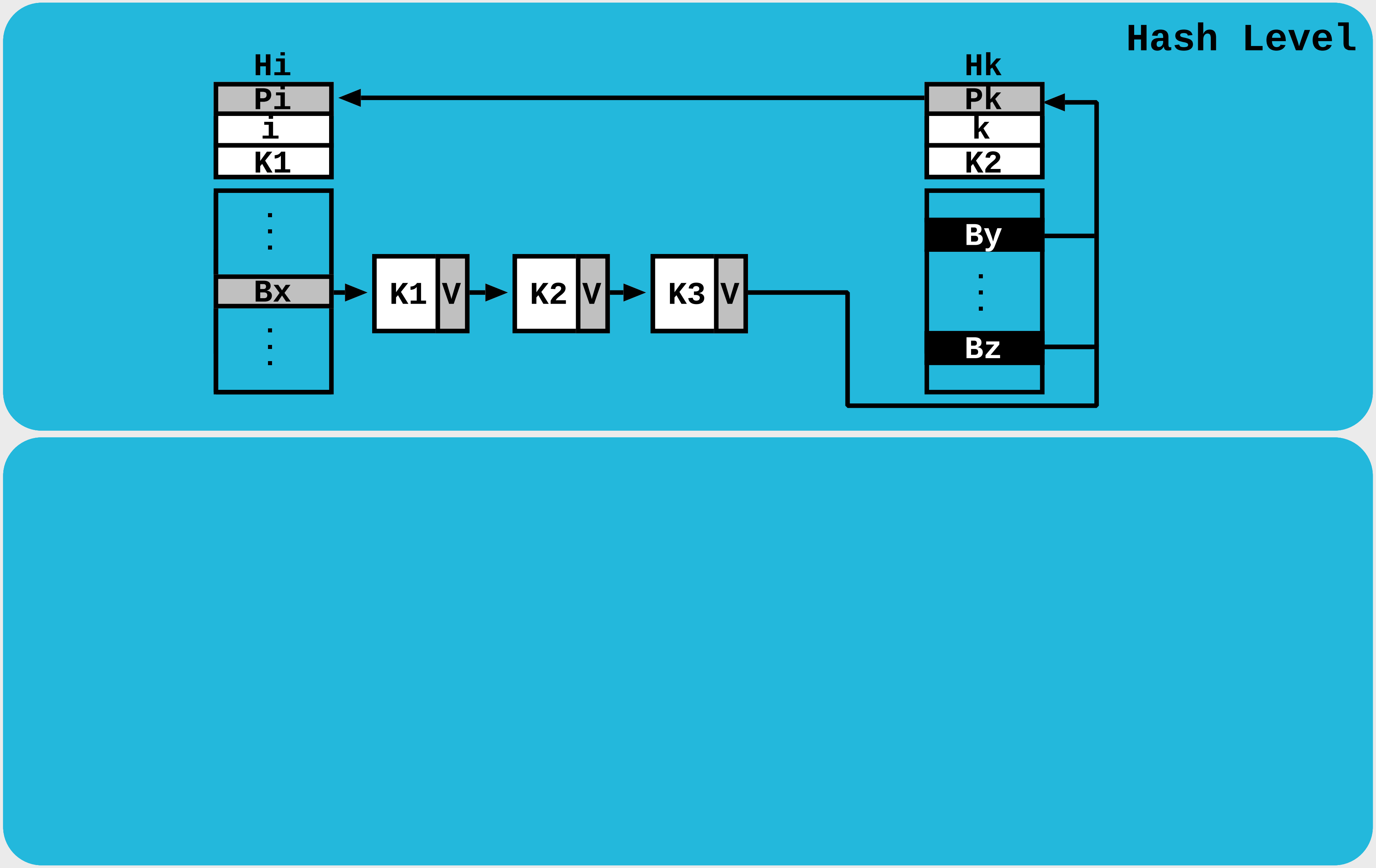


The FP Design - Front Expansion

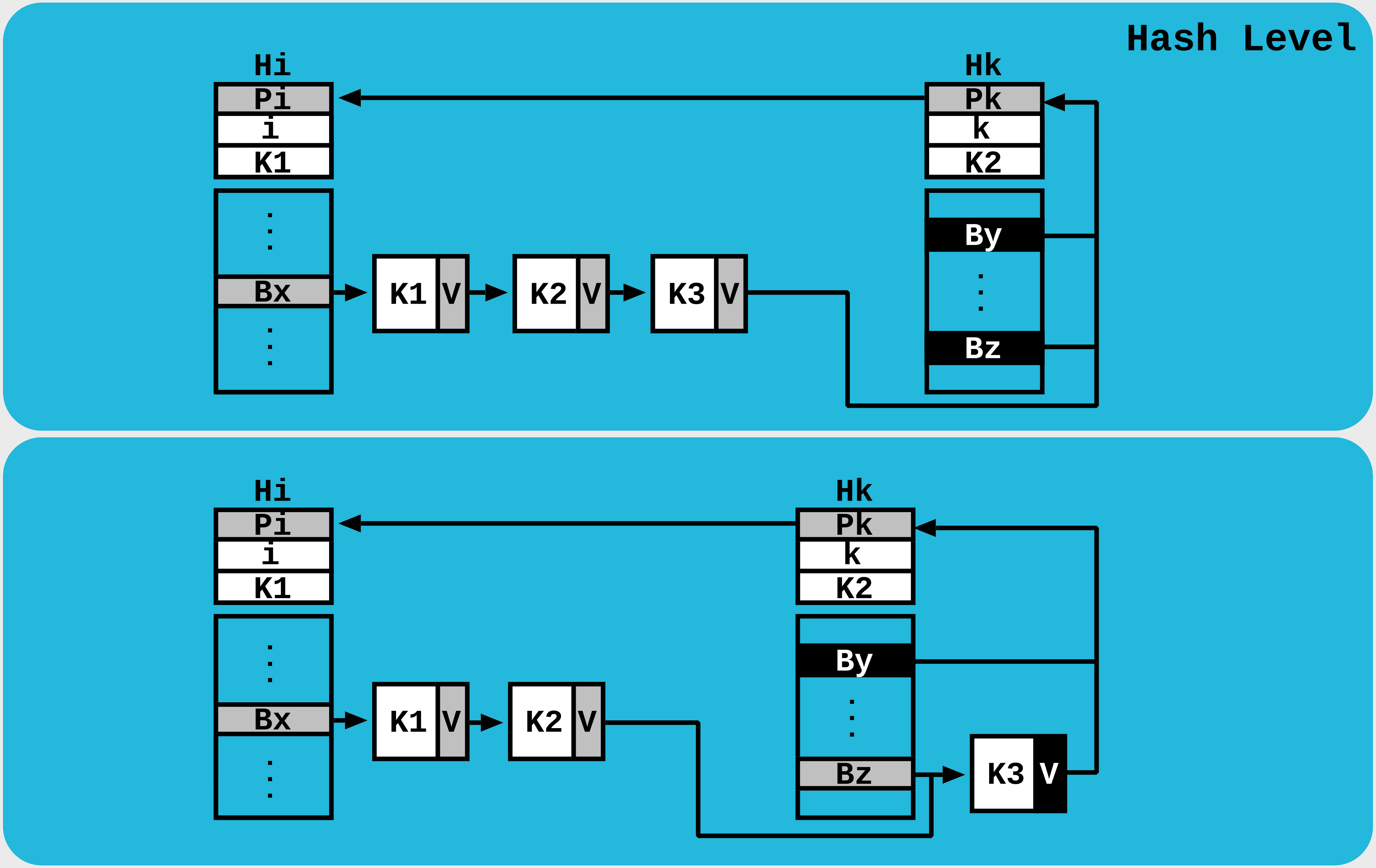




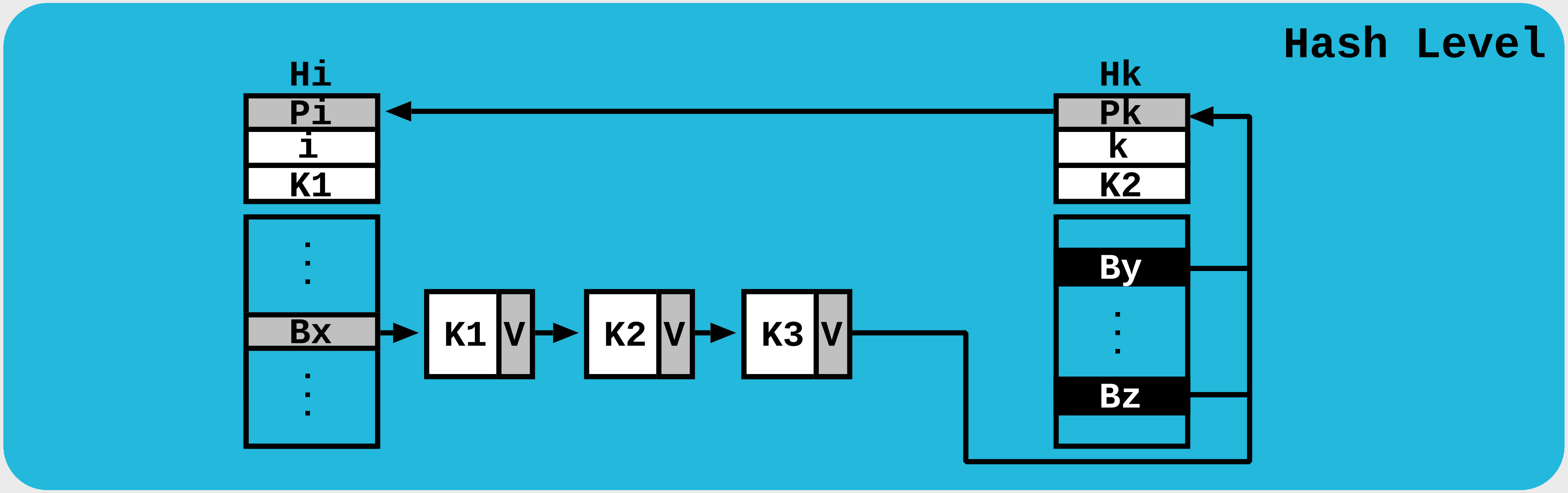
The FP Design - Front Expansion

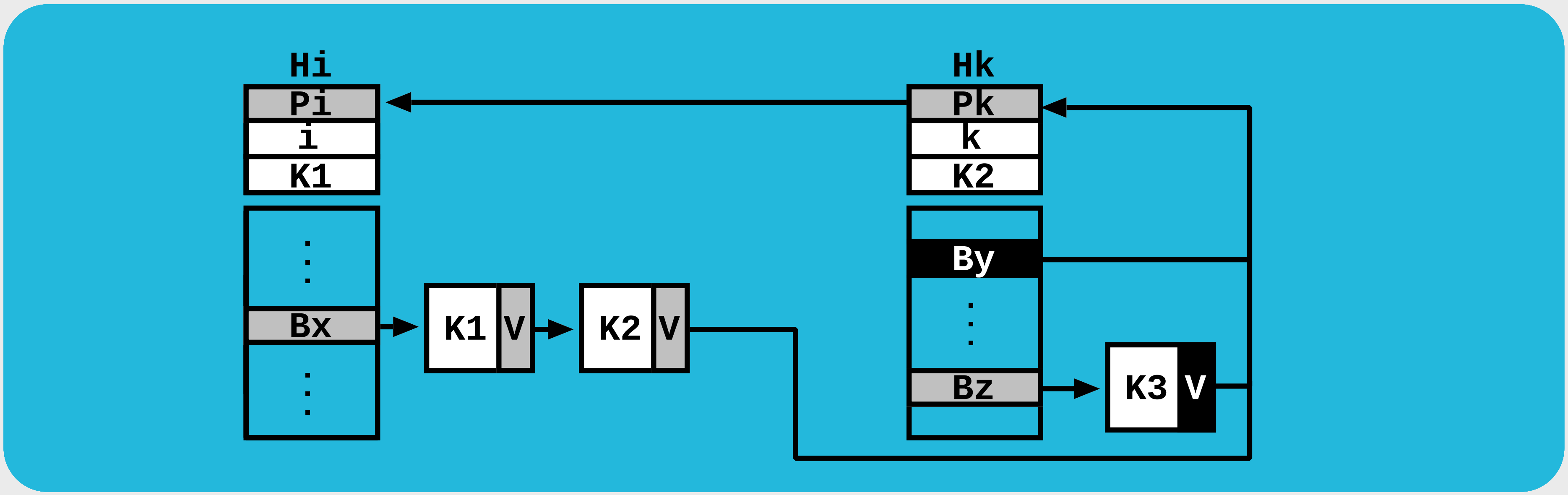


The FP Design - Front Expansion

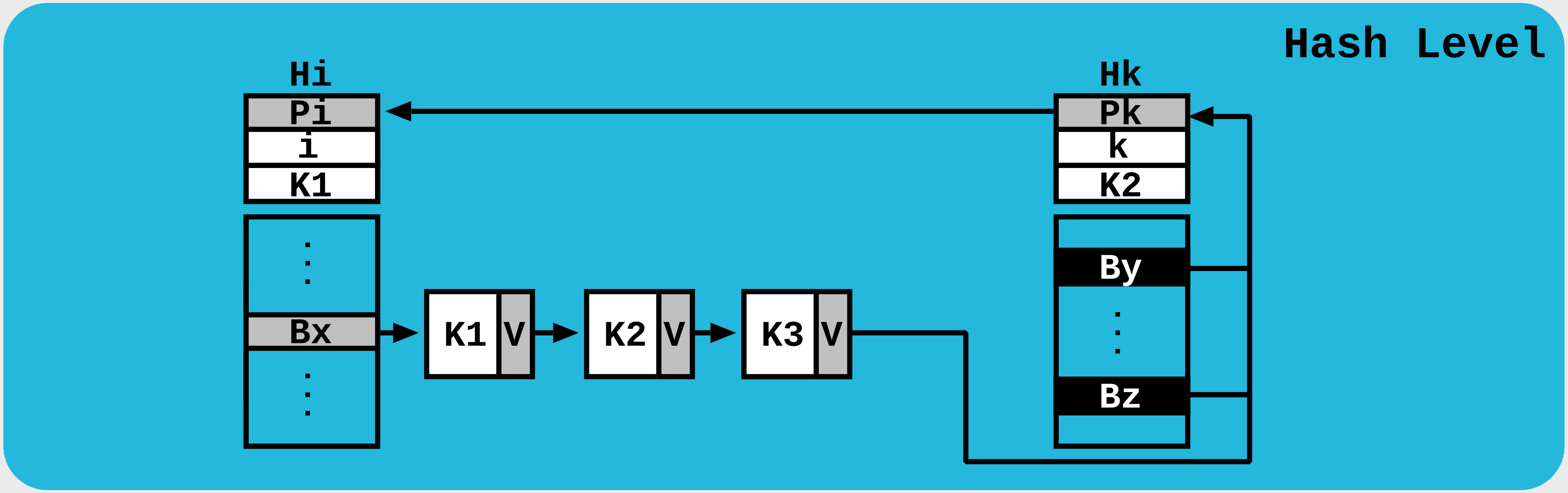


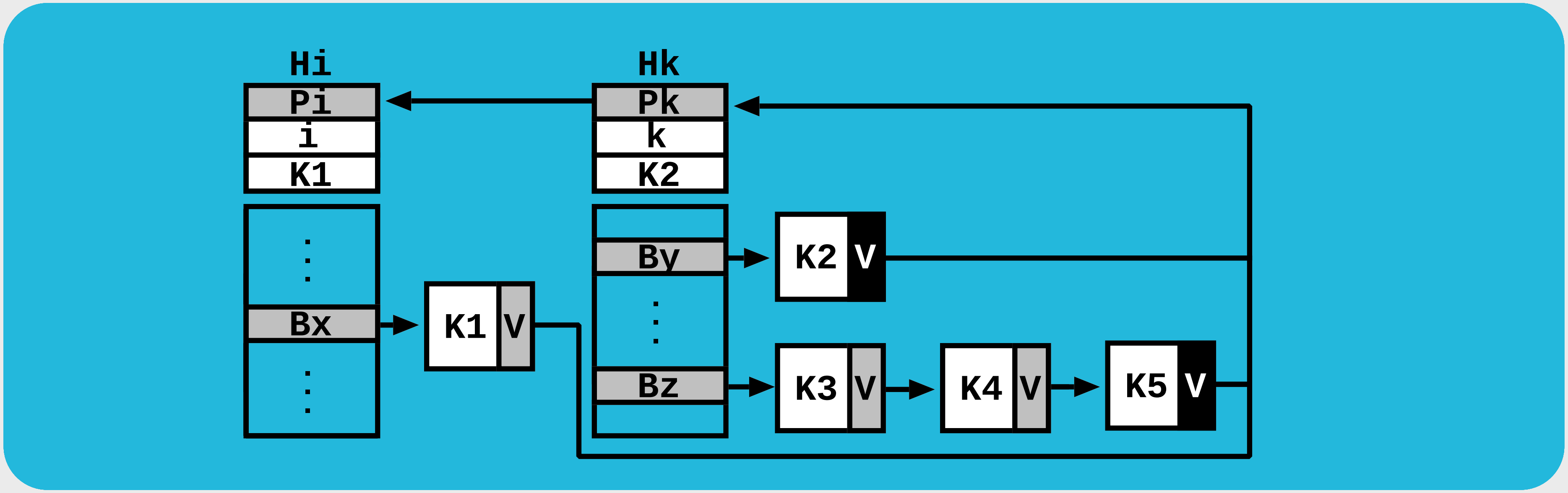
The FP Design - Front Expansion



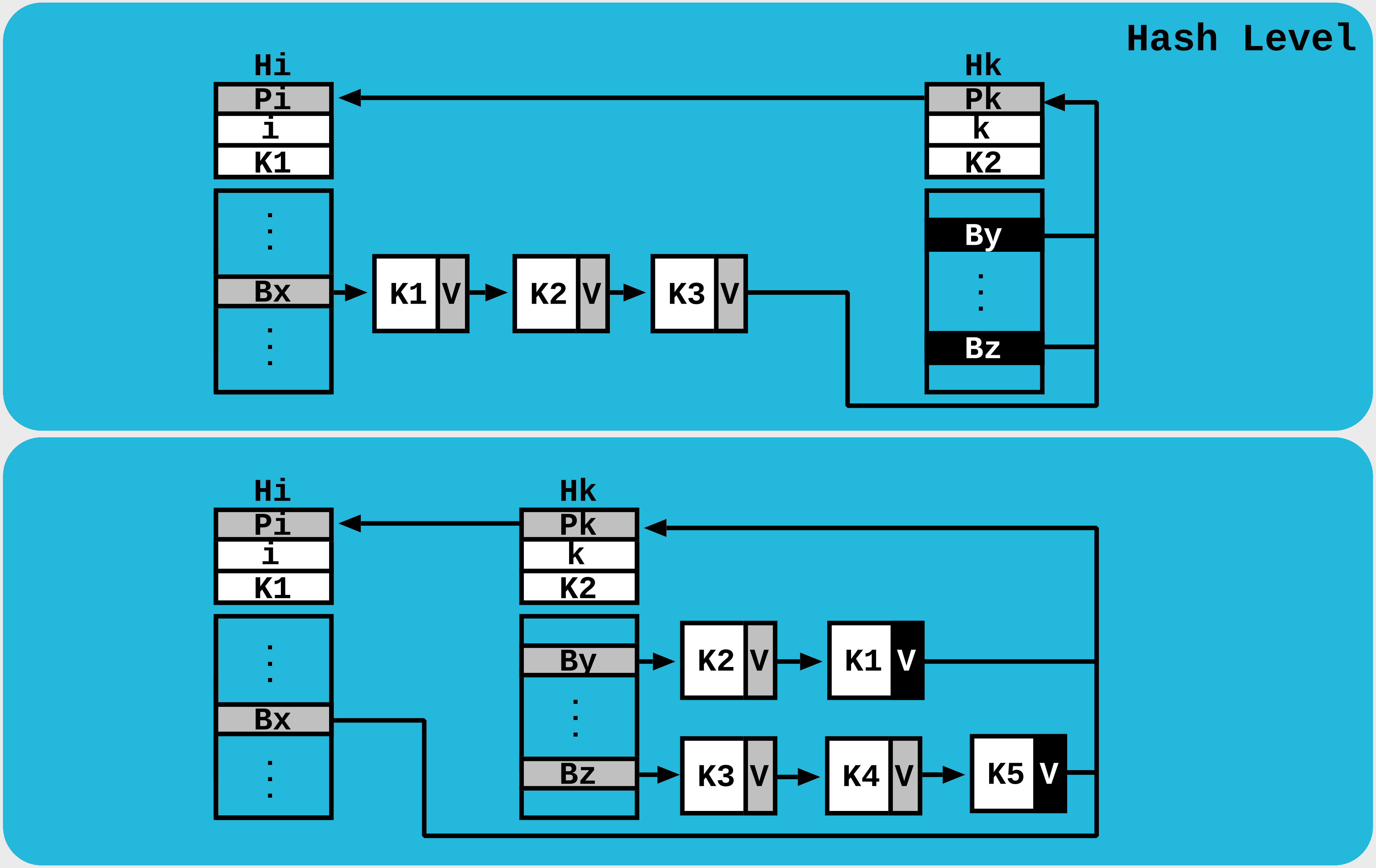


The FP Design - Front Expansion

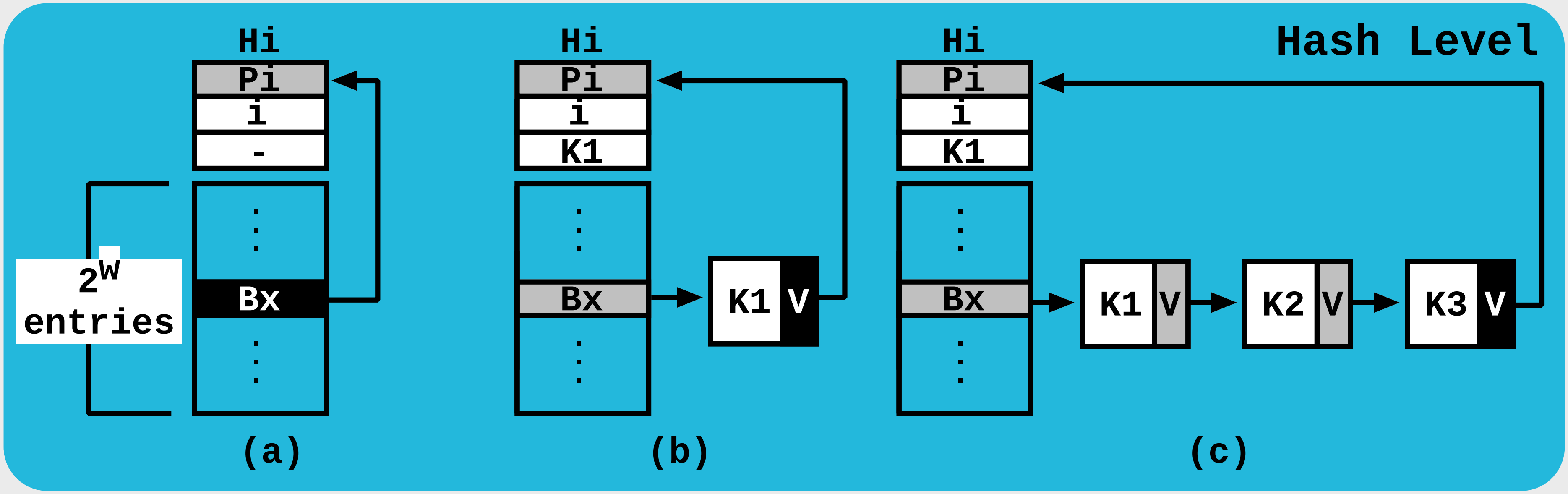


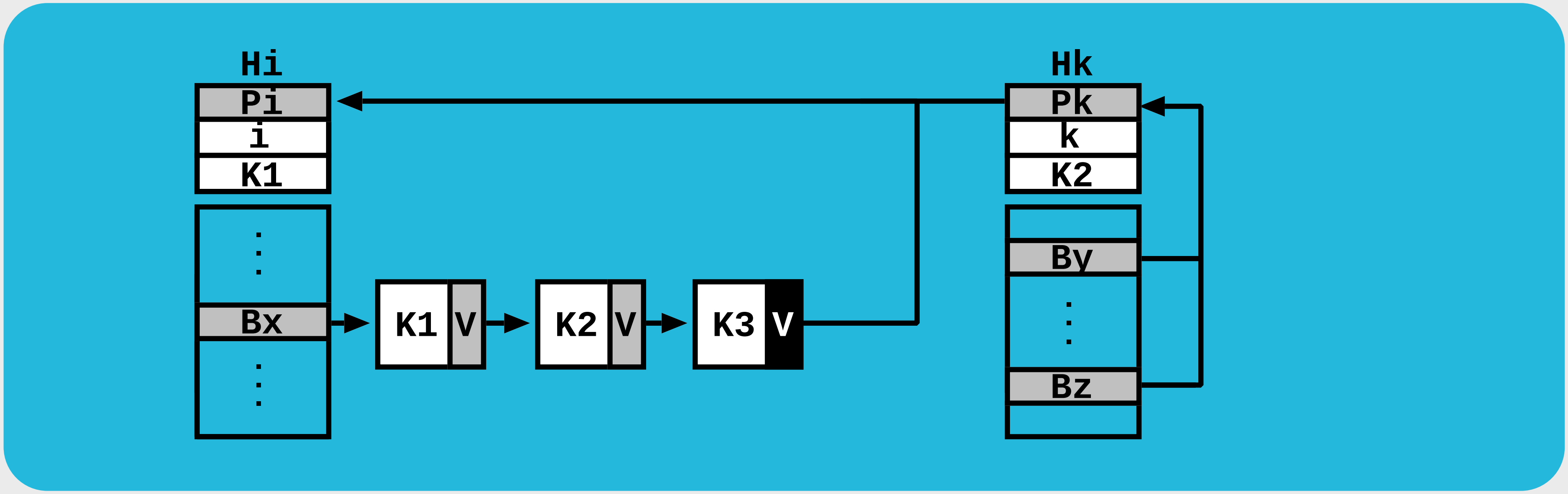


The FP Design - Front Expansion

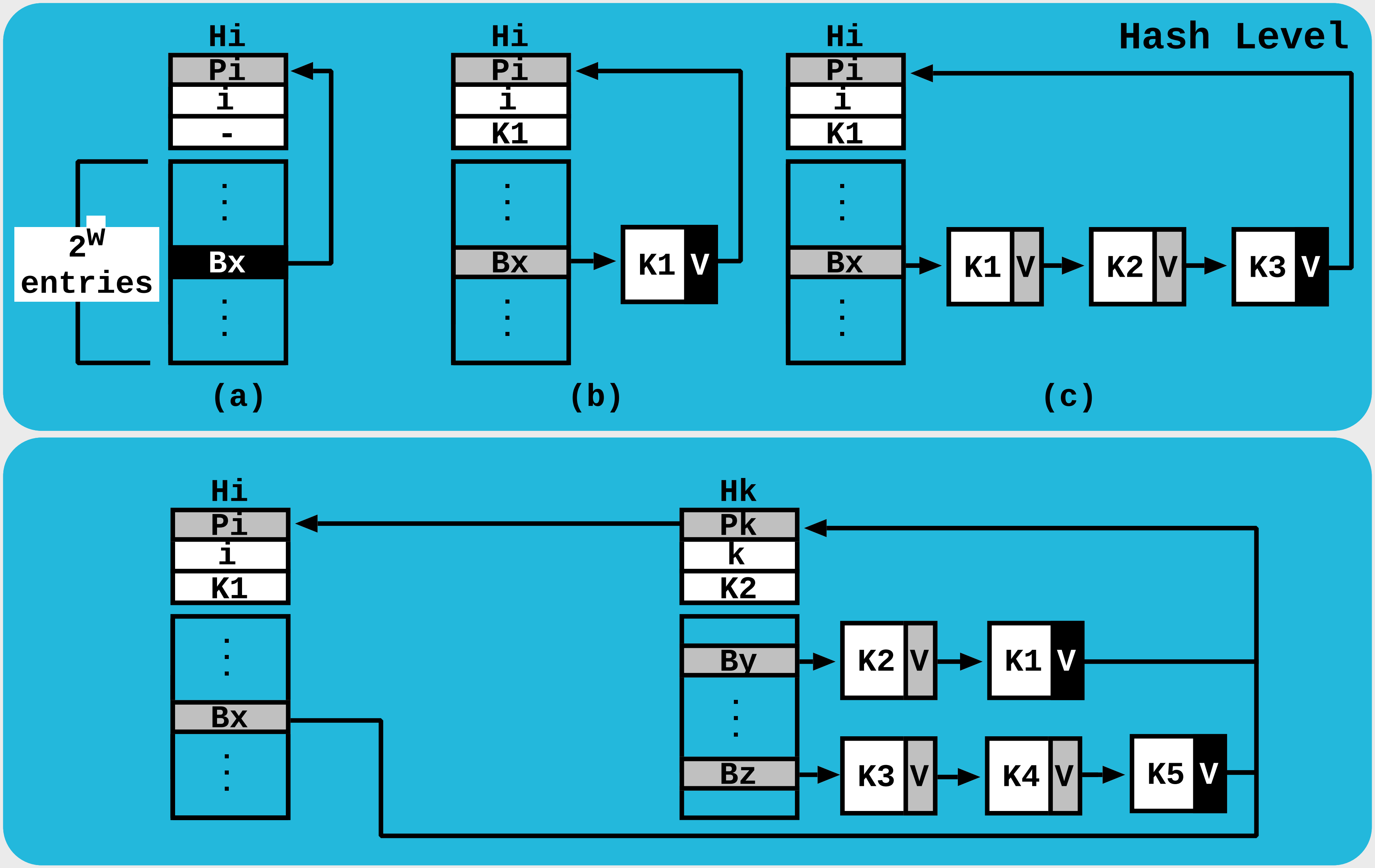


The FP Design - Back Expansion

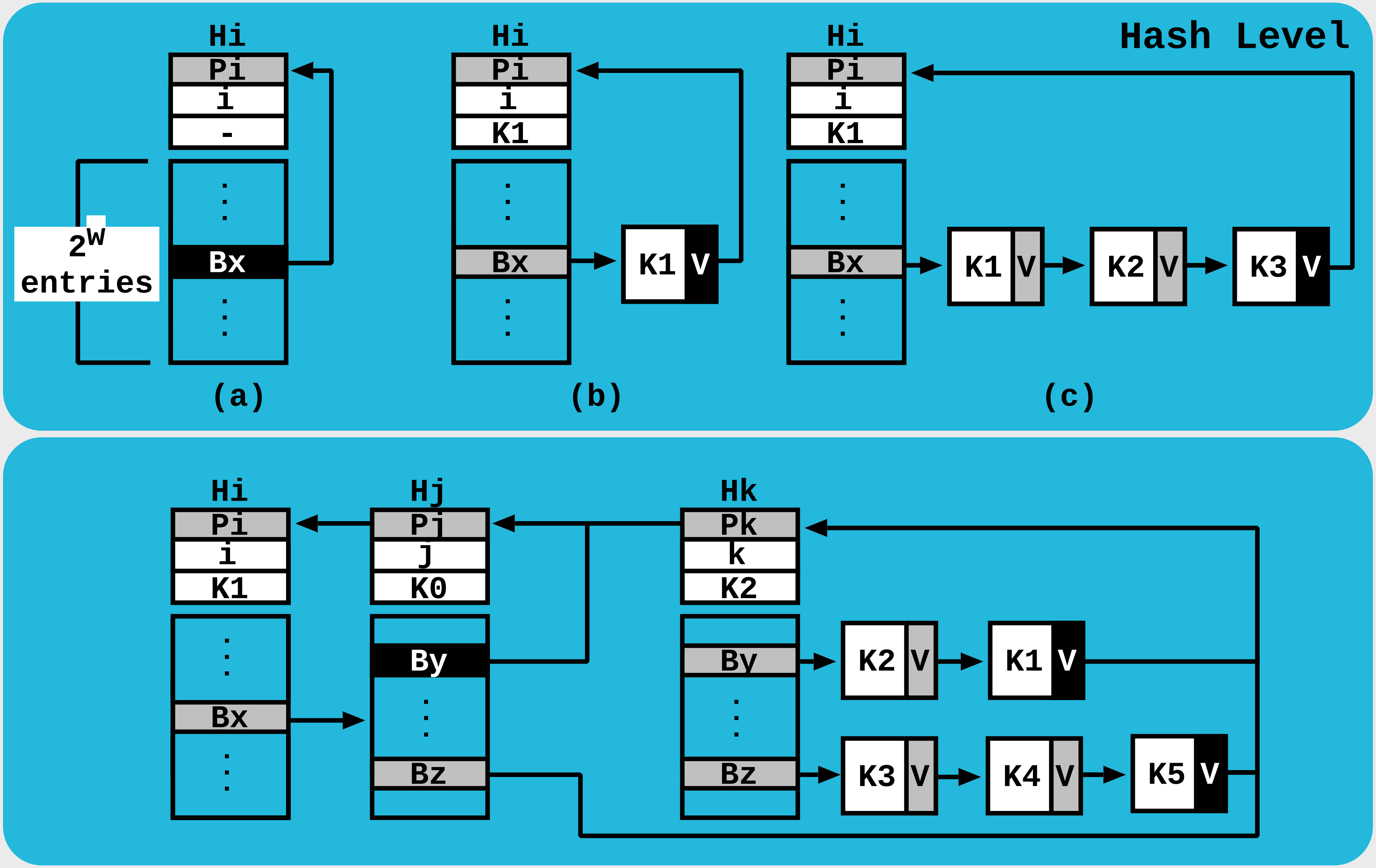




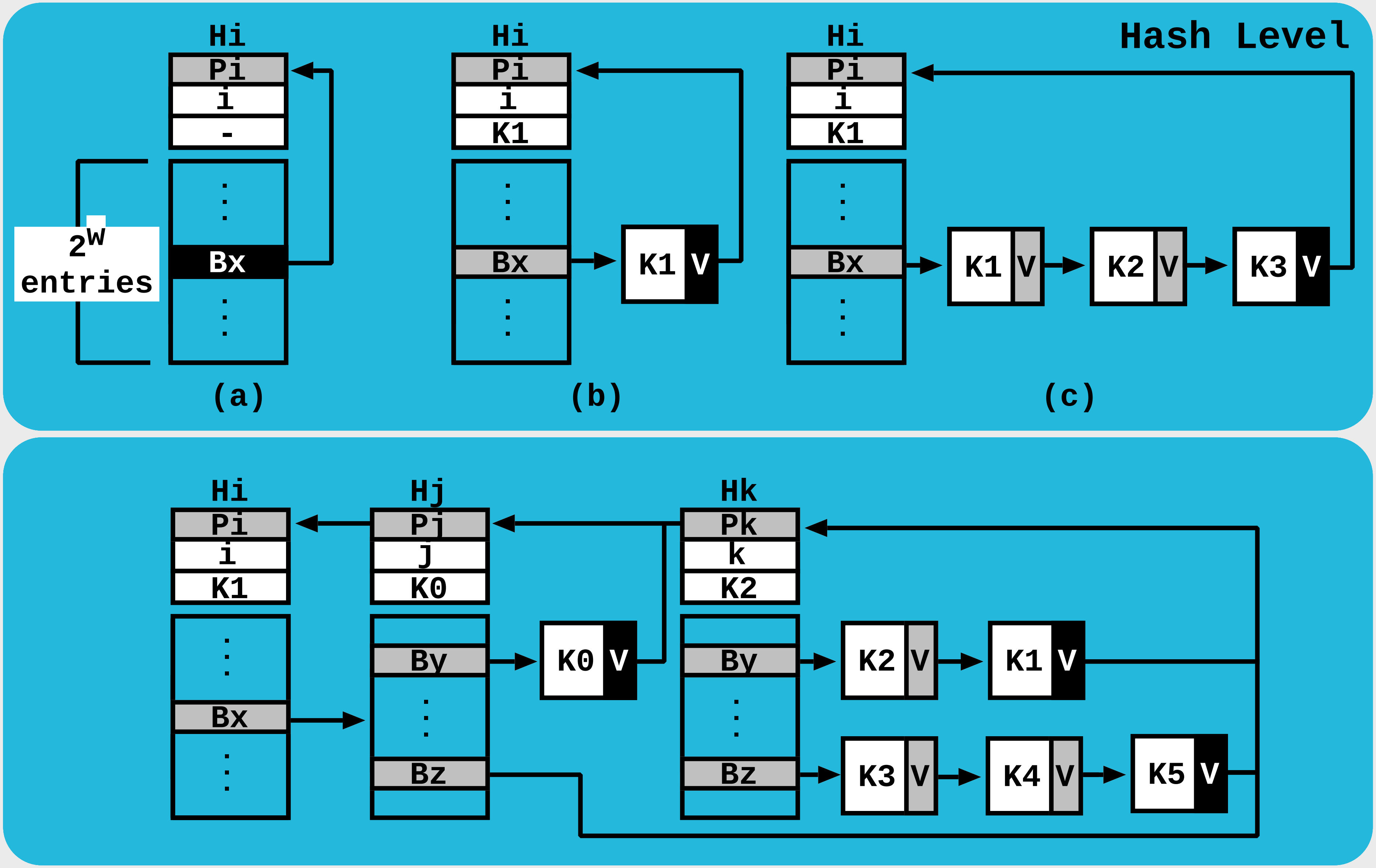
The FP Design - Back Expansion



The FP Design - Back Expansion



The FP Design - Back Expansion



Performance Analysis

- **Hardware**: $32\ (2 * 16)$ core **AMD** with 32 GB of main memory.
- **Software**: Linux **Fedora** 20 with **Oracle's Java Development Kit** 10.0.1.
- **Benchmarks**: Sets of $3 * 10^6$ **randomized keys** with **insert**, **search** and **remove operations** (each **benchmark** had **5 warm up** runs and **20 standard** runs).
- **FP** design: **Expanded** with 3 **valid** nodes and each hash bucket had 16 **entries**.
- **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	CT	$FP_{NSorted}$	FP_{Sorted}	Speedup Ratio (E_{T_1}/E_{T_p}) CH	CS	CT	$FP_{NSorted}$	FP_{Sorted}
1st – Insert: 100% Remove: 0% Search (existing items): 0% Search (missing items): 0%										
1	1,166	2,079	3,285	1,304	1,019					
8	771	560	745	398	697	1.51	3.71	4.41	3.28	1.46
16	729	348	573	313	608	1.60	5.97	5.73	4.17	1.68
24	913	298	588	366	623	1.28	6.98	5.59	3.56	1.64
32	869	276	531	317	765	1.34	7.53	6.19	4.11	1.33
2nd – Insert: 0% Remove: 100% Search (existing items): 0% Search (missing items): 0%										
1	385	2,983	4,178	2,174	1,067					
8	105	905	607	470	633	3.67	3.30	6.88	4.63	1.69
16	104	525	452	350	294	3.70	5.68	9.24	6.21	3.63
24	102	447	436	424	428	3.77	6.67	9.58	5.13	2.49
32	101	455	334	343	191	3.81	6.56	12.51	6.34	5.59
3rd – Insert: 0% Remove: 0% Search (existing items): 100% Search (missing items): 0%										
1	198	2,715	2,043	977	327					
8	79	451	359	196	151	2.51	6.02	5.69	4.98	2.17
16	82	319	228	163	171	2.41	8.51	8.96	5.99	1.91
24	94	325	196	172	174	2.11	8.35	10.42	5.68	1.88
32	90	409	230	162	170	2.20	6.64	8.88	6.03	1.92

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	CT	$FP_{NSorted}$	FP_{Sorted}	CH	CS	CT	$FP_{NSorted}$	FP_{Sorted}
4th – Insert: 0%	Remove: 0%		Search (existing items): 50%			Search (missing items): 50%				
1	135	1,874	1,258	815	288					
8	55	301	241	142	102	2.45	6.23	5.22	5.74	2.82
16	59	201	180	107	96	2.29	9.32	6.99	7.62	3.00
24	66	202	142	113	110	2.05	9.28	8.86	7.21	2.62
32	70	252	161	103	89	1.93	7.44	7.81	7.91	3.24
5th – Insert: 50%	Remove: 0%		Search (existing items): 25%			Search (missing items): 25%				
1	832	3,717	2,736	1,259	786					
8	688	539	493	272	396	1.21	6.90	5.55	4.63	1.98
16	475	341	301	238	351	1.75	10.90	9.09	5.29	2.24
24	519	295	261	222	390	1.60	12.60	10.48	5.67	2.02
32	395	307	236	135	573	2.11	12.11	11.59	9.33	1.37
6th – Insert: 20%	Remove: 10%		Search (existing items): 35%			Search (missing items): 35%				
1	505	3,709	2,457	996	497					
8	183	566	396	206	270	2.76	6.55	6.20	4.83	1.84
16	88	334	250	145	310	5.74	11.10	9.83	6.87	1.60
24	106	283	247	185	271	4.76	13.11	9.95	5.38	1.83
32	96	298	244	146	187	5.26	12.45	10.07	6.82	2.66

Conclusions and Further Work

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

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

Conclusions and Further Work

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

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

- **Experimental results** show that the design:
  - Is quite **competitive** when compared against other **state-of-the-art** designs implemented in Java.

Conclusions and Further Work

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

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

- **Experimental results** show that the design:
  - Is quite **competitive** when compared against other **state-of-the-art** designs implemented in Java.

- **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/ffps***
FCT grant: *SFRH/BPD/108018/2015*