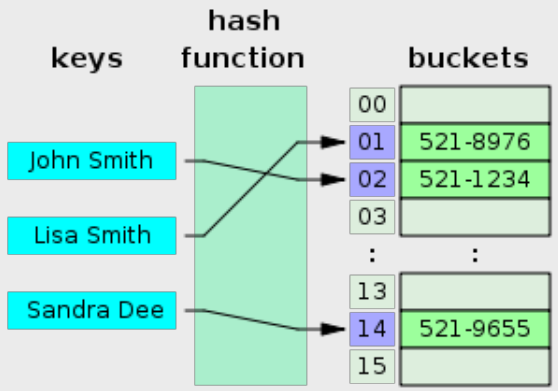
# Towards an Elastic Lock-Free Hash Trie Design

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

# Hash Maps

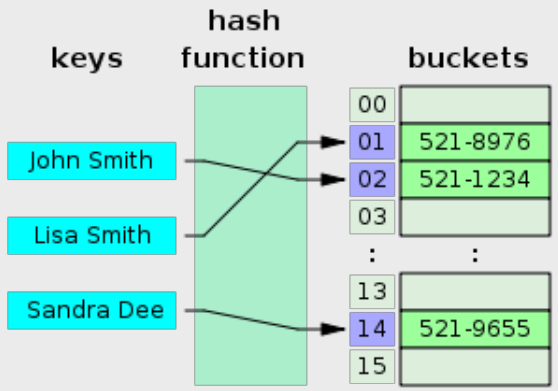
- **Hash maps** are **useful** to **store information** that can be organized as **pairs (K, C)**, where **K** is an identifier (or a **key**) and **C** is the **associated content**.




A small phone book as a hash map.

# 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**.

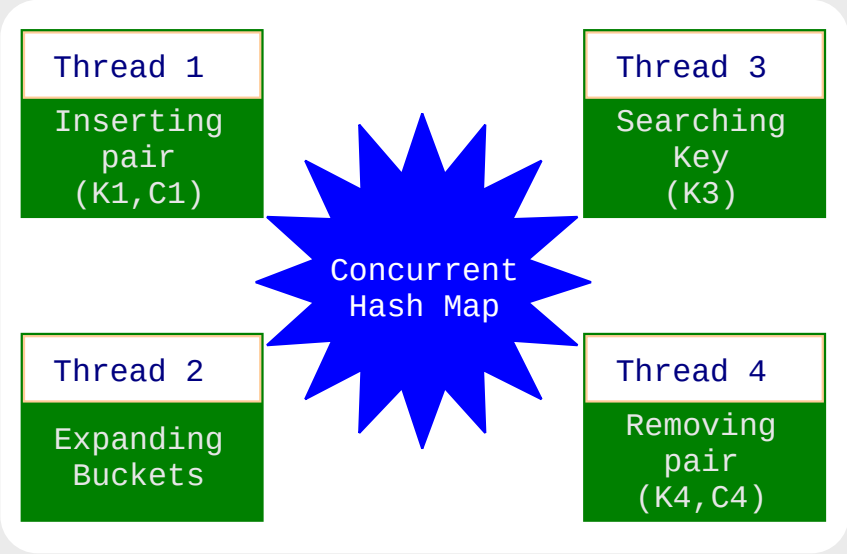



A small phone book as a hash map.

- 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** $\Rightarrow$ **elasticity** (or **elastic hashing**) is the ability to support both operations.

# Concurrent Hash Maps

- **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**), Concurrent Tries (**CT**) and Fixed-size Persistent Lock-free Hash Map (**FP**).

# 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**), Concurrent Tries (**CT**) and Fixed-size Persistent Lock-free Hash Map (**FP**).
- However, to the best of our knowledge, **non** of the **existent designs** combine five **properties**: **Lock-Free Progress**, **Persistent Memory References**, **Fixed-Size Data Structures**, **Store Sorted Keys** and **Elasticity**.

| Properties / Designs | CH | CS | NB | CT | FP |
|---|---|---|---|---|---|
| **Lock-Free Progress** | ✗ | ✓ | ✓ | ✓ | ✓ |
| **Persistent Memory References** | ✗ | ✓ | ✓ | ✗ | ✓ |
| **Fixed-Size Data Structures** | ✗ | - | ✗ | ✗ | ✓ |
| **Store Sorted Keys** | ✗ | ✓ | ✗ | ✗ | ✓ |
| **Elasticity (Elastic Hashing)** | ✗ | - | ✗ | ✓ | ✗ |

# Our Motivation

- However, to the best of our knowledge, **non** of the **existent designs** combine five **properties**: **Lock-Free Progress**, **Persistent Memory References**, **Fixed-Size Data Structures**, **Store Sorted Keys** and **Elasticity**.

| Properties / Designs | CH | CS | NB | CT | FP |
|---|---|---|---|---|---|
| **Lock-Free Progress** | ✗ | ✓ | ✓ | ✓ | ✓ |
| **Persistent Memory References** | ✗ | ✓ | ✓ | ✗ | ✓ |
| **Fixed-Size Data Structures** | ✗ | - | ✗ | ✗ | ✓ |
| **Store Sorted Keys** | ✗ | ✓ | ✗ | ✗ | ✓ |
| **Elasticity (Elastic Hashing)** | ✗ | - | ✗ | ✓ | ✗ |

- In **this talk** we will:
  - give a brief overview about the **Lock-Free Progress** property.

# Our Motivation

- However, to the best of our knowledge, **non** of the **existent designs** combine five **properties**: **Lock-Free Progress**, **Persistent Memory References**, **Fixed-Size Data Structures**, **Store Sorted Keys** and **Elasticity**.

| Properties / Designs | CH | CS | NB | CT | FP |
|---|---|---|---|---|---|
| **Lock-Free Progress** | ✗ | ✓ | ✓ | ✓ | ✓ |
| **Persistent Memory References** | ✗ | ✓ | ✓ | ✗ | ✓ |
| **Fixed-Size Data Structures** | ✗ | - | ✗ | ✗ | ✓ |
| **Store Sorted Keys** | ✗ | ✓ | ✗ | ✗ | ✓ |
| **Elasticity (Elastic Hashing)** | ✗ | - | ✗ | ✓ | ✗ |

- In **this talk** we will:
  - give a brief overview about **Lock-Free Progress** property.
  - present the internals of the **FP** design.

# Our Motivation

- However, to the best of our knowledge, **non** of the **existent designs** combine five **properties**: **Lock-Free Progress**, **Persistent Memory References**, **Fixed-Size Data Structures**, **Store Sorted Keys** and **Elasticity**.

| Properties / Designs | CH | CS | NB | CT | FP |
|---|---|---|---|---|---|
| **Lock-Free Progress** | ✗ | ✓ | ✓ | ✓ | ✓ |
| **Persistent Memory References** | ✗ | ✓ | ✓ | ✗ | ✓ |
| **Fixed-Size Data Structures** | ✗ | - | ✗ | ✗ | ✓ |
| **Store Sorted Keys** | ✗ | ✓ | ✗ | ✗ | ✓ |
| **Elasticity (Elastic Hashing)** | ✗ | - | ✗ | ✓ | ✓ |

- In **this talk** we will:
  - give a brief overview about **Lock-Free Progress** property.
  - present the internals of the **FP** design.
  - show how to extend the **FP** design to support **Elasticity**.

# Our Motivation

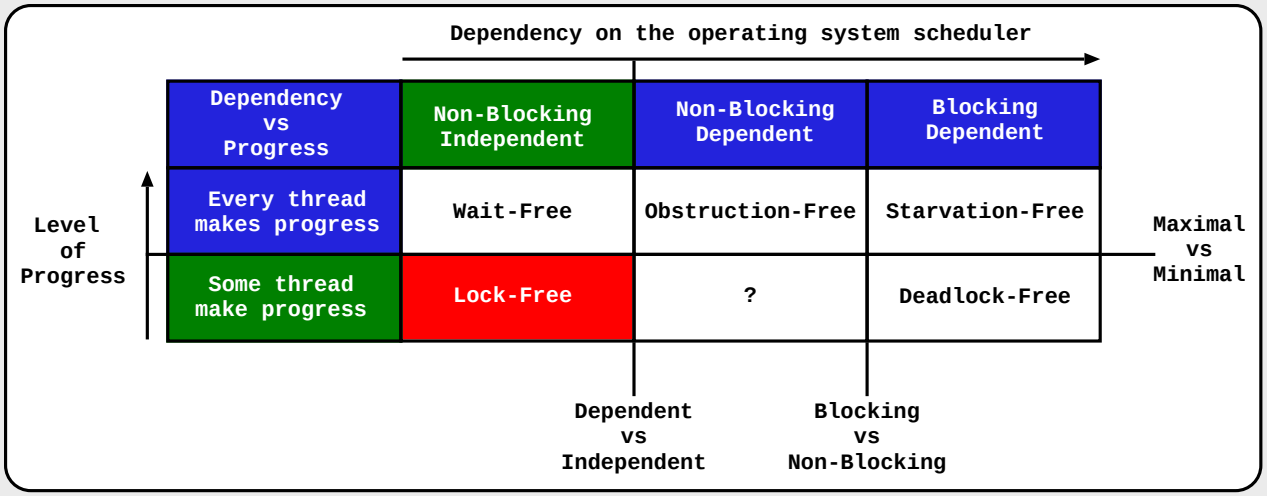
- However, to the best of our knowledge, **non** of the **existent designs** combine five **properties**: **Lock-Free Progress**, **Persistent Memory References**, **Fixed-Size Data Structures**, **Store Sorted Keys** and **Elasticity**.

| Properties / Designs | CH | CS | NB | CT | FP |
|---|---|---|---|---|---|
| **Lock-Free Progress** | ✗ | ✓ | ✓ | ✓ | ✓ |
| **Persistent Memory References** | ✗ | ✓ | ✓ | ✗ | ✓ |
| **Fixed-Size Data Structures** | ✗ | - | ✗ | ✗ | ✓ |
| **Store Sorted Keys** | ✗ | ✓ | ✗ | ✗ | ✓ |
| **Elasticity (Elastic Hashing)** | ✗ | - | ✗ | ✓ | ✓ |

- In **this talk** we will:
  - give a brief overview about **Lock-Free Progress** property.
  - present the internals of the **FP** design.
  - show how to extend the **FP** design to support **Elasticity**.
  - show a **performance** analysis **comparison** (will skip the **CH** and **NB** designs).

# Lock-Free Progress

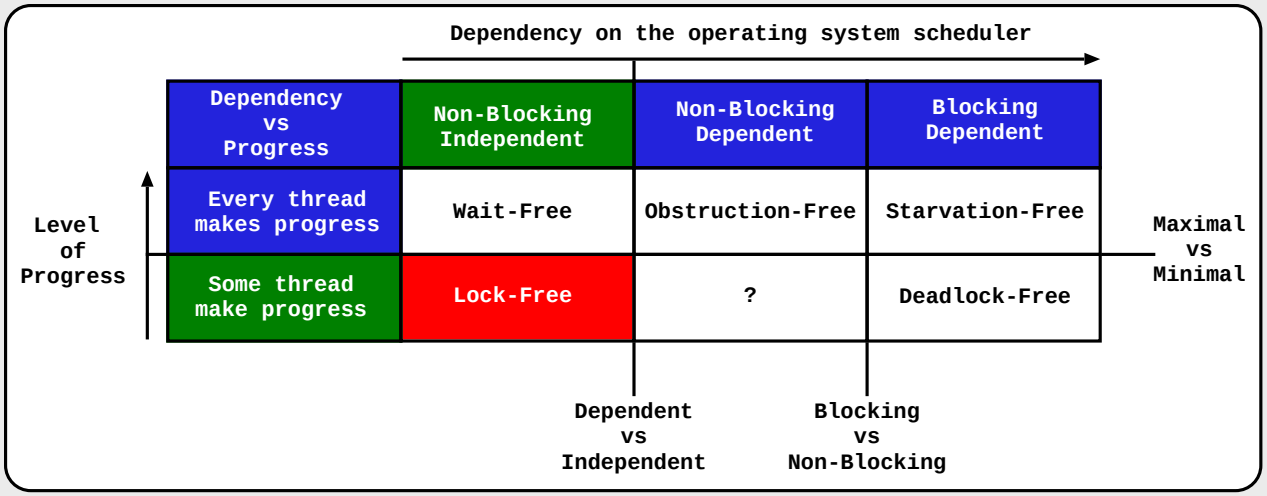
- **Lock-Free linearizable objects** permit a **greater concurrency** since **semantically consistent** (non-interfering) operations **may execute in parallel**.




- **Lock-free** guarantees then that, on **every instant** of the execution of operations (between their invocation and their response), **at least one thread** is doing **progress** on its work.

# Lock-Free Progress

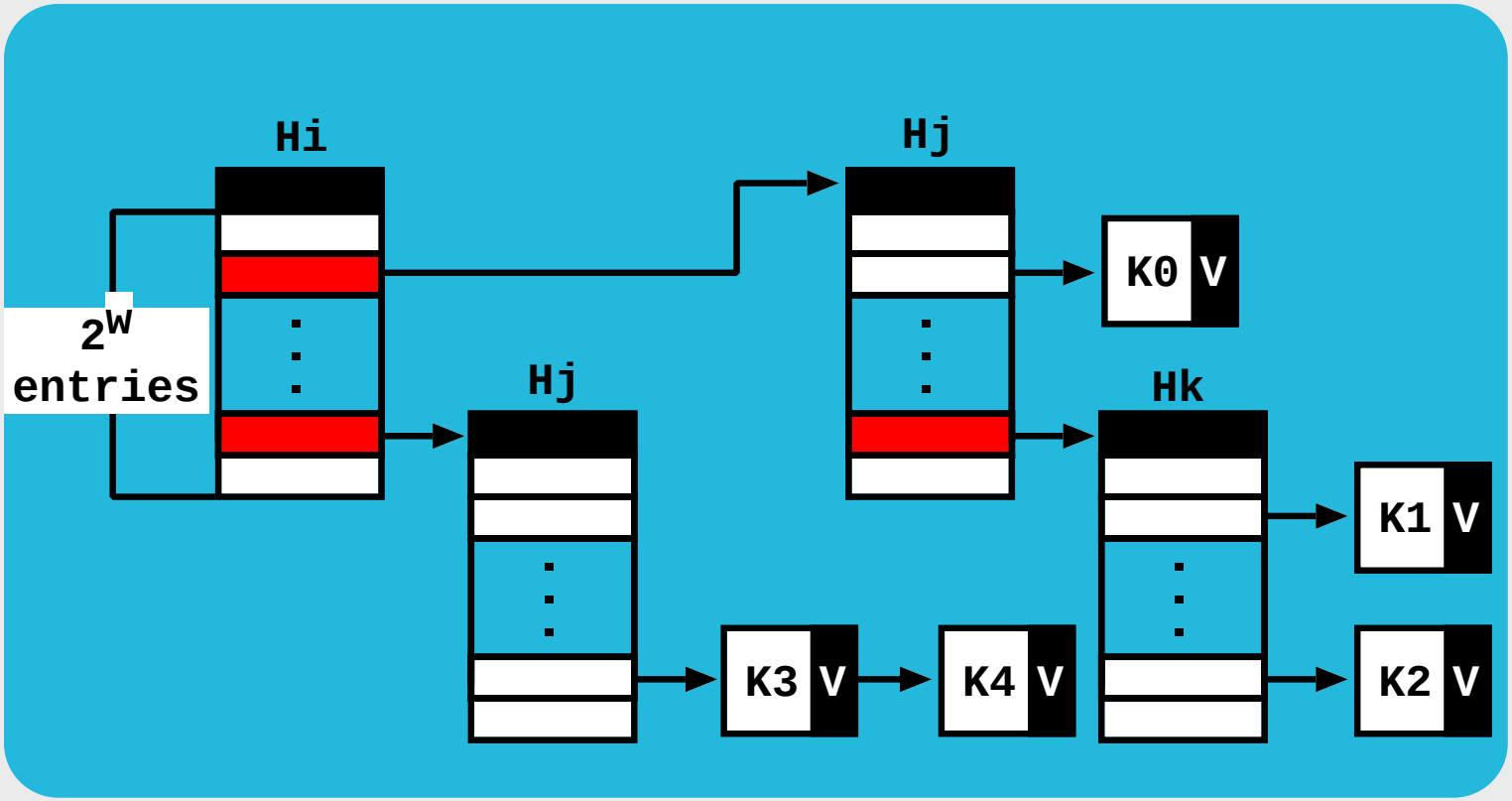
- **Lock-Free linearizable objects** permit a **greater concurrency** since **semantically consistent** (non-interfering) operations **may execute in parallel**.



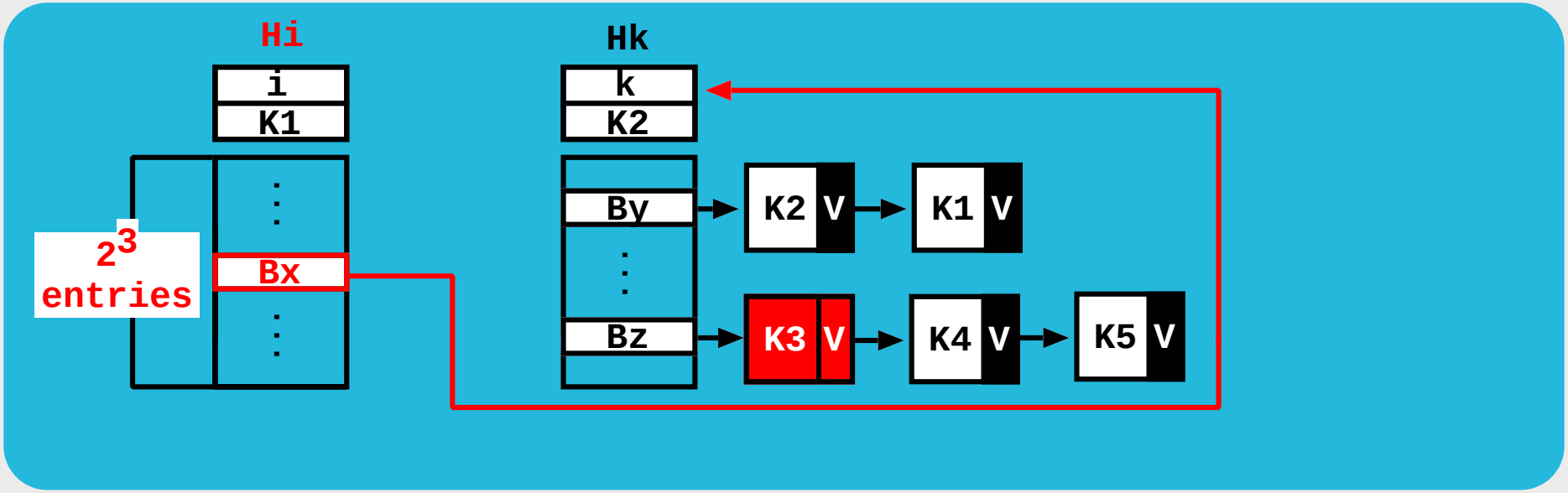

- **Lock-free** guarantees then that, on **every instant** of the execution of operations (between their invocation and their response), **at least one thread** is doing **progress** on its work.
- At the **implementation level**, they take advantage of the **CAS (Compare-and-Swap) atomic operation** (intrinsically **thread safe**), that nowadays **can be found** in many common **hardware** architectures.

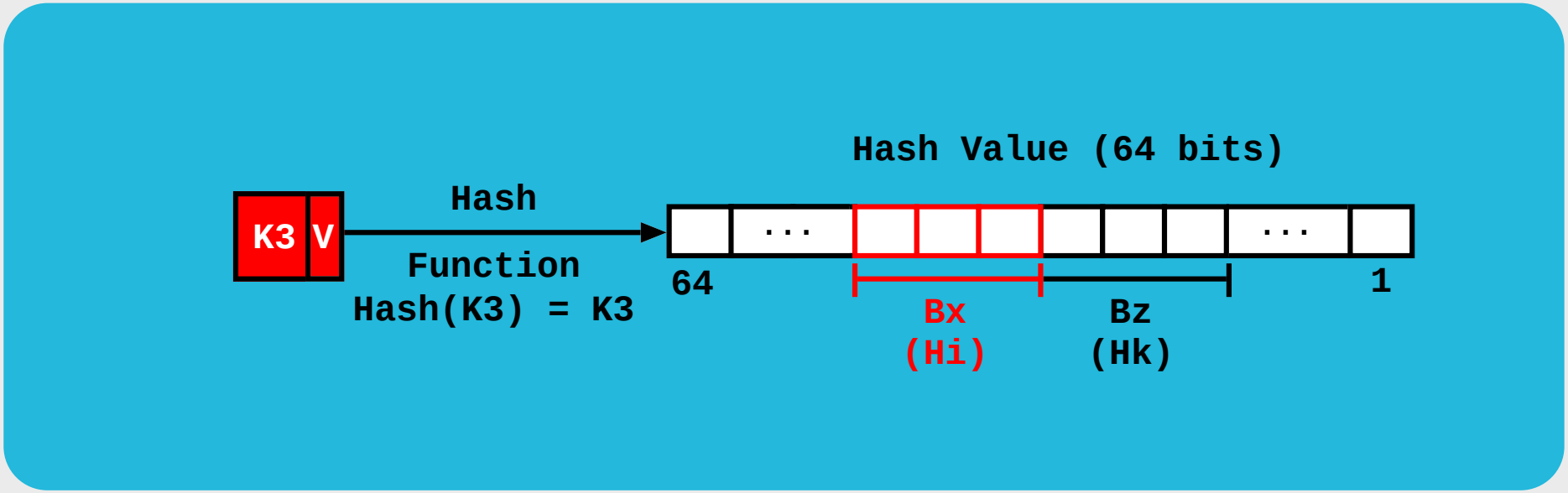
# FP Design - Key Ideas

- **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)**.

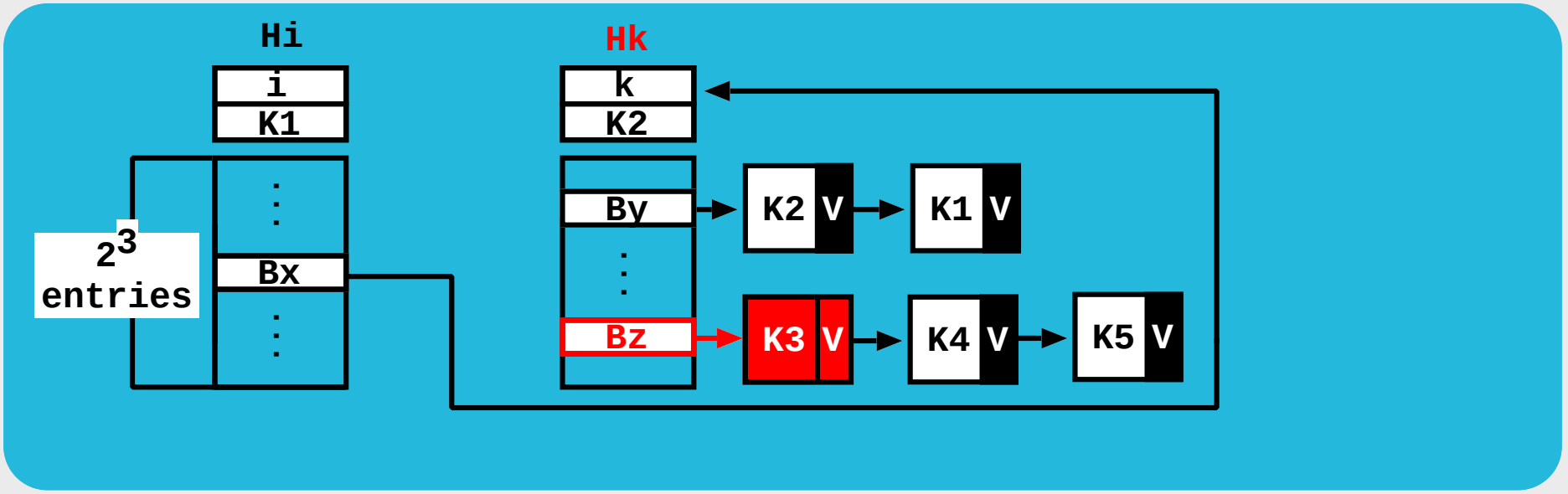


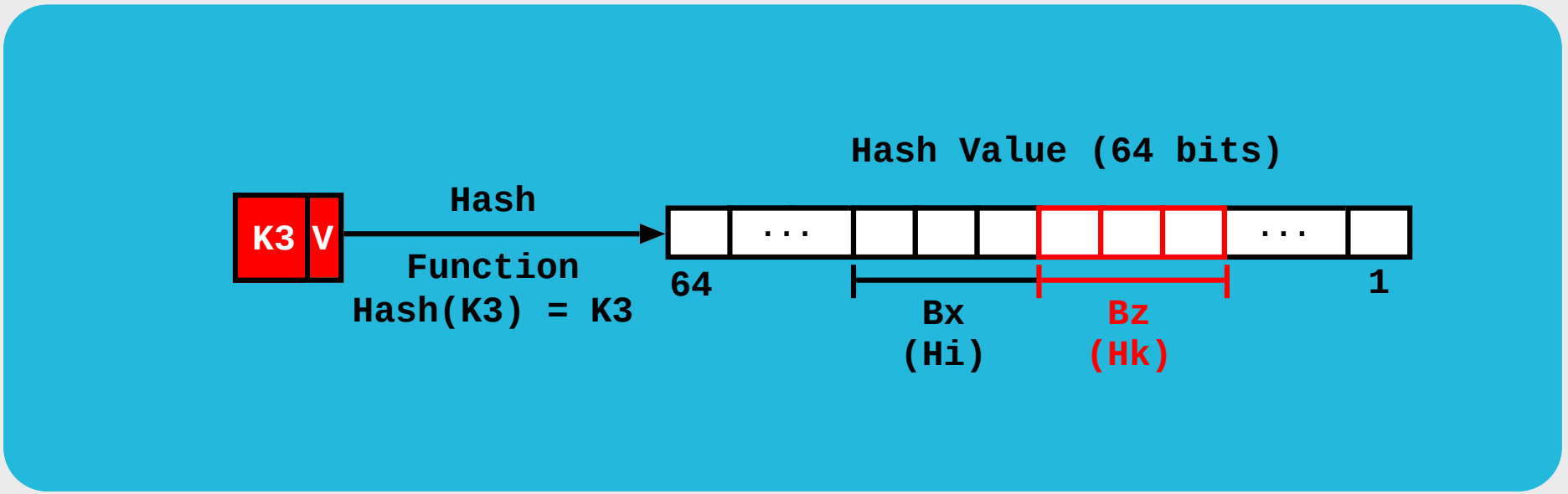
# FP Design - Searching for K3



# FP Design - Searching for K3



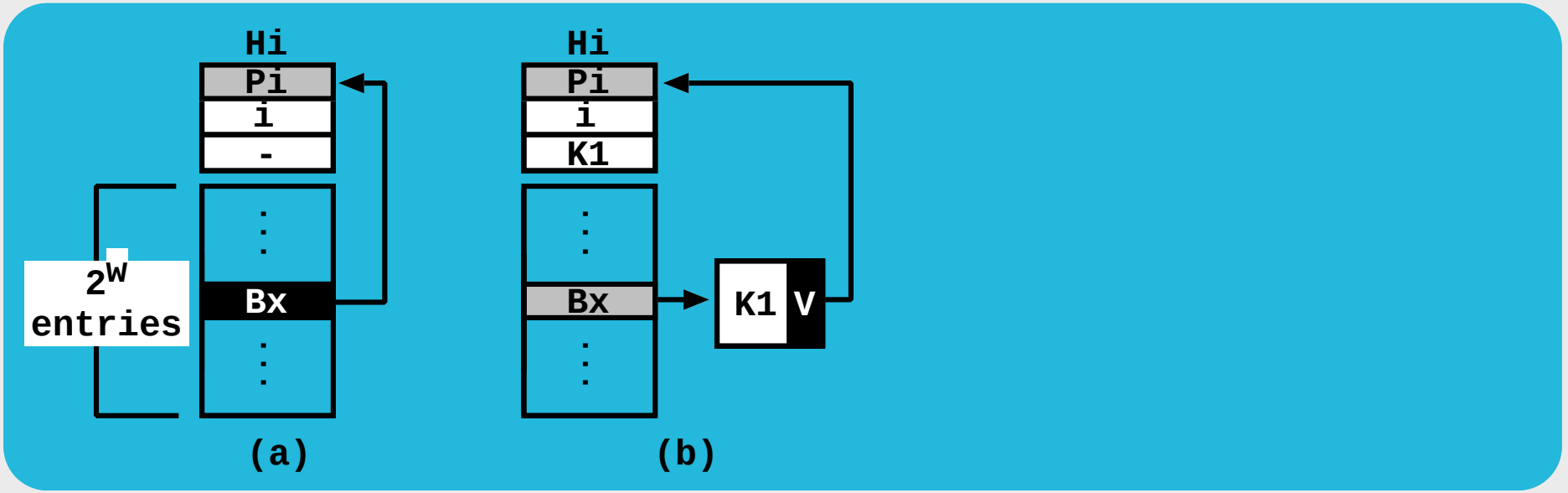



# FP Design - Internals

- 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.

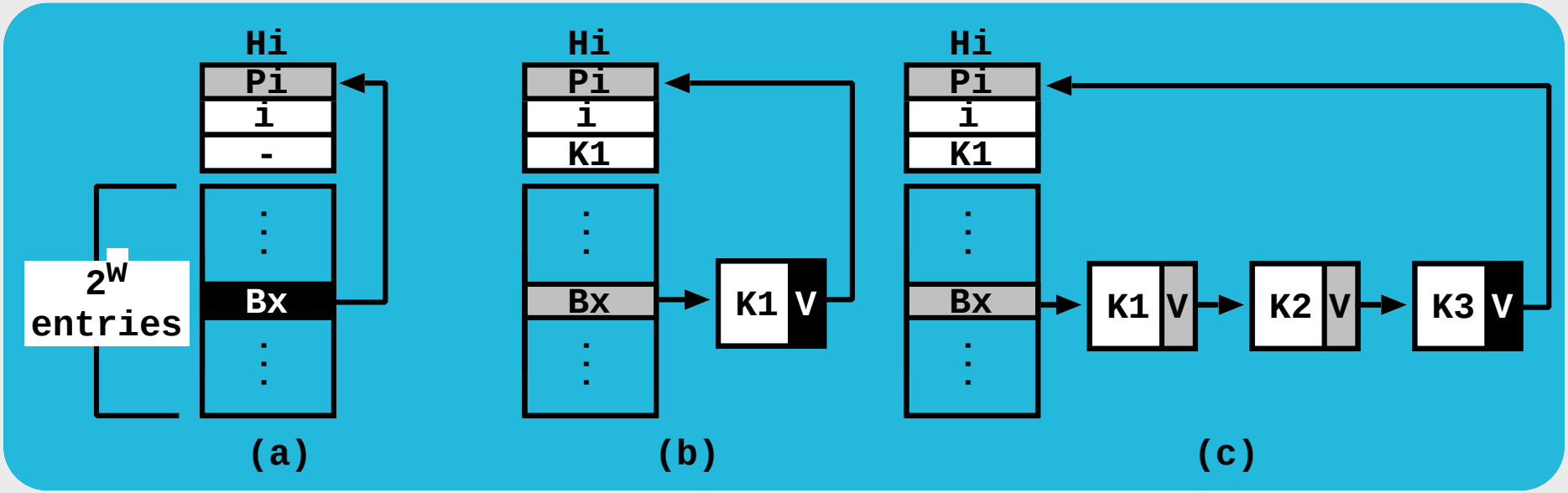
# FP Design - Internals

- 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.

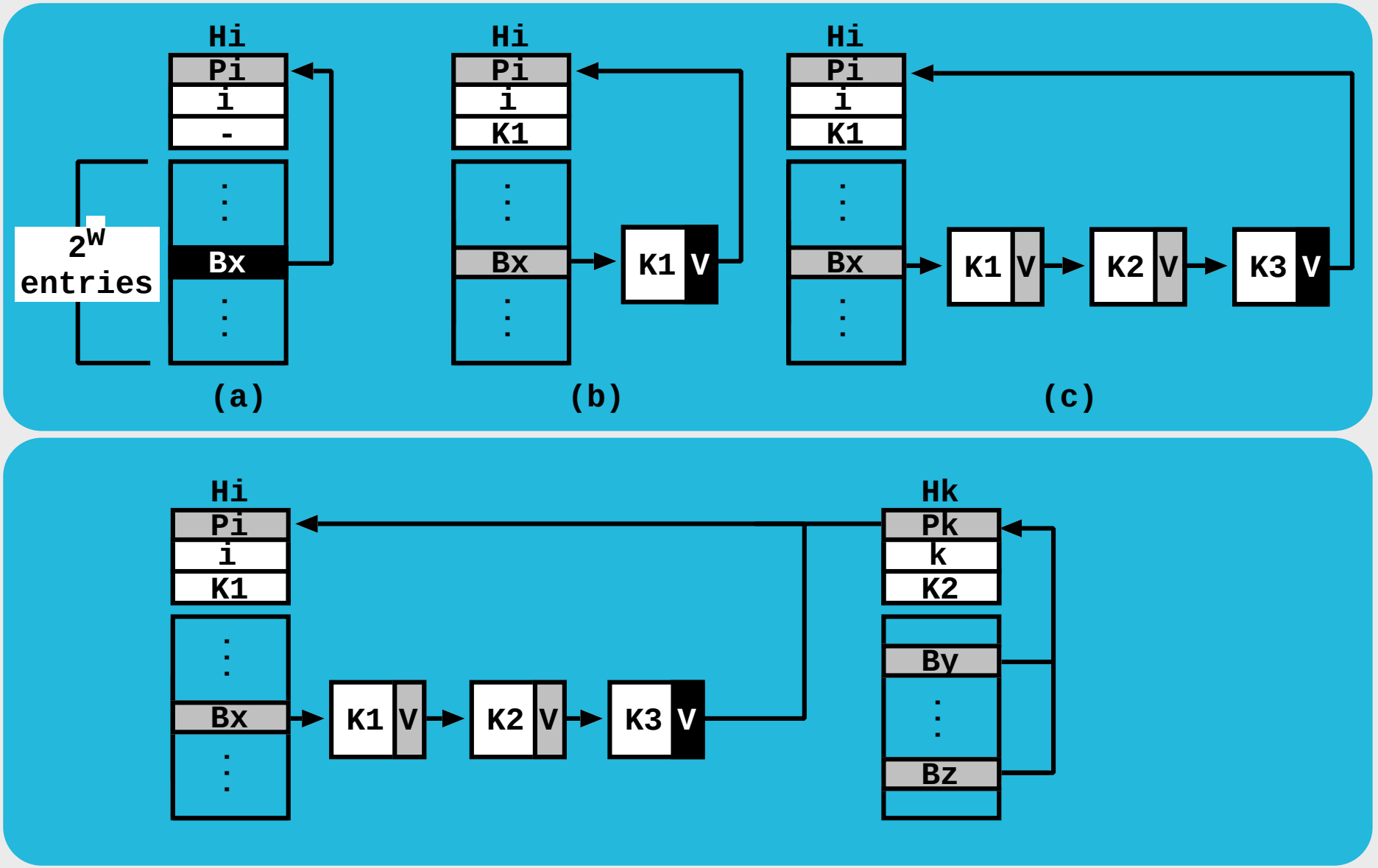


# FP Design - Internals

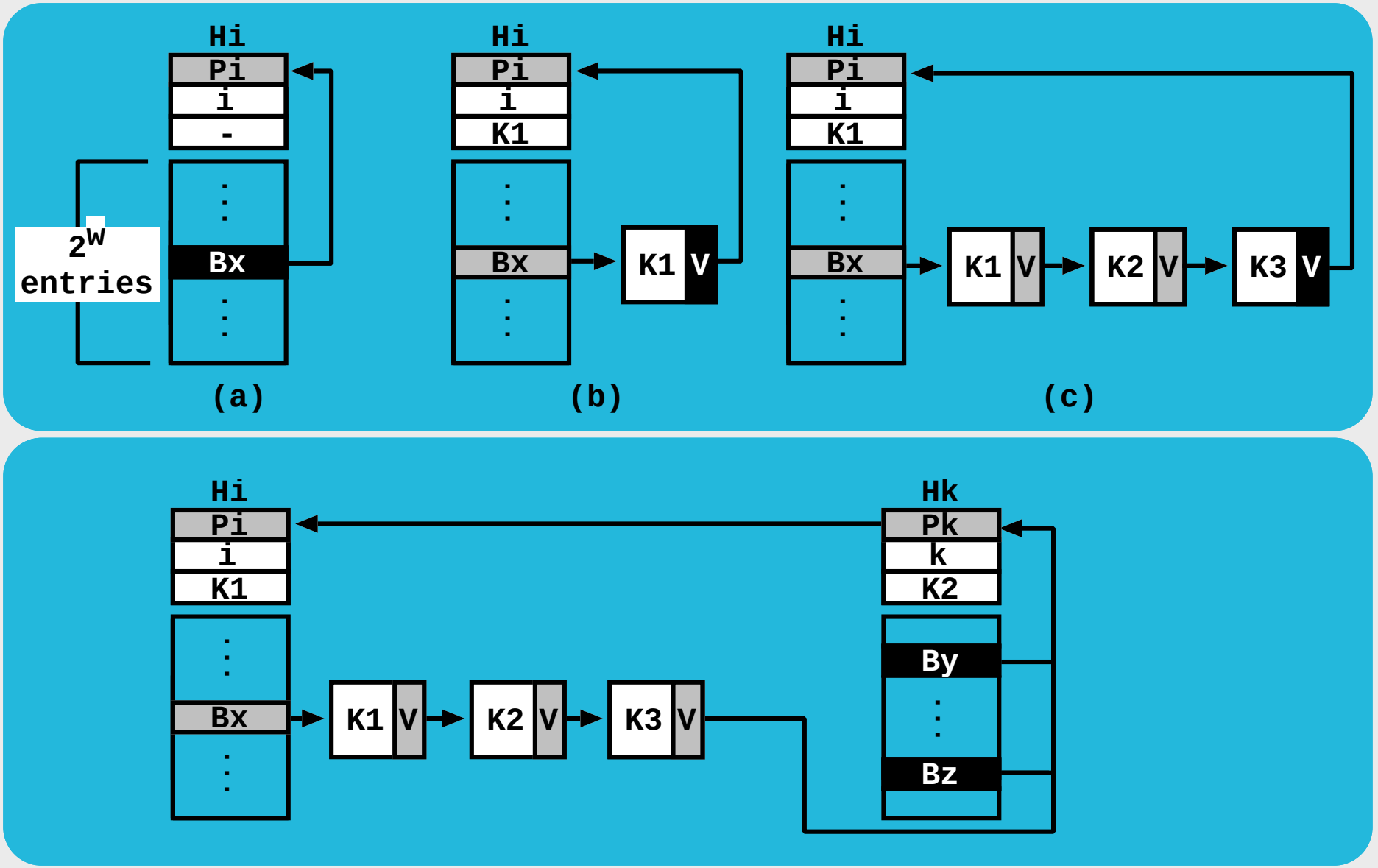
- To support **multithreading**, our design allows **threads** to:
  - **Recover from preemption**, by using a **previous field (Pi)** to traverse the **hash buckets** backwards.
  - **Identify chains**, by using a **back-reference** on the end of each chain.
  - **Maintain consistency**, by using **CAS**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



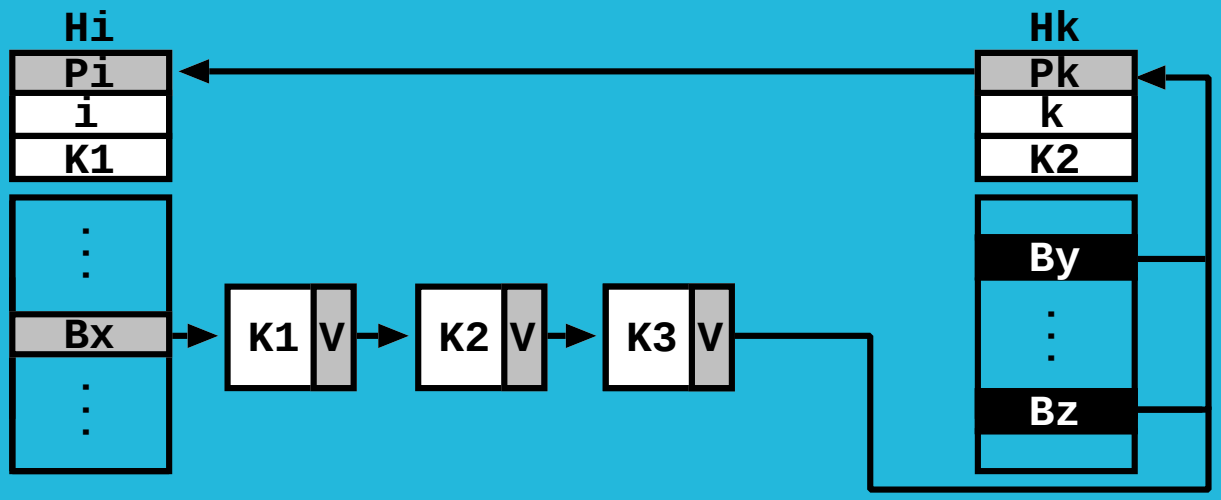
# FP Design - Front Expansion



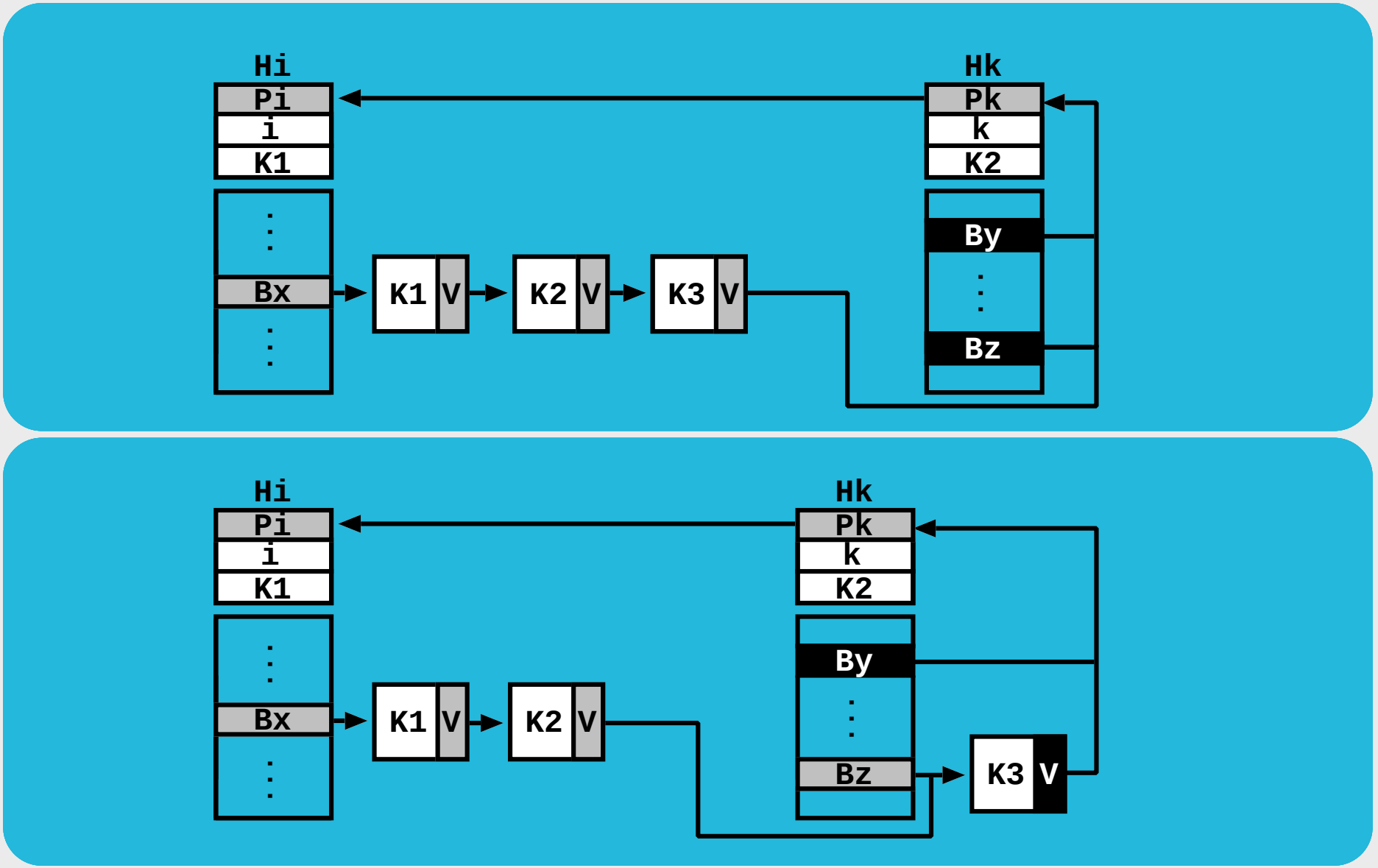
# FP Design - Front Expansion



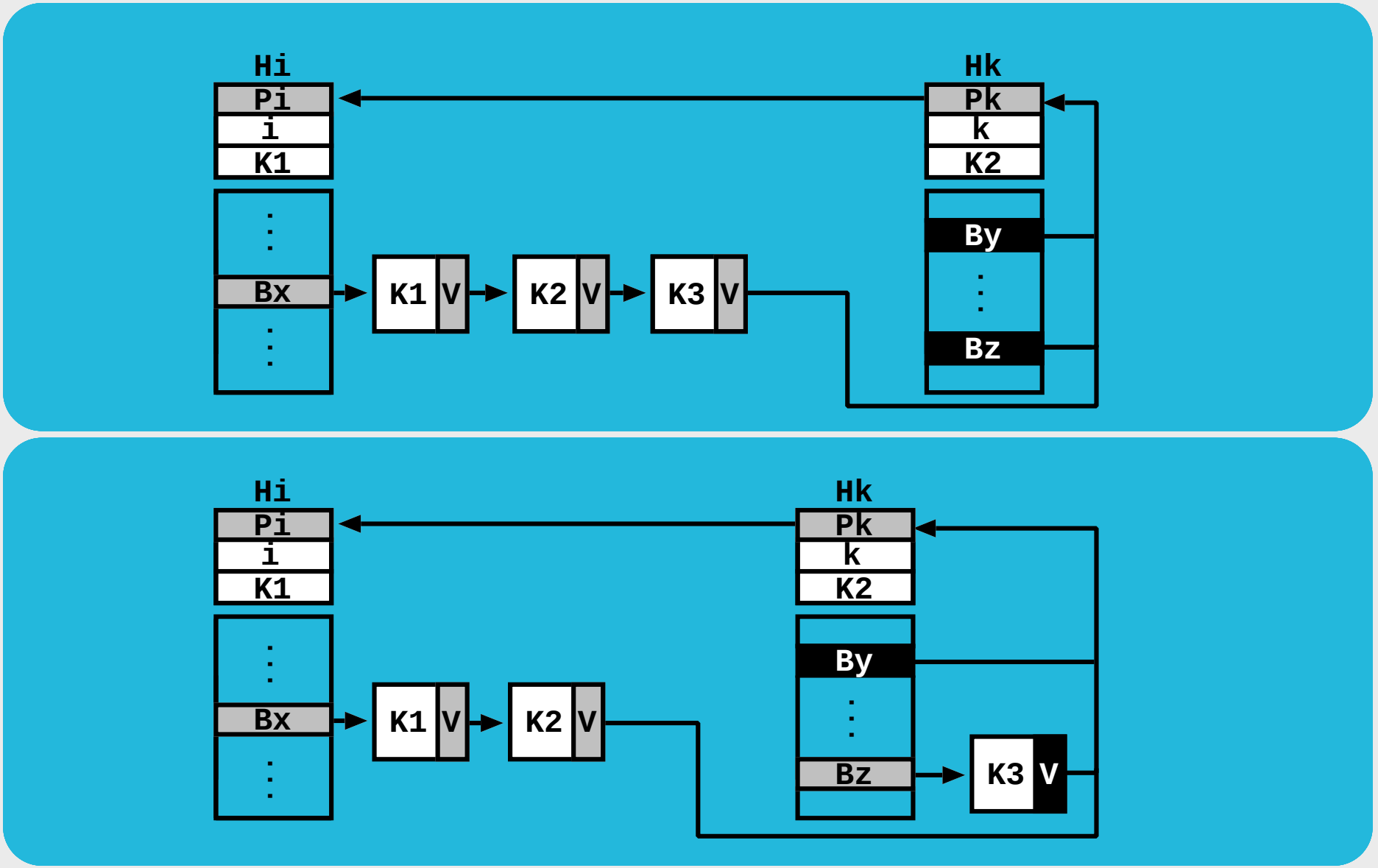
# FP Design - Front Expansion



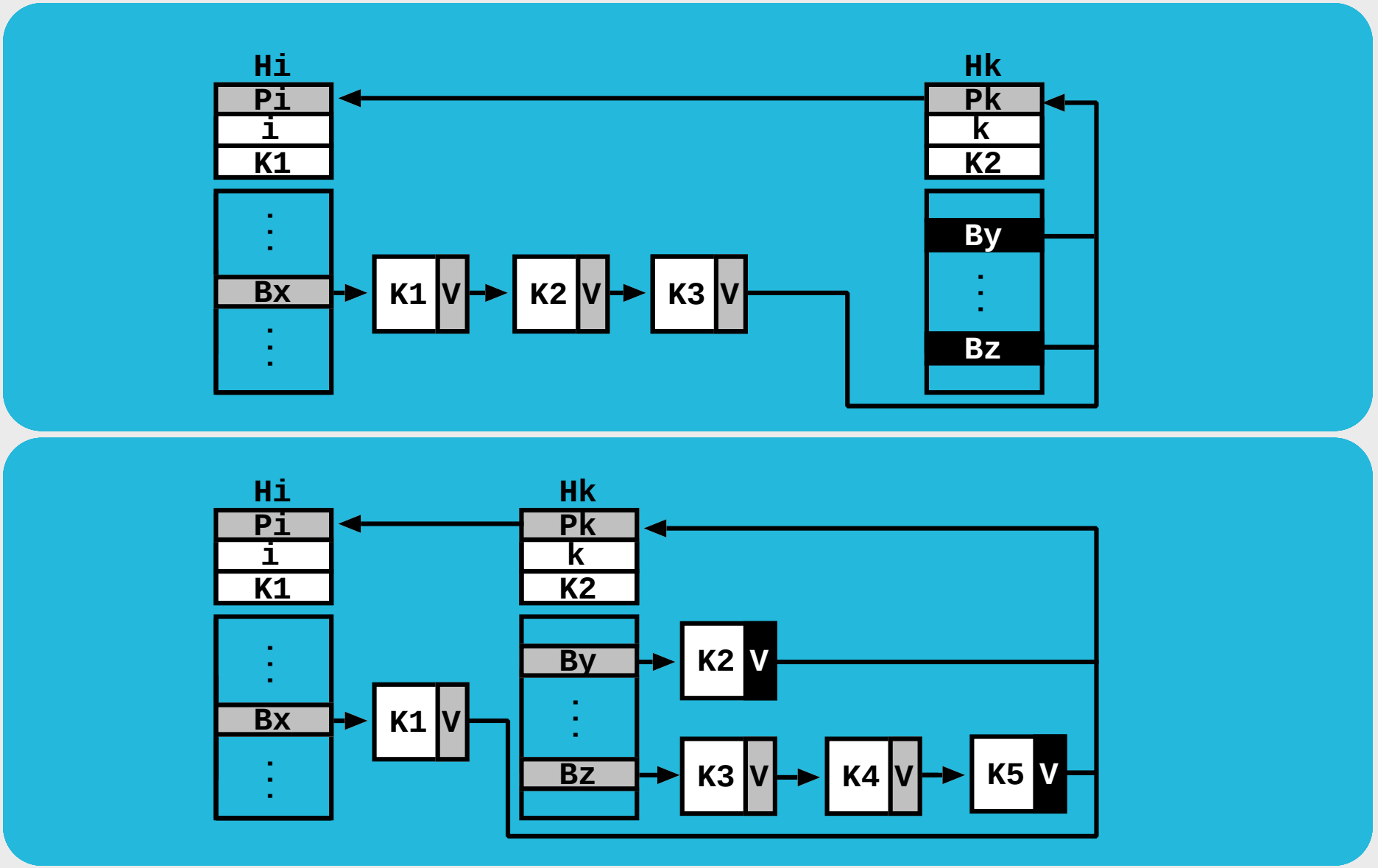
# FP Design - Front Expansion



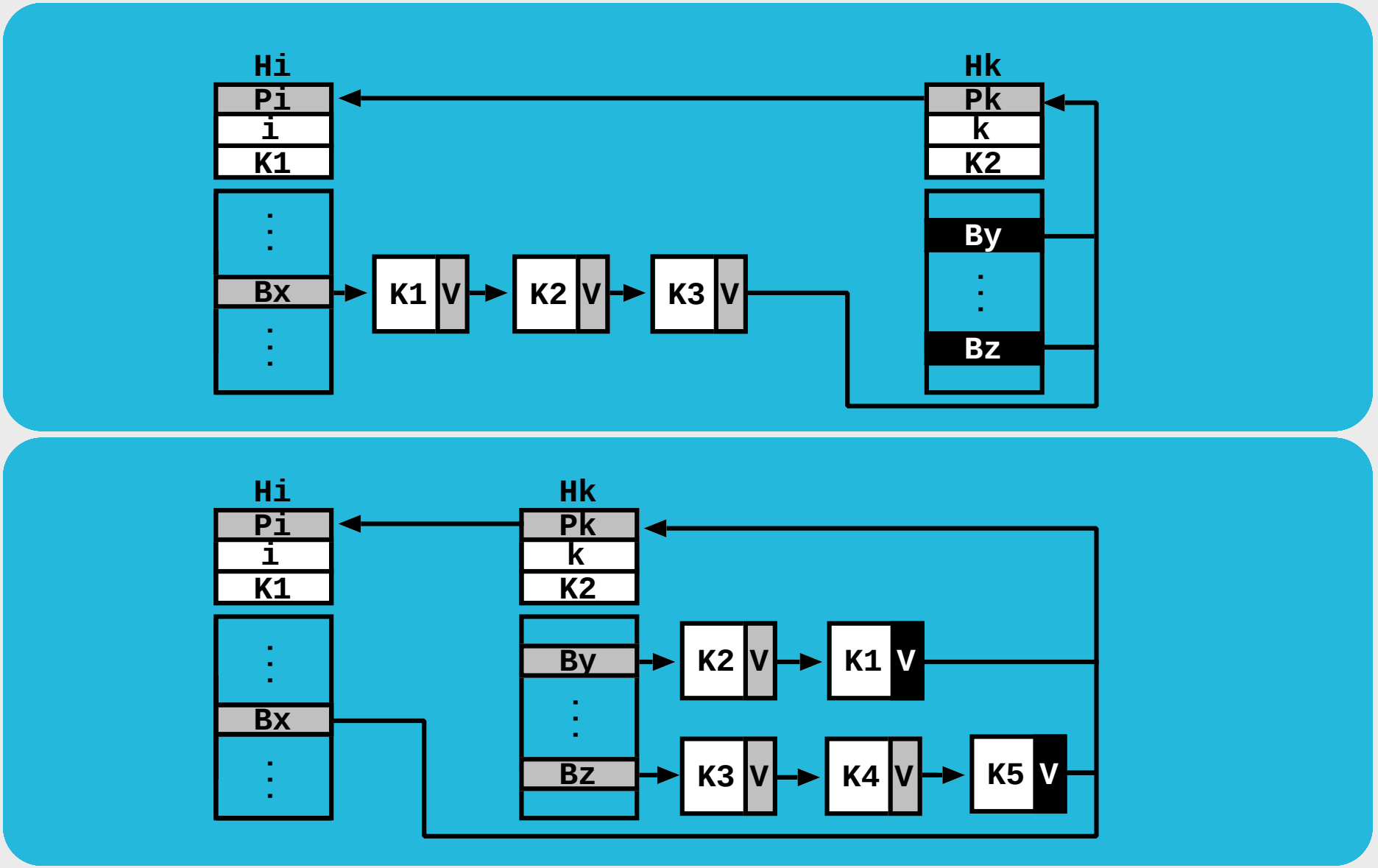
# FP Design - Front Expansion



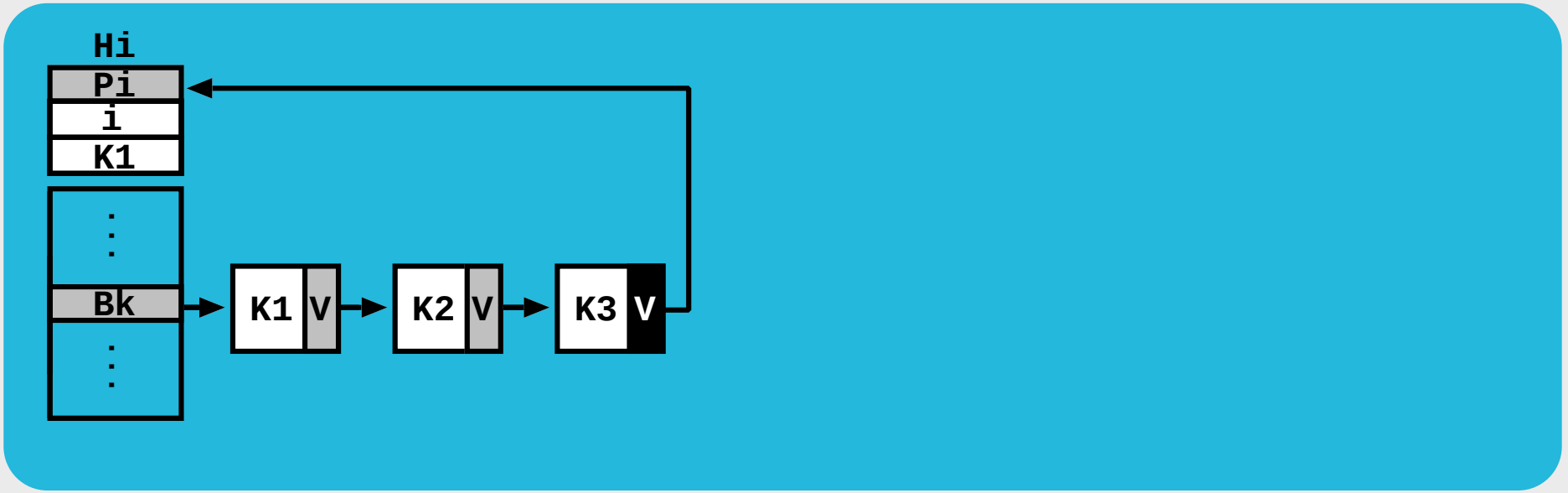
# FP Design - Front Expansion



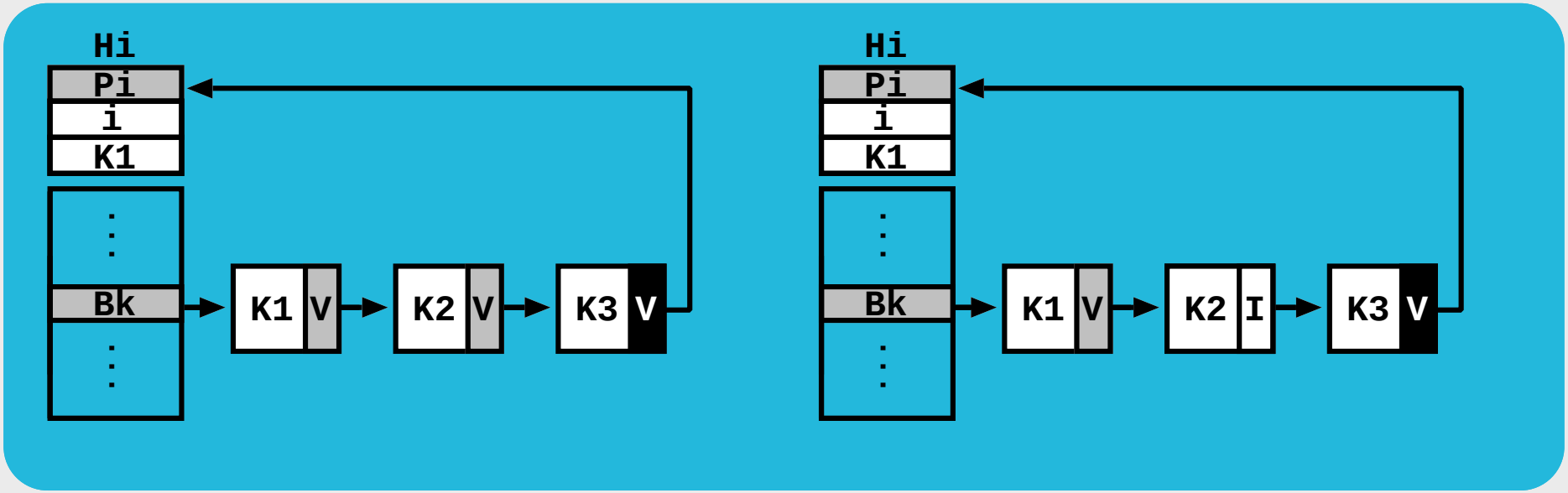
# FP Design - Front Expansion



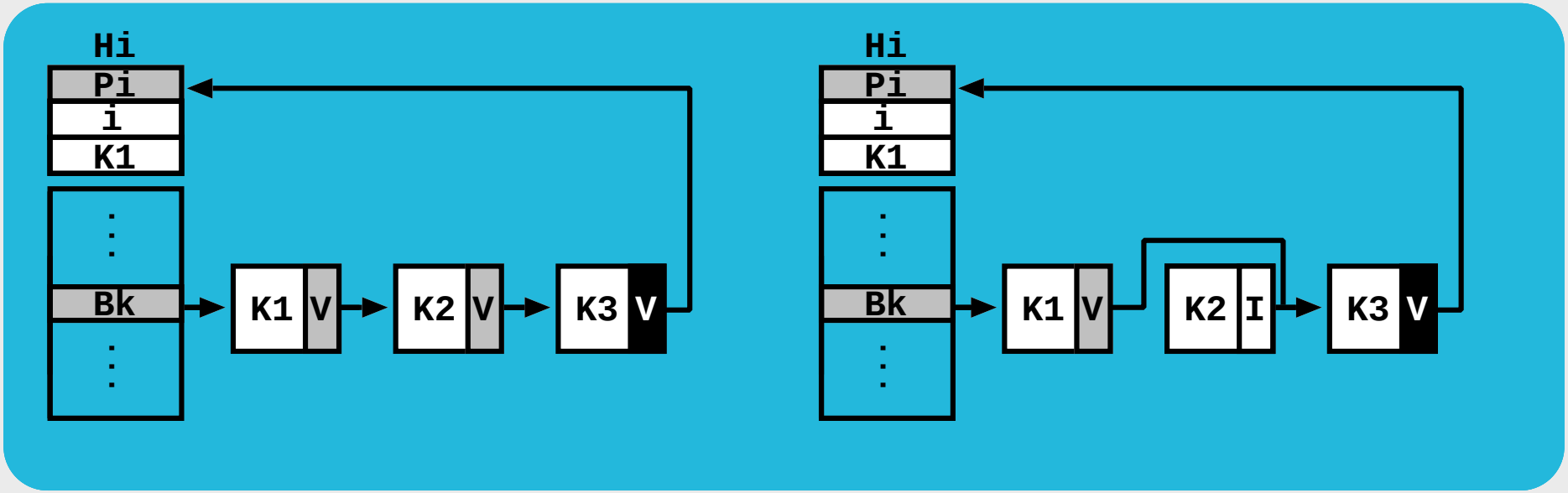
# FP Design - Removal of a Key



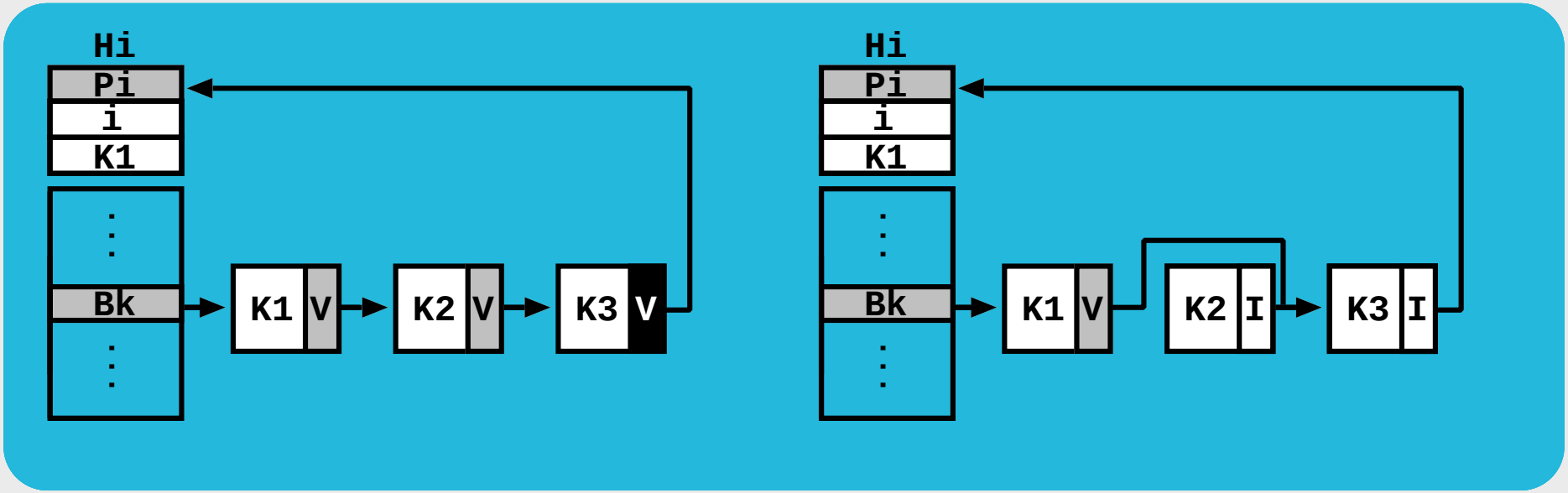
# FP Design - Removal of a Key



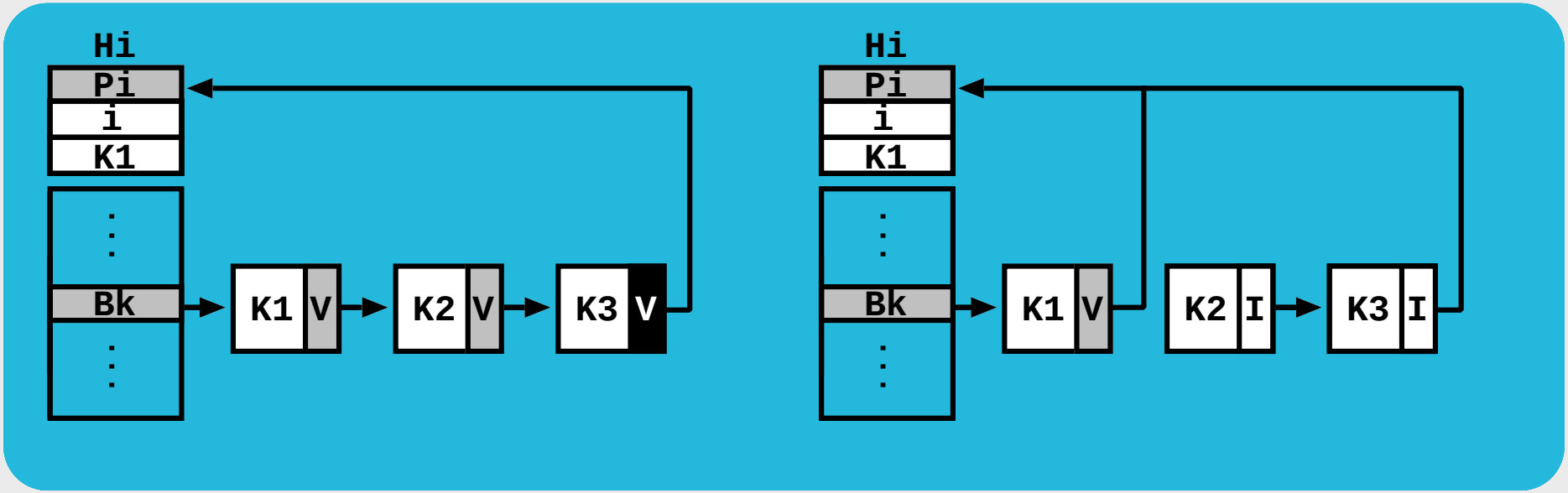
# FP Design - Removal of a Key



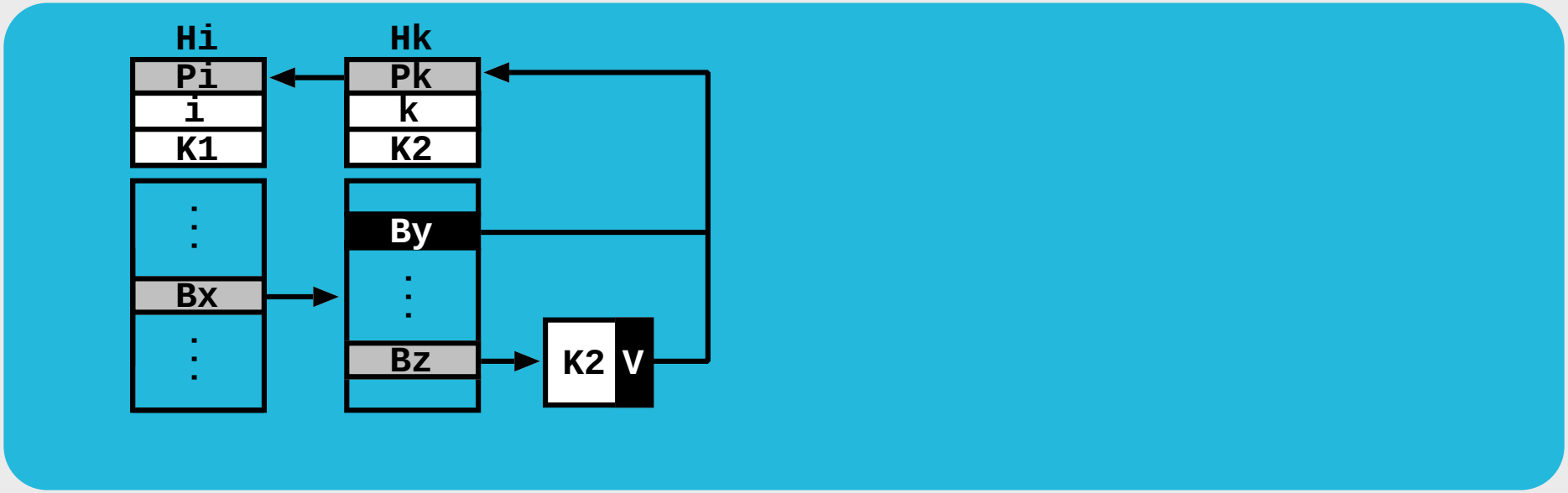
# FP Design - Removal of a Key



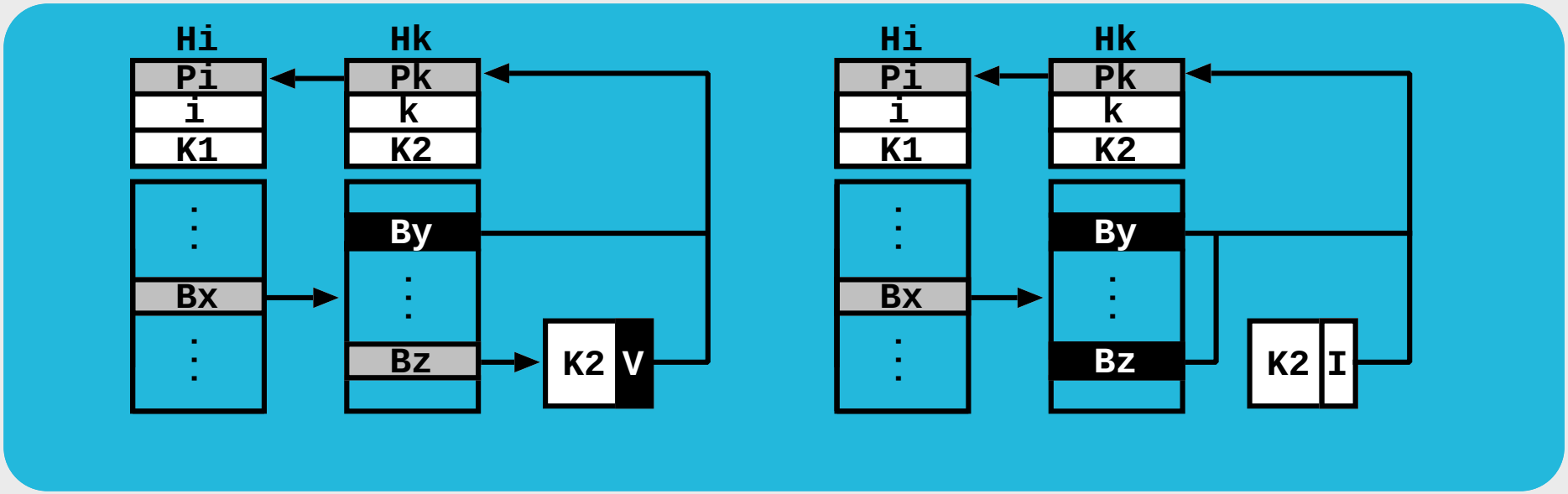
# FP Design - Removal of a Key



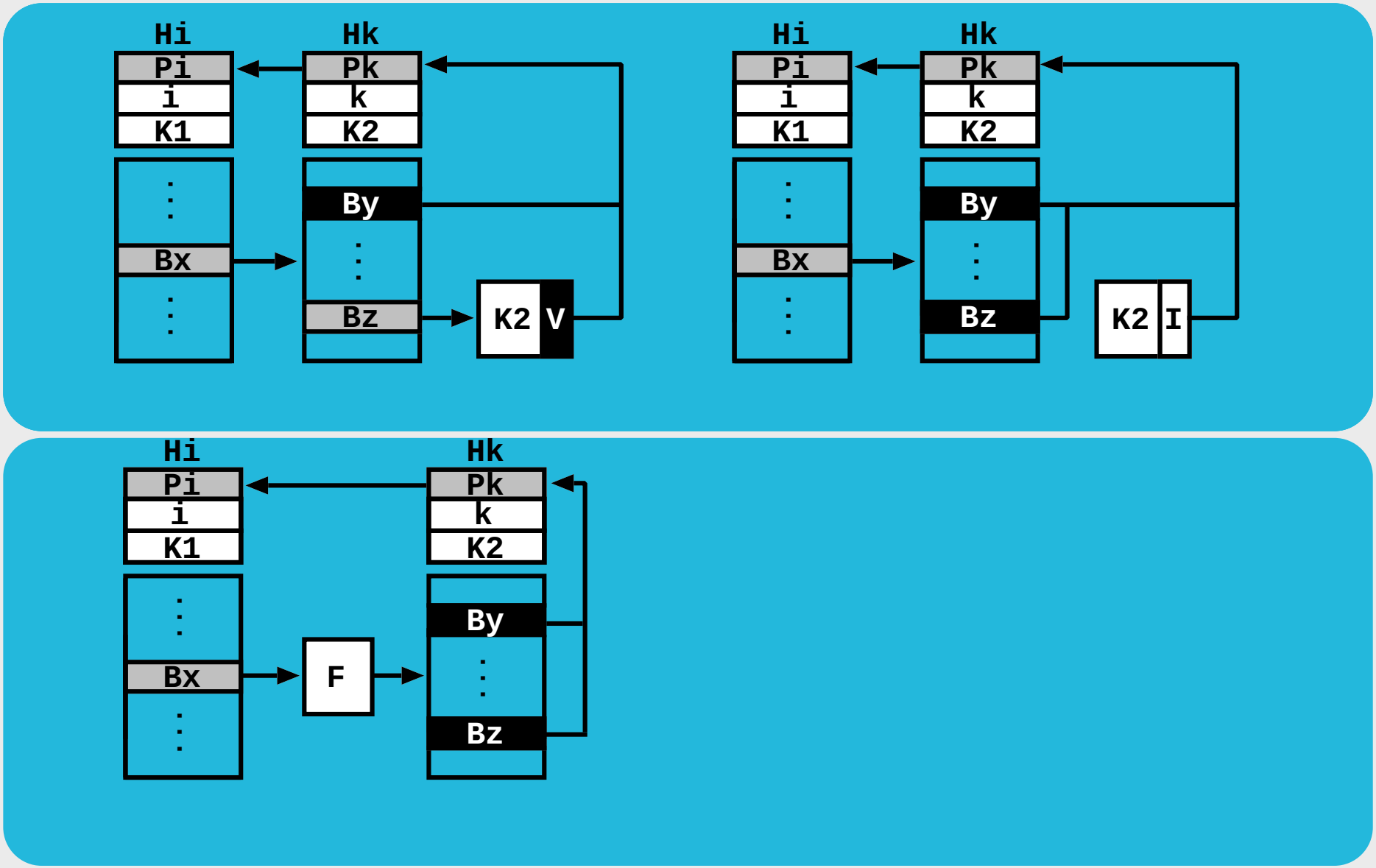
# FP Design - Successful Compression ** NEW **



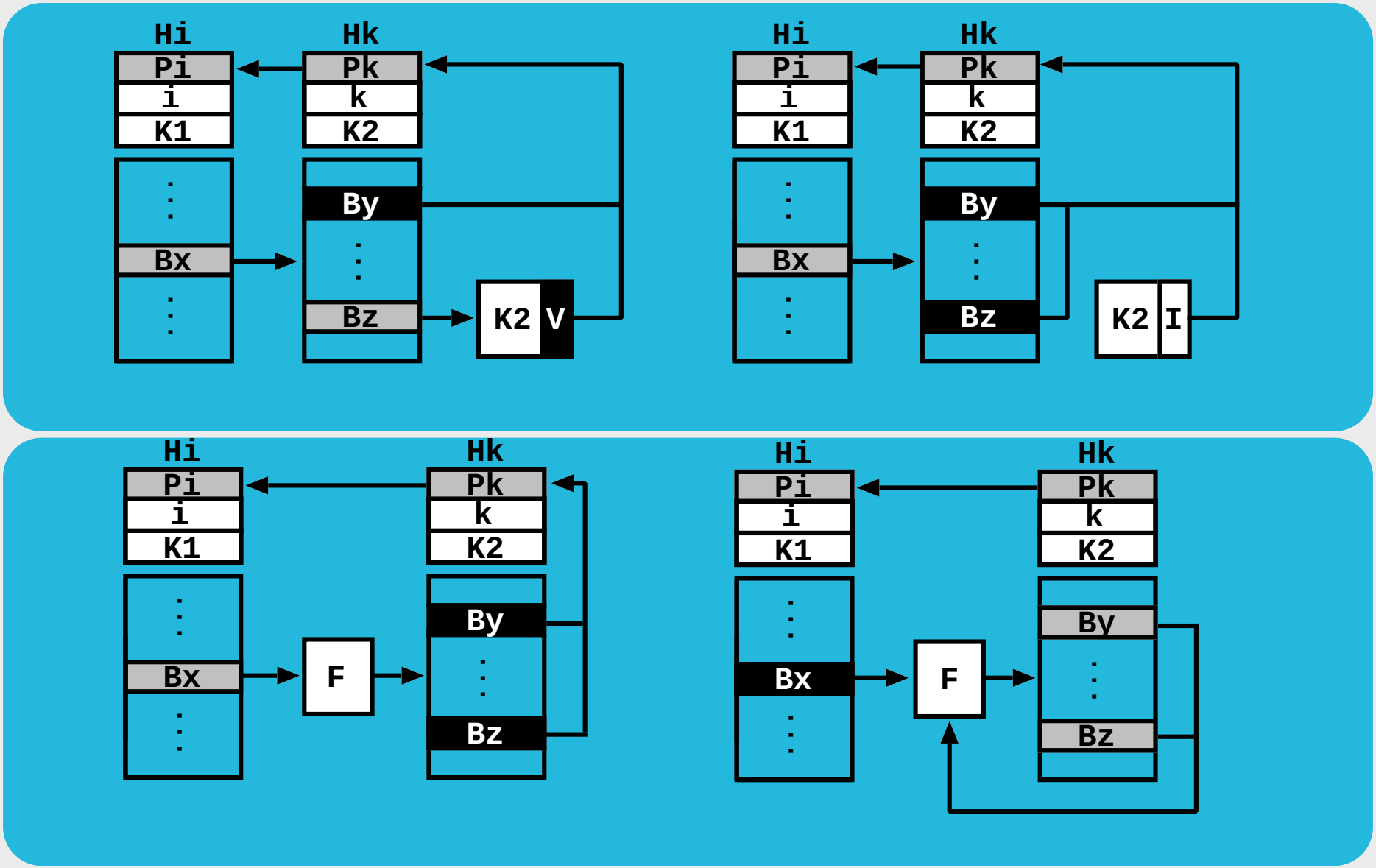
# FP Design - Successful Compression ** NEW **



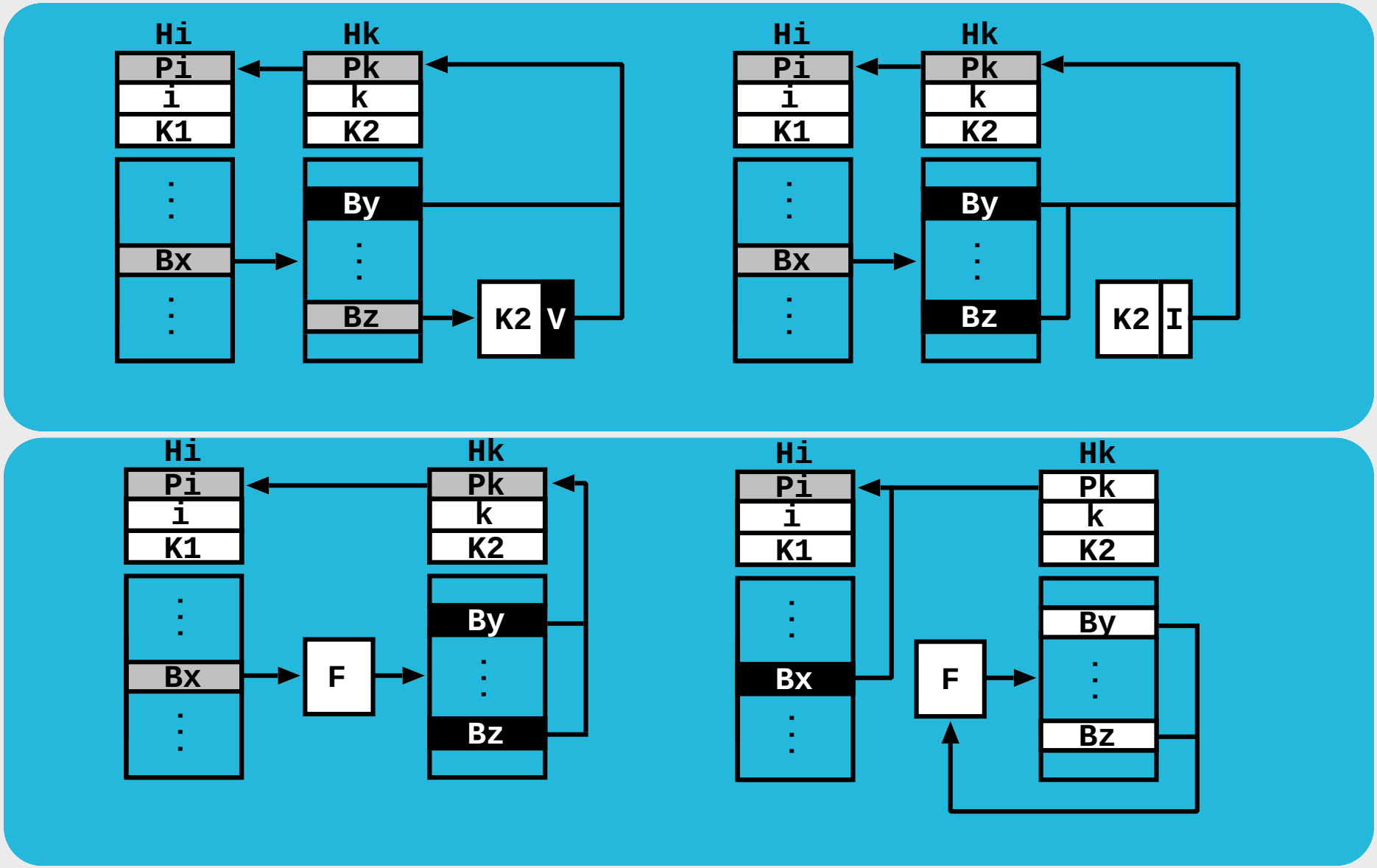
# FP Design - Successful Compression ** NEW **



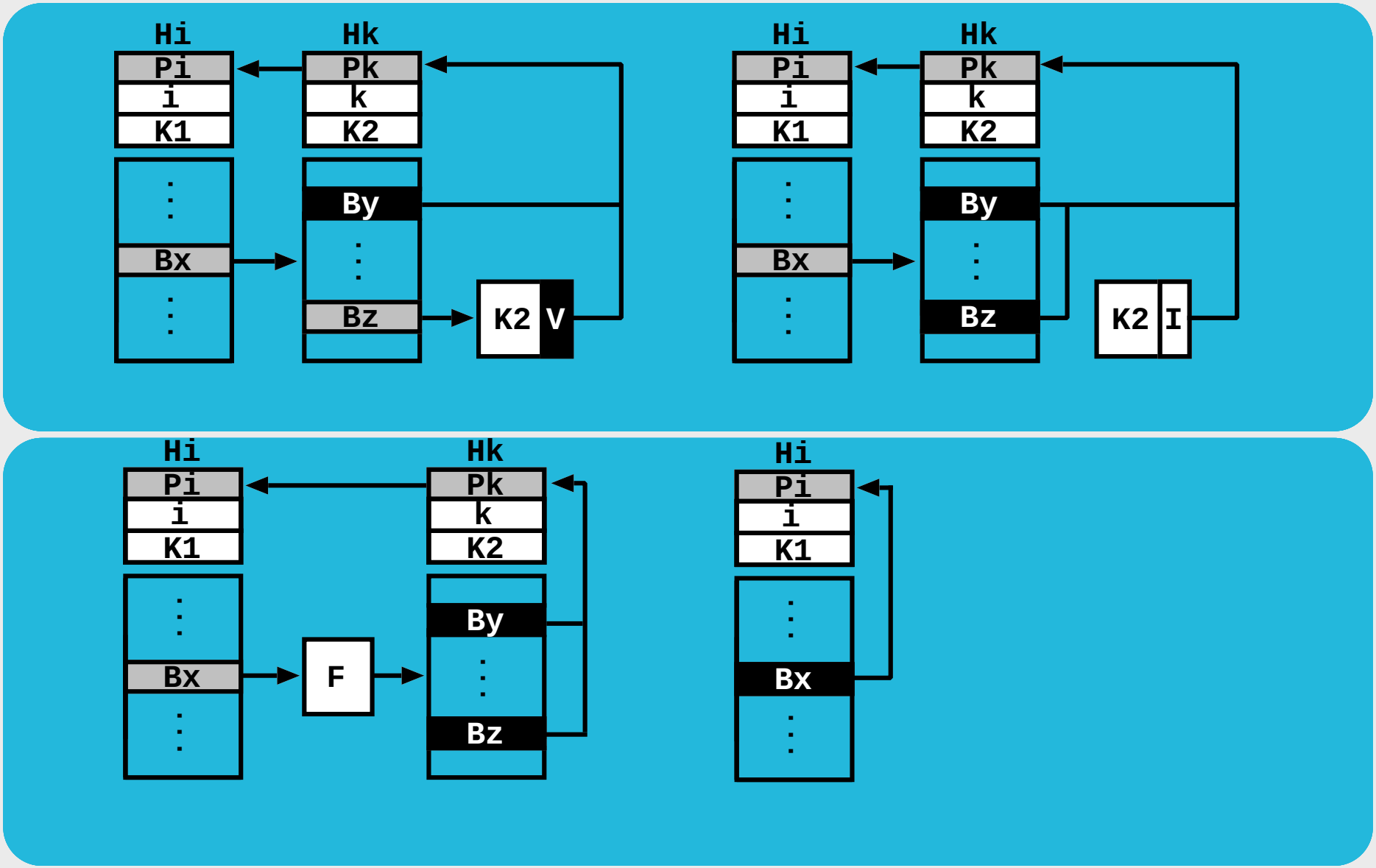
# FP Design - Successful Compression ** NEW **



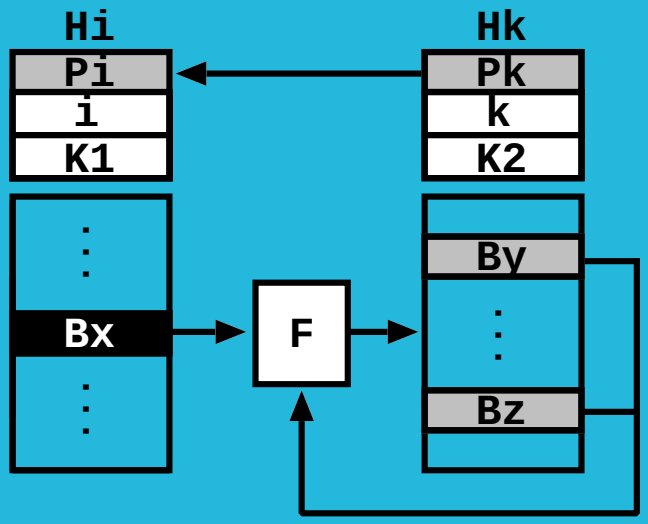
# FP Design - Successful Compression ** NEW **



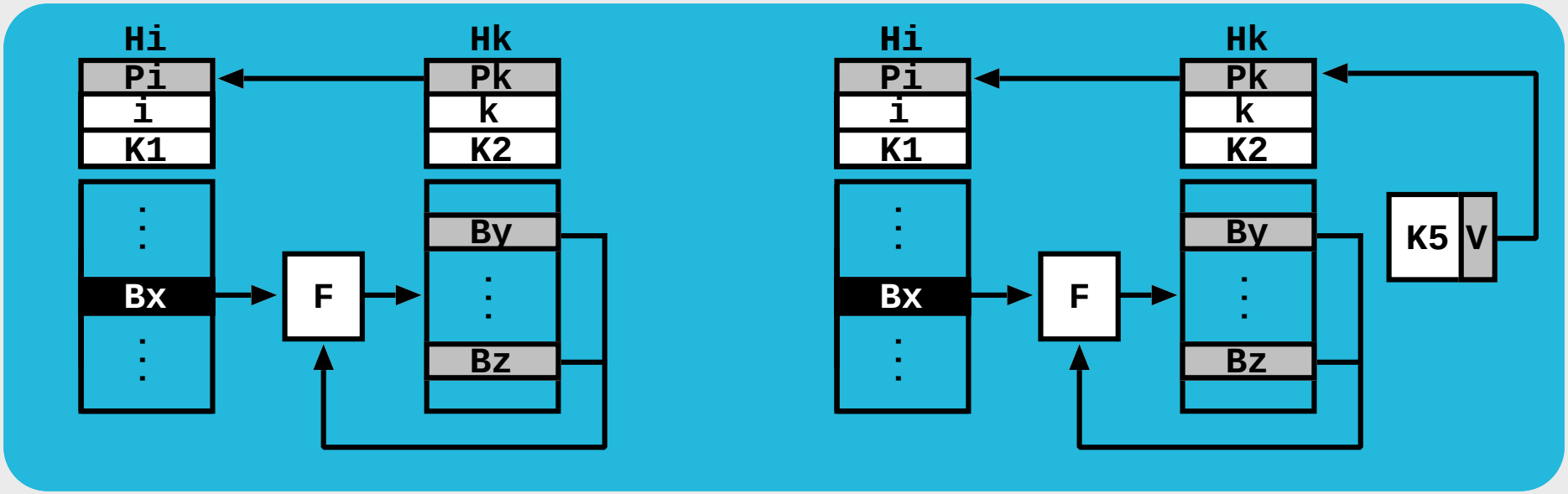
# FP Design - Successful Compression ** NEW **



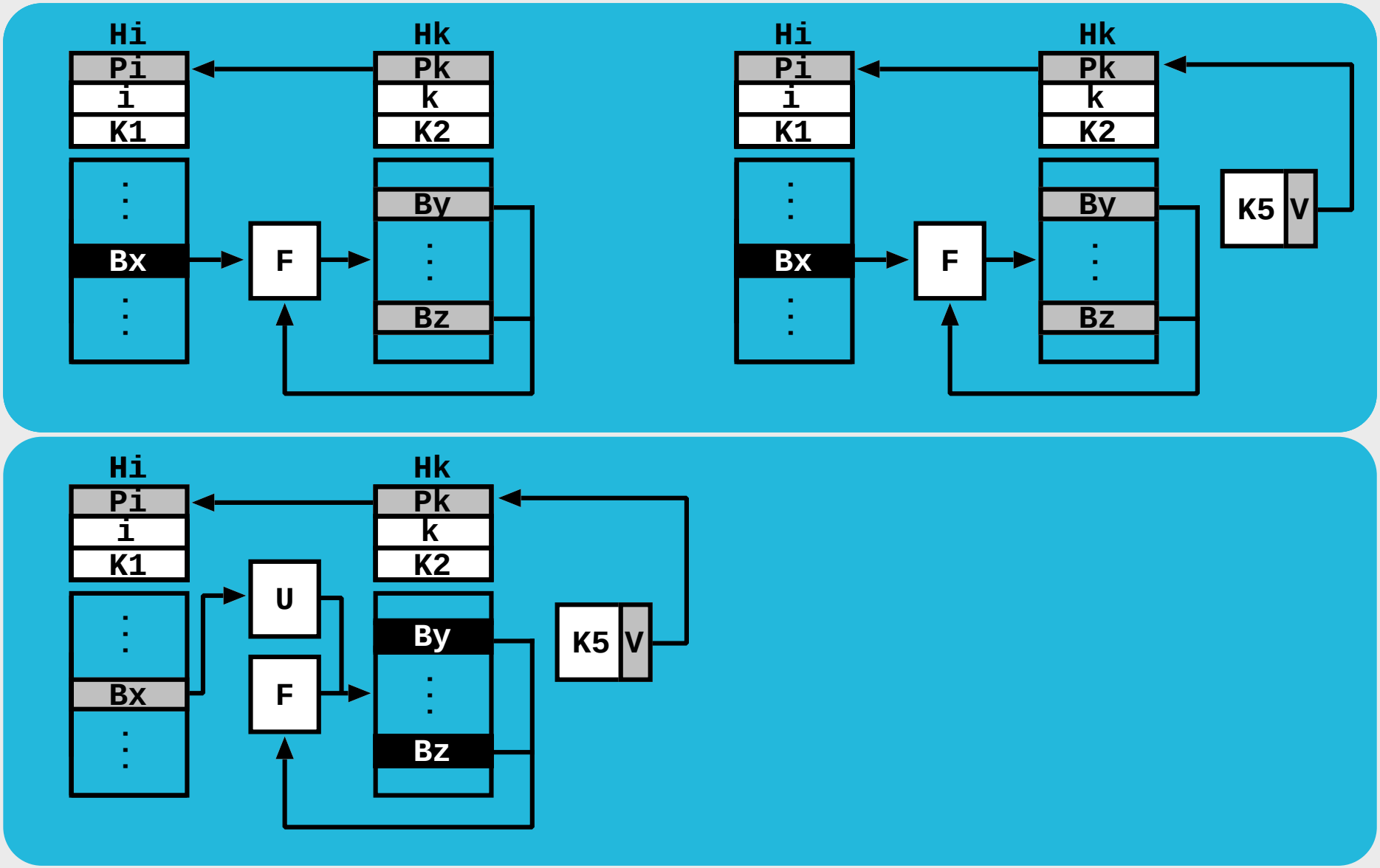
# FP Design - Aborting a Compression ** NEW **



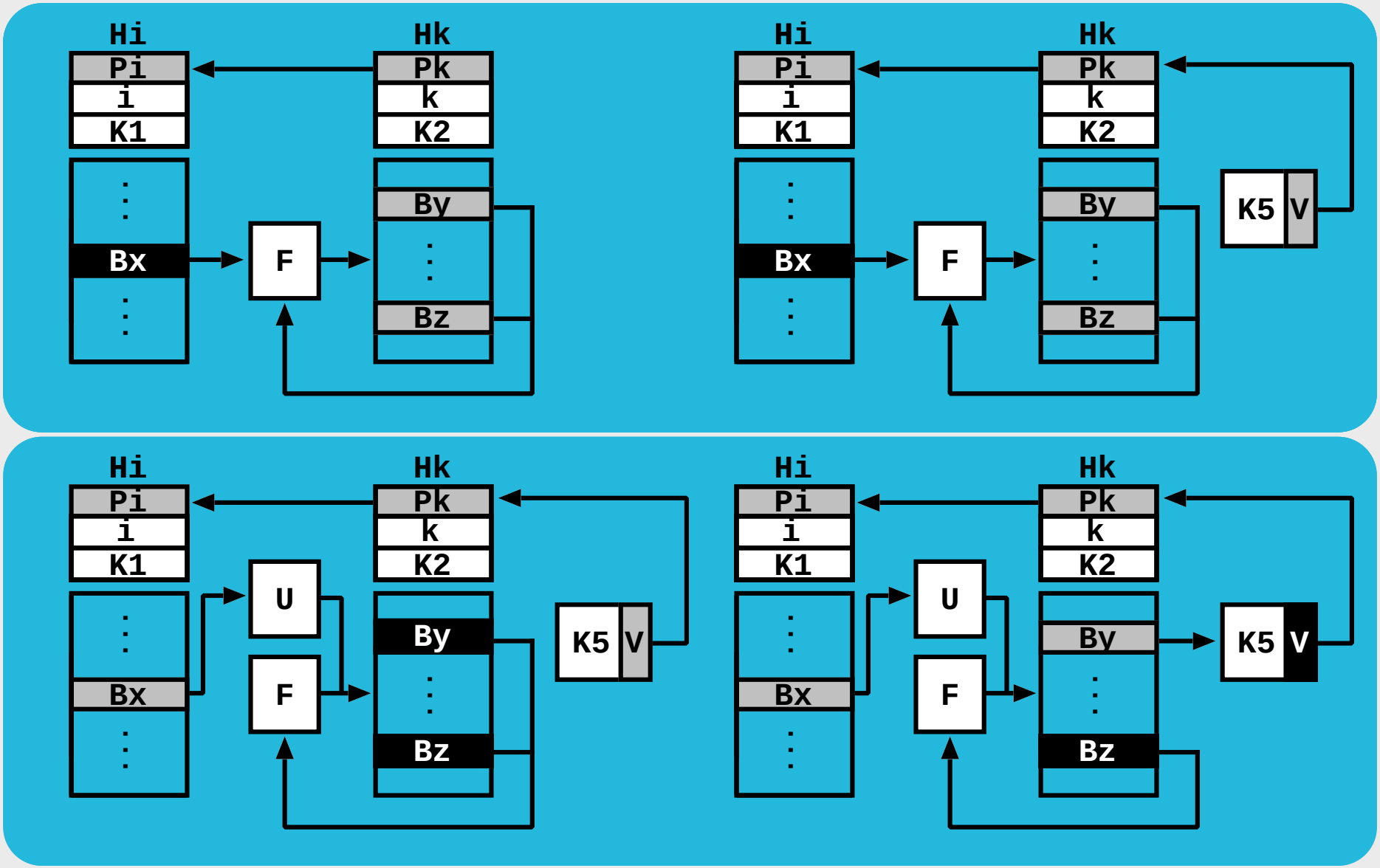
# FP Design - Aborting a Compression ** NEW **



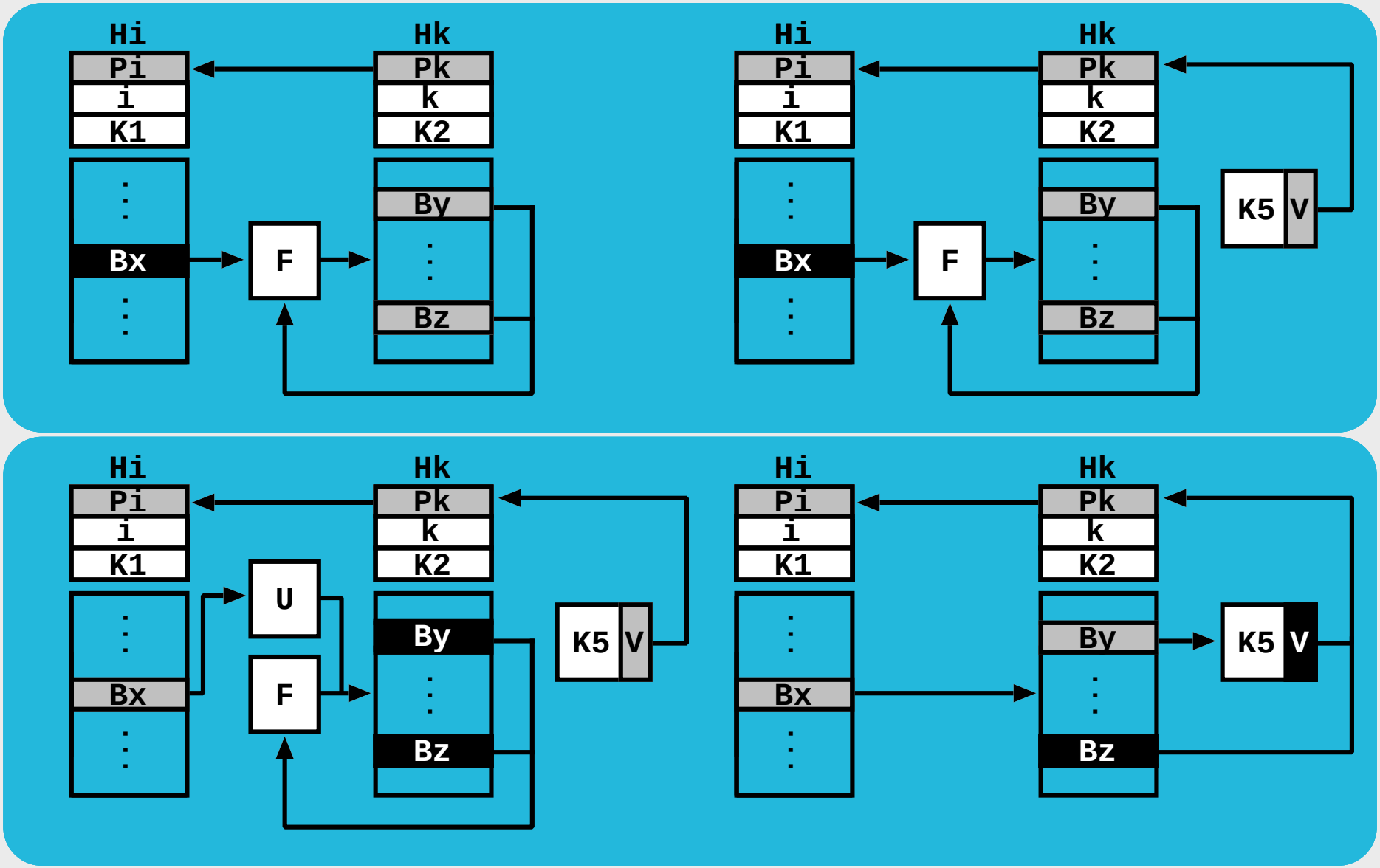
# FP Design - Aborting a Compression ** NEW **



# FP Design - Aborting a Compression ** NEW **



# FP Design - Aborting a Compression ** NEW **



# Performance Analysis

- **Hardware**: $32$ $(2 * 16)$ core AMD with 32 GB of main memory.
- **Software**: Linux Fedora 20 with OpenJDK 13.0.1.
- **Benchmarks**: Sets of $8^8$ (about 17 million) randomized keys with **insert**, **search** and **remove** operations (5 warm up runs and 10 standard runs per benchmark).
- **FP design**: expanded with **2 valid nodes** and each hash bucket had **8 entries**.
- **Podium colors**: **first place**, **second place** and **third place**.

# Performance Analysis


- **Execution time** in milliseconds (**lower** is **better**) and **Speedup Ratio** (**higher** is **better**).

| # Threads ($T_p$) | Execution Time ($E_{T_p}$) | | | | Speedup Ratio ($E_{T_1}/E_{T_p}$) | | | |
|---|---|---|---|---|---|---|---|---|
| | CSL | CT | $FP_{Orig}$ | $FP_{Elastic}$ | CS | CT | $FP_{Orig}$ | $FP_{Elastic}$ |
| **1st – Remove: 0% Search: 100% Insert: 0%** | | | | | | | | |
| 1 | 54,850 | 14,720 | 25,529 | 9,511 | | | | |
| 8 | 7,825 | 2,093 | 3,021 | 1,282 | 7.01 | 7.03 | 8.45 | 7.42 |
| 16 | 4,807 | 1,251 | 1,804 | 859 | 11.41 | 11.77 | 14.15 | 11.07 |
| 24 | 4,773 | 990 | 1,448 | 733 | 11.49 | 14.87 | 17.63 | 12.98 |
| 32 | 4,428 | 904 | 1,570 | 631 | 12.39 | 16.28 | 16.26 | 15.07 |
| **2nd – Remove: 0% Search: 0% Insert: 100%** | | | | | | | | |
| 1 | 100,033 | 36,781 | 48,321 | 31,666 | | | | |
| 8 | 16,089 | 7,119 | 11,048 | 5,537 | 6.22 | 5.17 | 4.37 | 5.72 |
| 16 | 9,903 | 5,341 | 9,983 | 3,871 | 10.10 | 6.89 | 4.84 | 8.18 |
| 24 | 9,191 | 4,980 | 9,083 | 3,691 | 10.88 | 7.39 | 5.32 | 8.58 |
| 32 | 8,636 | 4,838 | 9,177 | 3,923 | 11.58 | 7.60 | 5.27 | 8.07 |

# Performance Analysis


- **Execution time** in milliseconds (**lower** is **better**) and **Speedup Ratio** (**higher** is **better**).

| # Threads ($T_p$) | Execution Time ($E_{T_p}$) CSL | CT | $FP_{Orig}$ | $FP_{Elastic}$ | Speedup Ratio ($E_{T_1}/E_{T_p}$) CS | CT | $FP_{Orig}$ | $FP_{Elastic}$ |
|---|---|---|---|---|---|---|---|---|
| **3rd – Remove: 50%** | **Search: 50%** | | **Insert: 0%** | | | | | |
| 1 | 52,188 | 16,008 | 25,874 | 9,801 | | | | |
| 8 | 8,544 | 2,399 | 3,263 | 1,480 | 6.11 | 6.67 | 7.93 | 6.62 |
| 16 | 5,591 | 1,524 | 2,023 | 1,108 | 9.33 | 10.50 | 12.79 | 8.85 |
| 24 | 5,274 | 1,280 | 1,415 | 945 | 9.90 | 12.51 | 18.29 | 10.37 |
| 32 | 5,188 | 1,344 | 1,768 | 952 | 10.06 | 11.91 | 14.63 | 10.30 |
| **4th – Remove: 33%** | **Search: 33%** | | **Insert: 33%** | | | | | |
| 1 | 77,543 | 23,910 | 35,272 | 24,115 | | | | |
| 8 | 13,811 | 4,163 | 4,785 | 3,776 | 5.61 | 5.74 | 7.37 | 6.39 |
| 16 | 9,093 | 3,038 | 3,131 | 2,518 | 8.53 | 7.87 | 11.27 | 9.58 |
| 24 | 7,974 | 2,681 | 2,918 | 2,484 | 9.72 | 8.92 | 12.09 | 9.71 |
| 32 | 8,444 | 2,552 | 3,038 | 2,428 | 9.18 | 9.37 | 11.61 | 9.93 |

# Performance Analysis


- **Execution time** in milliseconds (**lower** is **better**) and **Speedup Ratio** (**higher** is **better**).

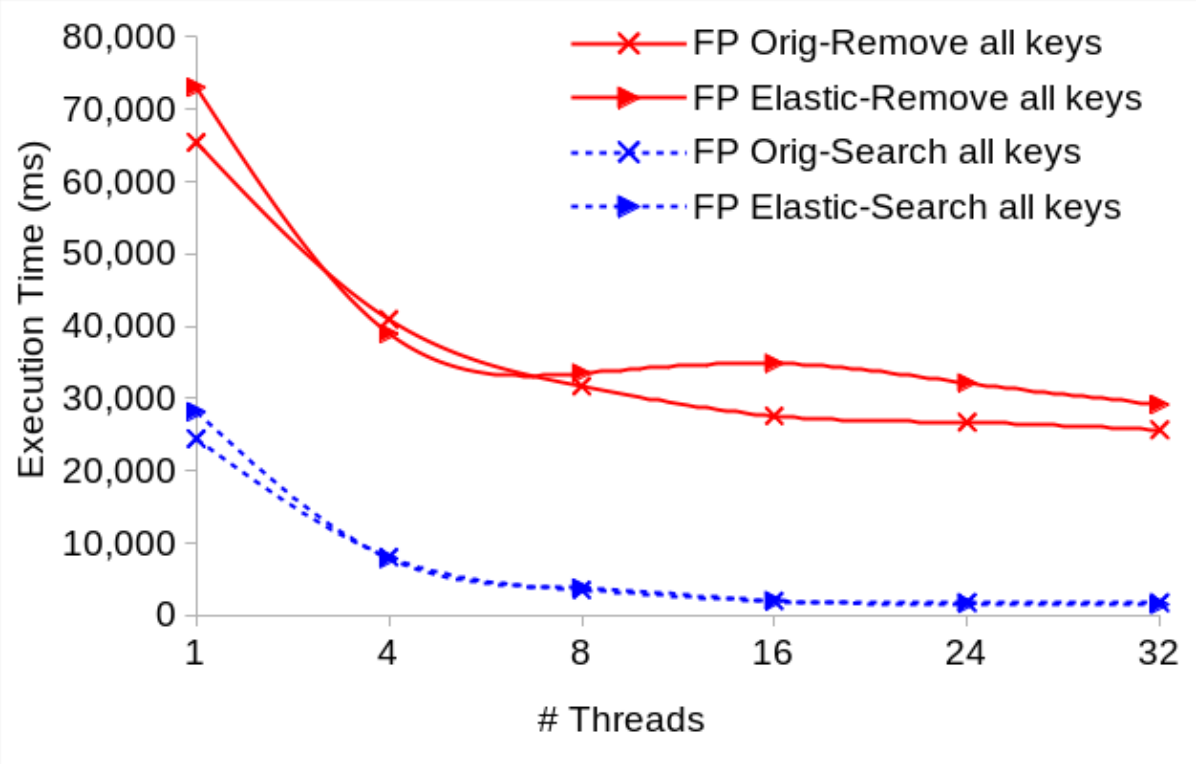
| # Threads ($T_p$) | Execution Time ($E_{T_p}$) | | | | Speedup Ratio ($E_{T_1}/E_{T_p}$) | | | |
|---|---|---|---|---|---|---|---|---|
| | CSL | CT | $FP_{Orig}$ | $FP_{Elastic}$ | CS | CT | $FP_{Orig}$ | $FP_{Elastic}$ |
| **5th – Remove: 40% Search: 40% Insert: 20%** | | | | | | | | |
| 1 | 76,120 | 21,843 | 30,589 | 21,690 | | | | |
| 8 | 12,511 | 3,515 | 3,980 | 3,156 | 6.08 | 6.21 | 7.69 | 6.87 |
| 16 | 7,875 | 2,386 | 2,629 | 1,998 | 9.67 | 9.15 | 11.64 | 10.86 |
| 24 | 7,906 | 2,209 | 2,452 | 1,779 | 9.63 | 9.89 | 12.48 | 12.19 |
| 32 | 7,027 | 2,200 | 2,333 | 1,791 | 10.83 | 9.93 | 13.11 | 12.11 |
| **6th – Remove: 20% Search: 40% Insert: 40%** | | | | | | | | |
| 1 | 82,145 | 25,061 | 34,771 | 26,087 | | | | |
| 8 | 13,898 | 4,373 | 4,865 | 3,915 | 5.91 | 5.73 | 7.15 | 6.66 |
| 16 | 8,659 | 3,047 | 3,441 | 3,043 | 9.49 | 8.22 | 10.10 | 8.57 |
| 24 | 8,514 | 2,877 | 3,144 | 2,694 | 9.65 | 8.71 | 11.06 | 9.68 |
| 32 | 6,854 | 2,773 | 3,096 | 2,385 | 11.98 | 9.04 | 11.23 | 10.94 |

# $FP_{Orig}$ vs $FP_{Elastic}$

- Within the setup stage, we **inserted all keys** in the set $\mathbf{I} = \{0, ..., 8^8 - 1\}$
  - (since we used **8 bucket entries** per hash level, all chain nodes were located in a hash level with **depth 8**).

# $FP_{Orig}$ vs $FP_{Elastic}$

- Within the setup stage, we **inserted all keys** in the set $\mathbf{I} = \{0, ..., 8^8 - 1\}$
  - (since we used **8 bucket entries** per hash level, all chain nodes were located in a hash level with **depth 8**).
- And then, we measured the **execution time** that both designs took to: **remove all keys** and **search for all keys**.

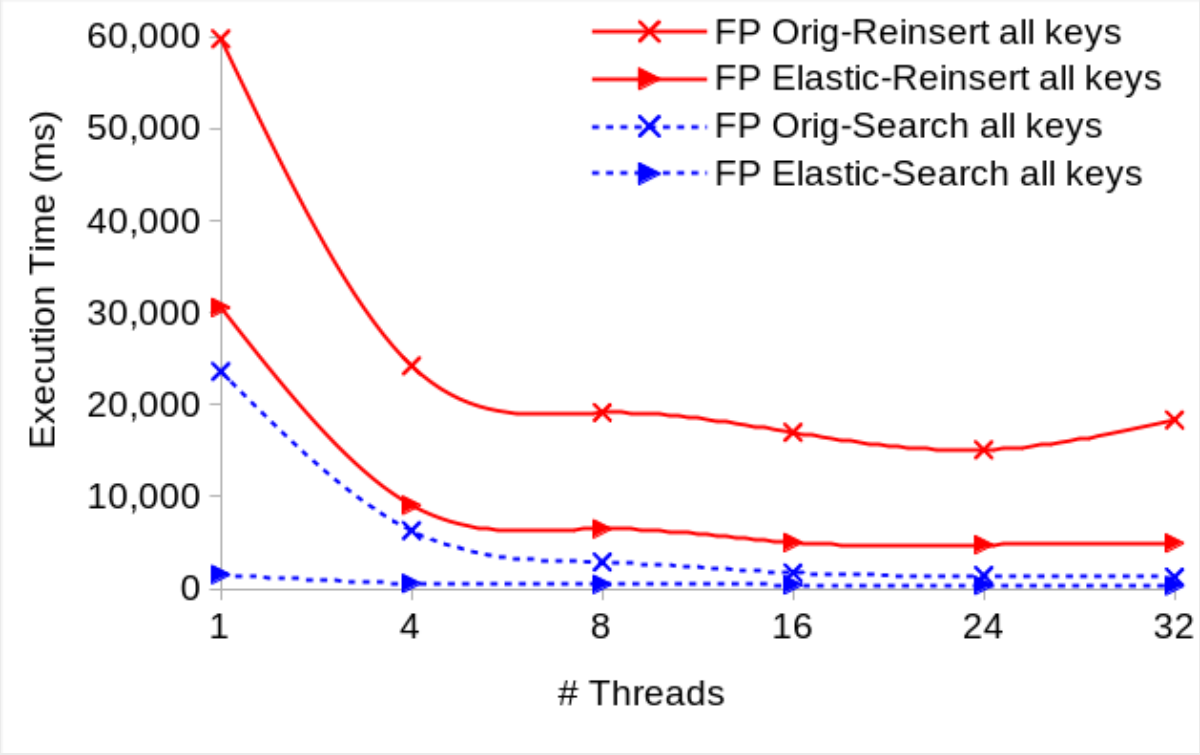


# $\mathbf{FP_{Orig}}$ vs $\mathbf{FP_{Elastic}}$

- Within the setup stage, we **inserted and then removed all keys** in the set $\mathbf{I}$
  - (recall that $\mathbf{FP_{Orig}}$ **removes** the **keys**, but **keeps** its hash hierarchy unchanged. $\mathbf{FP_{Elastic}}$ **removes keys** and **hashes** (except the root hash)).

# $\mathbf{FP_{Orig}}$ vs $\mathbf{FP_{Elastic}}$

- Within the setup stage, we **inserted and then removed all keys** in the set $\mathbf{I}$
  - (recall that $\mathbf{FP_{Orig}}$ **removes** the **keys**, but **keeps** its hash hierarchy unchanged. $\mathbf{FP_{Elastic}}$ **removes keys** and **hashes** (except the root hash)).
- And then, we measured the **execution time** that both designs took to: **reinsert all keys** and **search for all keys**.



# Conclusions and Further Work

- We have presented a novel, scalable and **elastic hash trie design** that fully supports the concurrent search, insert, remove, expand and compress operations.

# Conclusions and Further Work

- We have presented a novel, scalable and **elastic hash trie design** that fully supports the concurrent search, insert, remove, expand and compress operations.
- **Experimental results** show that **elasticity** overheads are largely overcome by its **benefits**.

# Conclusions and Further Work

- We have presented a novel, scalable and **elastic hash trie design** that fully supports the concurrent search, insert, remove, expand and compress operations.
- **Experimental results** show that **elasticity** overheads are largely overcome by its **benefits**.
  - **Elasticity** effectively **improves** the **search** operation, and, by doing so, the design became **very competitive**, when compared against:
    - Other **state-of-the-art** designs implemented in Java.

# Conclusions and Further Work

- We have presented a novel, scalable and **elastic hash trie design** that fully supports the concurrent search, insert, remove, expand and compress operations.
- **Experimental results** show that **elasticity** overheads are largely overcome by its **benefits**.
  - **Elasticity** effectively **improves** the **search** operation, and, by doing so, the design became **very competitive**, when compared against:
    - Other **state-of-the-art** designs implemented in Java.
    - The **non-elastic** version of the design.

# Conclusions and Further Work

- We have presented a novel, scalable and **elastic hash trie design** that fully supports the concurrent search, insert, remove, expand and compress operations.
- **Experimental results** show that **elasticity** overheads are largely overcome by its **benefits**.
  - **Elasticity** effectively **improves** the **search** operation, and, by doing so, the design became **very competitive**, when compared against:
    - Other **state-of-the-art** designs implemented in Java.
    - The **non-elastic** version of the design.
- As further work, we plan to use our design as the building block for a novel distributed hash map design.

# Thank You !!!

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

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


CRACS INESCTEC ASSOCIATE LABORATORY PORTUGAL

FCT Fundação para a Ciência e a Tecnologia MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

U.PORTO