Data Structure and Algorithm Design

ME/MSc (Computer) – Pokhara University

Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np

Chapter 4:
Advance Hashing Techniques (6 hrs)

Outline

4. Advanced Hashing Techniques (6 hrs)

- 4.1. Hardware-Accelerated Hashing
- 4.2. Perfect Hashing
- 4.3. Cuckoo Hashing
- 4.4. Hashing in Distributed Systems
- 4.5. Case Studies in GPUs and FPGAs

Hashing (Review)

- In many applications, we need to quickly find, insert, or delete data.
- Traditional data structures like arrays or linked lists can be inefficient for these operations, especially as the size of the data grows.
- Hashing addresses this by providing a way to map data to a specific
- Hashing involves using a **hash function** to convert data (like a string or a number) into a fixed-size integer, which serves as an index in an array called a **hash table**.

1. **Hash Function**: This function takes an input (or 'key') and returns an integer known as the hash value. A good hash function distributes keys uniformly across the hash table to minimize collisions.
2. **Hash Table**: This is an array where the data is stored. Each position in the array is called a bucket, and the hash value determines which bucket a key-value pair is stored in.
3. **Handling Collisions**: Sometimes, different keys may hash to the same index. This is known as a collision.

Hashing (Review)

Hash Functions:

1. Division Method

- This method uses the modulo operation. The hash value is calculated as: $h(key) = key \% m$, where m is the size of the hash table.
- This method is simple and effective if m is chosen wisely (usually a prime number).

2. Multiplication Method

- This method involves multiplying the key by a constant A $(0 < A < 1)$ and taking the fractional part.
- Hash function: $h(key) = \lfloor m \cdot (key \cdot A \bmod 1) \rfloor$

Hashing (Review)

Hash Functions:

3. Mid-Square Method

- The *key* is squared, and the middle part of the result is taken as the hash value. This method can provide a good distribution of hash values.
- Example: For $k=123$, $k^2=15129$; middle digits (*512*) form the hash.

4. Folding Method

- The *key* is divided into equal parts, and these parts are added together to get the hash value. If the parts are not of equal length, the last part can be shorter.
- Example: For $k=123456$, split into *123* and *456*, add them ($123 + 456 = 579$), and use $579 \% m$ as the hash value.
- Use: Handles large numeric keys.

Hashing (Review)

Hash Functions:

5. Universal Hashing

- It is a probabilistic method where a hash function is chosen randomly from a family of hash functions.
- It provides good average-case performance and reduces the likelihood of collisions.
- For universal hashing, the probability of two distinct keys *k1* and *k2* colliding under a randomly chosen hash function *h* is: $Pr[h(k1)=h(k2)] \leq 1/m$, where:
  - $k1 \neq k2$,
  - *m*: The size of the hash table.
- This guarantees that:
  - No pair of keys is disproportionately likely to collide.
  - The hashing is evenly distributed across buckets.
- Universal hashing is particularly useful in scenarios where input patterns are not predictable, ensuring robustness against adversarial or clustered inputs.

Hashing (Review)

- A **collision** occurs in hashing when two or more keys map to the same index in the hash table.
- In mathematical terms, if $h(k1)=h(k2)$ and $k1 \neq k2$, it is a collision.
- Collisions are inevitable in hashing because the number of possible keys is typically much larger than the size of the hash table.
- Example:

  Hash function: $h(k)=k \% 10$

  Keys: 23 and 33

  $h(23)=23 \% 10 = 3$

  $h(33)=33 \% 10 = 3$

  Both keys map to the same index, causing a collision.

Hashing (Review)

Collision Handling Techniques

1. Chaining (Separate Chaining)

- Each slot in the hash table contains a linked list or a dynamic data structure to store multiple keys mapping to the same index.
- if a collision occurs, append the new key to the linked list at the hashed index.
- When searching, traverse the linked list to find the key.
- **Advantages**: Simple and efficient when the load factor is low.
- **Disadvantages**: Performance degrades if many keys map to the same index.
- **Example**:
  - Hash table: Size 10 , Keys: 12, 22, 32
  - Hash function: $h(k) = k \bmod 10$
  - Result: At index 2, the linked list contains [12, 22, 32].

Hashing (Review)

2. Open Addressing

- Store all keys directly in the hash table and probe for the next available slot if a collision occurs.
- **Types of probing:**

1. Linear Probing:
- Search sequentially for the next empty slot
- Hash function: $h(k,i)=(h(k)+i) \% m$, where i is the probe number.
- Example: Keys: 12, 22
- Hash function: $h(k)=k \% 10$
- 12 goes to index 2, 22 also maps to 2 → place 22 in the next slot (index 3).

2. Quadratic Probing:
- Search for empty slots using a quadratic formula.
- Hash function: $h(k,i)=(h(k)+i^2) \% m$.

3. Double Hashing:
- Use a second hash function for probing.
- Hash function: $h(k,i)=(h_1(k)+i \cdot h_2(k))\% m$

- **Advantages**: Avoids the overhead of linked lists.
- **Disadvantages**: May lead to clustering (keys clustering around certain indices).

Hashing (Review)

Collision Handling Techniques

3. Rehashing

- When the hash table becomes too full, increase the table size and recompute the hash values for all keys.
- **Advantages**: Reduces collisions by increasing the number of available slots.
- **Disadvantages**: Expensive operation as all keys need to be rehashed.

4. Universal Hashing

- **Method**: Use a randomly chosen hash function from a family of hash functions to reduce collision likelihood.
- **Advantage**: Ensures uniform distribution of keys.

Universal Hashing (Example)

Define a Family of Hash Functions

- A common universal hash function family is defined as:
  - $H_{a,b}(x)=((a \cdot x+b) \bmod p) \bmod m$

  where:
  - p is a prime number greater than the maximum key value.
  - a and b are randomly chosen integers such that $1 \leq a < p$ and $0 \leq b < p$.
  - m is the number of hash table slots.
- **Choose Parameters**
  - Let's take:
  - $p=7$ (a prime greater than the max key)
  - $m=5$ (hash table size)
  - Randomly choose $a=3$ and $b=2$
- Thus, our hash function becomes:
  - $h(x)=((3x+2) \bmod 7) \bmod 5$

Universal Hashing (Example)

Compute Hash Values

- For different keys, say **4, 5, and 6**:

1. **For x = 4**
   - $h(4)=((3\times4+2)\%\ 7)\ \%\ 5=((12+2)\ \%\ 7)\ \%\ 5=(14\ \%\ 7)\%\ 5=0\%\ 5=0$
2. **For x = 5**
   - $h(5)=((3\times5+2)\%\ 7)\%\ 5=((15+2)\%\ 7)\%\ 5=(17\%\ 7)\%\ 5=3$
3. **For x = 6**
   - $h(6)=((3\times6+2)\%\ 7)\%\ 5=((18+2)\%\ 7)\%\ 5=(20\%\ 7)\%\ 5=6\%\ 5=1$

- Keys **4, 5, and 6** were mapped to different hash buckets: **0, 3, and 1**.
- By choosing a function randomly from a universal family, we reduce the probability of collisions.
- **Universal hashing + chaining** is commonly used in cryptographic applications and secure hash tables.

Hardware Accelerated Hashing

Hashing

Hardware Accelerated Hashing

- Hardware accelerated hashing refers to the use of specialized hardware to perform hash computations more efficiently than traditional software-based methods.
- Instead of relying solely on the CPU to perform the complex mathematical operations involved in hashing, hardware acceleration offloads these tasks to dedicated units designed specifically for this purpose.

Benefits of Hardware Accelerated Hashing:

- **Increased Speed**: Hardware accelerators can process data much faster than software implementations, resulting in significantly faster hash generation.
- **Reduced CPU Load**: By offloading hashing operations to hardware, the CPU is freed up to perform other tasks, improving overall system performance.
- **Improved Efficiency**: Hardware accelerators are often designed for optimal performance and power efficiency for hashing operations.
- **Enhanced Security**: Hardware-based implementations can provide additional security measures, such as protecting cryptographic keys and preventing tampering.

Hashing

Examples of Hardware Accelerated Hashing

1. **AES-NI and SSE2**: Modern processors, such as those from Intel and AMD, include instructions like AES-NI (Advanced Encryption Standard New Instructions) and SSE2 (Streaming SIMD Extensions 2) that accelerate cryptographic operations, including hashing.
2. **GPUs**: Graphics Processing Units (GPUs) are used for massively parallel hash computations, leveraging their high memory bandwidth and computational power to accelerate data structures like hash maps.
3. **FPGA and ASIC**: Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) can be designed to perform specific hash functions at very high speeds, often used in specialized applications like blockchain mining.
4. **CRC (Cyclic Redundancy Check) Instructions**: Many modern processors have CRC-specific instructions (e.g., Intel's **CRC32** instruction) for fast checksum calculations, often used in network applications.

Hashing

Applications of Hardware Accelerated Hashing

1. **Cryptographic Hashing**: Enhancing the performance of cryptographic algorithms like SHA-256 and SHA-3.
2. **Data Storage and Retrieval**: Improving the efficiency of databases and caching systems.
3. **Blockchain**: SHA-256 or other hashing algorithms are hardware-accelerated for efficient transaction verification and mining.
4. **Networking**: Hardware hash functions are used for packet classification, load balancing, and flow hashing in routers and switches.
5. **Real-Time Applications**:Time-sensitive systems such as financial trading platforms benefit from low-latency hashing.

Hashing

Challenges of Hardware Accelerated Hashing

1. **Cost**: Specialized hardware (e.g., ASICs, FPGAs) can be expensive.
2. **Flexibility**: Custom hardware (e.g., ASICs-**Application-Specific Integrated Circuit**) is not easily upgradable for new hashing algorithms.
3. **Development Complexity**: Designing and integrating hardware-accelerated solutions can be complex.
4. **Limited Generality**: Hardware-accelerated hashing is optimized for specific algorithms, limiting its scope.

Hashing

Summary from the Paper:

A. A. Fairouz, M. Abusultan, V. V. Fedorov and S. P. Khatri, "Hardware Acceleration of Hash Operations in Modern Microprocessors," in *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1412-1426, 1 Sept. 2021, doi: 10.1109/TC.2020.3010855.

- Modern applications like cloud computing and networking demand high-speed data processing. Traditional software-based hashing can be a bottleneck.
- **Proposed Solution**: The authors propose a hardware hash unit (HU) to accelerate hash operations, potentially increasing performance significantly.
- By introducing a hardware-based approach, the authors achieve significant performance and energy efficiency improvements.

Hashing

Key Contributions:

1. **Microarchitecture Design**:
    1. The Hardware Hash Unit(HU) integrates into the processor pipeline and consists of:
        1. A **hash function (HF)** for fast key-to-bin mapping using bitwise XOR and AND operations.
        2. A **hash table (HT)** implemented using **content-addressable memory (CAM)** for efficient lookup, insert, and delete operations.
    2. Simulations using GEM5 demonstrate up to **15× speedup** over software-based hashing.
2. **Circuit-Level Implementation**:
    1. The HU is implemented using a 45nm CMOS process and achieves:
        1. **5.48× reduction in power consumption** compared to traditional CAM-based designs.
        2. A maximum operating frequency of **1.39 GHz**, ensuring fast and reliable operations.
3. **Scalability**:
    1. The HU handles larger hash tables and parallel hash operations efficiently.
    2. It supports multiple processes with context switching, ensuring correctness and reducing cache misses.

Hashing

Experimental Results

- **Simulation Environment**: The HU was simulated using GEM5 and Synopsys tools, verifying its correctness and performance.
- **Benchmarks**: The HU was tested with various benchmarks, including Yahoo! Cloud Serving Benchmark (YCSB) and PARSEC benchmarks, showing significant improvements in speed and power efficiency.

Conclusion

- **Impact**: The proposed HU design offers a substantial performance boost for hashing operations in modern microprocessors, making it suitable for a wide range of applications.

- ***Note: Provide above mention article.***

Perfect Hashing

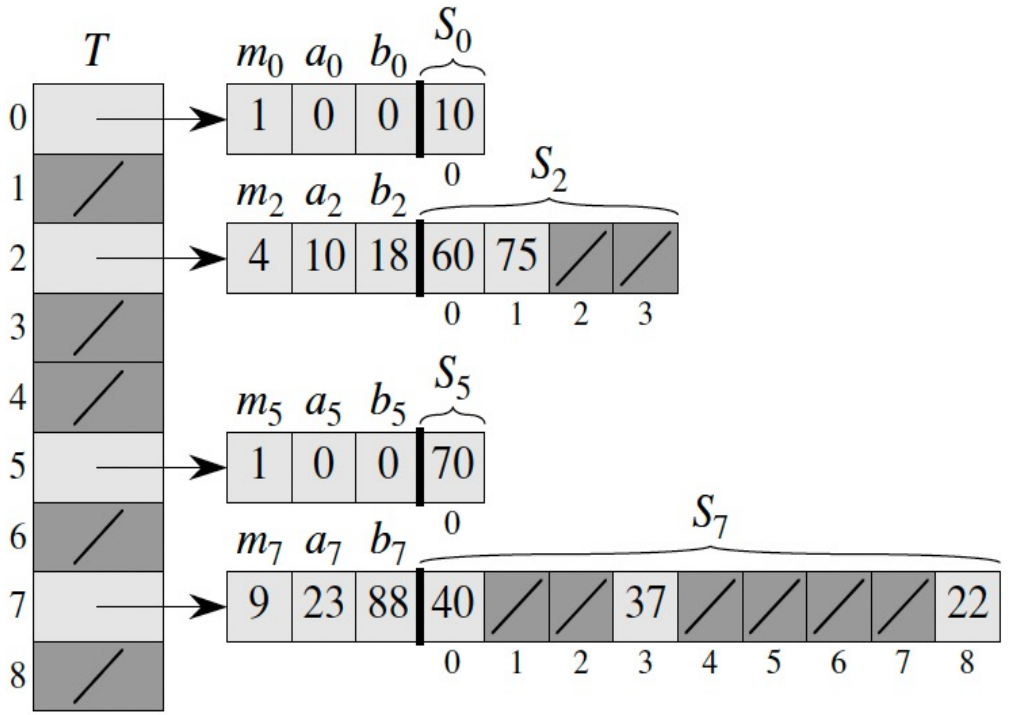
Perfect Hashing

- Hashing is most often used for its excellent expected performance
- It can be used to obtain excellent ***worst-case performance*** when the set of keys is ***static***: once the keys are stored in the table, the set of keys never changes.
- Some applications naturally have static sets of keys: *set of reserved words in a programming language*, or *the set of file names on a CD-ROM*.
- We call a hashing technique ***perfect hashing*** if the worst-case number of memory accesses required to perform a search is **O(1)**.
- Basic idea to create a perfect hashing:
  - Use a two-level hashing scheme with universal hashing at each level ***(Fig: Next Slide).***
  - The first level is essentially the same as for hashing with chaining: the n keys are hashed into m slots using a hash function h carefully selected from a family of universal hash functions.
  - Instead of making a list of the keys hashing to slot j, use a small secondary hash table S_j with an associated hash function h_j. By choosing the hash functions h_j carefully, we can guarantee that there are no collisions at the secondary level.

Perfect Hashing

Basic idea…

- In order to guarantee that there are no collisions at the secondary level, we need to let the size m_j of hash table S_j be the *square of the number* n_j of keys hashing to slot j.
- While having such a quadratic dependence of m_j on n_j may seem likely to cause the overall storage requirements to be excessive. But, it has to be show that by choosing the first level hash function well, the expected total amount of space used is still **O(n)**.



Using perfect hashing to store the set K = {10, 22, 37, 40, 60, 70, 75}.
The outer hash function is $h(k) = ((ak + b) \% p) \% m$, where a = 3, b = 42, p = 101, and m = 9.
For example, h(75) = 2, so key 75 hashes to slot 2 of table T.
A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is $m_j = n_j^2$, and the associated hash function is
$h_j(k) = ((a_j k + b_j) \% p) \% m_j$.
Since h2(75) = 1, key 75 is stored in slot 1 of secondary hash table S2. There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.

Perfect Hashing

In Short:

- Perfect hashing is a **two-level hashing scheme** that ensures **zero collisions** in the second level. The process involves:

1. **First-Level Hashing** [*Outer Hash Function* $h(k)$]
   - Distributes keys into buckets of a primary hash table T.
   - Keys that map to the same slot form a *secondary hash table* S_j.
2. **Second-Level Hashing** [*Inner Hash Function* $h_j(k)$]
   - Each secondary table S_j has a different hash function $h_j(k)$.
   - The goal is to store keys without collisions in S_j.

Calculation of Given Example

- Given Set $K=\{10,22,37,40,60,70,75\}$
- Outer hash function : $h(k) = ((ak + b) \% p) \% m$, where $a = 3, b = 42, p = 101$, and $m = 9$.
  - i.e. $h(k)=((3k+42)\%\ 101)\%\ 9$

Perfect Hashing Example

Step1: Computer the first level hashing using outer hash function [$h(k)=((3k+42)\%\ 101)\%\ 9$]

Key (k)	Compute h(k)	First-Level Slot (T[h(k)])
10	((3×10+42)% 101)% 9=(72% 101)% 9=72% 9=0	0
22	((3×22+42)% 101)% 9=(108% 101)% 9=7% 9=7	7
37	((3×37+42)% 101)% 9=(153% 101)% 9=52% 9=7	7
40	((3×40+42)% 101)% 9=(162% 101)% 9=61% 9=7	7
60	((3×60+42)% 101)% 9=(222% 101)% 9=20% 9=2	2
70	((3×70+42)% 101)% 9=(252% 101)% 9=50% 9=5	5
75	((3×75+42)% 101)% 9=(267% 101)% 9=65% 9=2	2

Perfect Hashing Example

Primary Hash Table T after First-Level Hashing

Index	Keys Stored (Bucket S_j)
0	{10}
1	-
2	{60, 75}
3	-
4	-
5	{70}
6	-
7	{22, 37, 40}
8	-

Perfect Hashing Example

Step 2: Compute Second-Level Hashing $h_j(k)$

Each secondary table S_j requires its own hash function:

$h_j(k)=((a_jk+b_j)\%\ p)\%\ m_j$ where a_j, b_j are randomly chosen and m_j is the size of S_j and m_j is usually set to n_j^2 *(square of bucket size)* to eliminate collisions.

Secondary Hash Tables S_j Construction

1.Bucket S_0 (*Keys:* $\{10\}$, $m_0=1^2=1$, randomly chosen $a_0=0$ *and* $b_0=0$)

1. Use $h_0(k)=(0k+0)\%\ 101\ \%\ 1 = h_0(10)=(0\times10+0)\%\ 101\ \%\ 1 = (0\%101)\%1 = 0\%1=0$
2. Since only one key exists, no collision.

2.Bucket S_2 (*Keys:* $\{60, 75\}$, $m_2=2^2=4$, randomly chosen $a_2=10$ *and* $b_2=18$)

1. $h_2(k)=((10k+18)\%\ 101)\%\ 4$
2. Compute for each key:
   - $h_2(60)=((10\times60+18)\%\ 101)\%\ 4=(618\%\ 101)\%\ 4=12\%4=0$
   - $h_2(75)=((10\times75+18)\%\ 101)\%\ 4=(768\%\ 101)\%\ 4=61\%4=1$

Perfect Hashing Example

Secondary Hash Tables S_j Construction…

3. Bucket S_5 (*Keys:* $\{70\}$, $m_5=1^2=1$, randomly chosen $a_5=0$ *and* $b_5=0$)

1. Use $h_5(k)=(0k+0)\%\ 101\ \%\ 1 = h_5(70)=(0\times70+0)\%\ 101\ \%\ 1 = (0\%101)\%1 = 0\%1=0$
2. Since only one key exists, no collision.

4. Bucket S_7 (*Keys:* $\{22,37,40\}$, $m_7=3^2=9$, randomly chosen $a_7=23$ *and* $b_7=88$)

1. $h_7(k)=((23k+88)\%\ 101)\%\ 9$
2. Compute for each key:
   - $h_7(22)=((23\times22+88)\%\ 101)\%\ 9=(594\%\ 101)\%\ 9=89\%9=8$
   - $h_7(37)=((23\times37+88)\%\ 101)\%\ 9=(939\%\ 101)\%\ 9=30\%9=3$
   - $h_7(40)=((23\times40+88)\%\ 101)\%\ 9=(1008\%\ 101)\%\ 9=99\%9=0$

Perfect Hashing Example

Final Hash Table with Perfect Hashing

Primary Table T

Index	Keys (Secondary Table S_j)
0	{10} (Stored at $S_0[0]$)
1	-
2	{60, 75} (Stored in $S_2[0], S_2[1]$)
3	-
4	-
5	{70} (Stored at $S_5[0]$)
6	-
7	{40, 37, 22} (Stored in $S_7[0], S_7[3], S_7[8]$)
8	-

Secondary Hash Tables

- S_0: {10 → $S_0[0]$ }
- S_2: {60 → $S_2[0]$, 75 → $S_2[1]$}
- S_5: {70 → $S_5[0]$ }
- S_7: {40 → $S_7[0]$, 37 → $S_7[3]$,22 → $S_7[8]$}

Results:

- First-level hashing groups keys into **buckets**.
- Second-level hashing ensures **no collisions** by choosing an appropriate hash function for each bucket.
- This guarantees **O(1)** worst-case search time.

Points to be remember during Perfect Hashing

- **Main hash table**
  - choose size $m = n$ where n is the number of data items, choose prime $p >$ *the biggest key value*.
  - Randomly choose a and b to get primary hash function $h(k) = ((ak + b)\%p)\%m$.
- **Test h as follows:**
  - For each key k, work out the home cell $h(k)$. Keep a count n_i for each cell i of how many keys hash to i.
  - Check whether space required is too big: is the sum of all the n_i^2 giving a total > $2n$? Then h is not good enough so repeat the process with new a, b for a new primary hash function.
- If the primary hash function h is good enough:
  - For each slot i, get the secondary hash function h_i by setting $p_i = p$ (i.e. p doesn't change), setting $m_i = n_i^2$), and choosing a_i and b_i randomly.
  - Check to make sure that the resulting h_i doesn't cause any collisions within the secondary table.

Try it out

Q1. Store the following keys, $K = \{8, 22, 36, 75, 61, 13, 84, 58\}$ into hash table using perfect hashing. Use primary hash function $h(k) = ((ak + b)\ \%p\ )\%m$ and secondary hash function $h_j(k) = ((a_j k + b_j)\ \%p\ )\%m_j$

Solution:

Step 1: Randomly generate values for **a** and **b**, select **p > K** ($p = 87, a = 64, b = 5$)

Step 2: For each key **k**, work out home cell **h(k)**. Keep a count. Assume ***m=8 (next slide)***

Step 3: Check if $\sum_{j=0}^{m-1} {n_j}^2 < 2n$

$$\sum_{j=0}^{m-1} {n_j}^2 = 0^2+1^2+2^2+0^2+1^2+1^2+1^2+2^2 < 2 * 8$$

$$= 0+1+4+0+1+1+1+4 = \mathbf{12} < \mathbf{16}$$ ✅

Step 4: Populate sub-tables, calculating new hash functions (**Secondary Hash Table**)[*next->next (slide)*]

Perfect Hashing Example

Computer the first level hashing using outer hash function [$h(k)=((64k+5)\%\ 87)\%\ 8$]

Key (k)	Compute h(k)	First-Level Slot (T[h(k)])
8	((64×8+5)% 87)% 8=82% 8=2	2
22	((64×22+5)% 87)% 8=21% 8=5	5
36	((64×36+5)% 87)% 8=47% 8=7	7
75	((64×75+5)% 87)% 8=20% 8=4	4
61	((64×61+5)% 87)% 8=81% 8=1	1
13	((64×13+5)% 87)% 8=54% 8=6	6
84	((64×84+5)% 87)% 8=74% 8=2	2
58	((64×58+5)% 87)%8= 63%8=7	7

			Count
0	/	0	0
1	/	61	1
2	/	8, 84	2
3	/	0	0
4	/	75	1
5	/	22	1
6	/	13	1
7	/	36, 58	2

Perfect Hashing Example

Secondary Hash Tables Construction

1.Bucket S_1 (*Keys:* $\{61\}$, $m_1=1^2=1$, randomly chosen $a_1=0$ *and* $b_1=0$)

1. Use $h_1(k)=(0k+0)\%\ 87\ \%\ 1 = h_1(61)=(0\times61+0)\%\ 87\ \%\ 1 = (0\%87)\%1 = 0\%1=0$

2.Bucket S_2 (*Keys:* $\{8, 84\}$, $m_2=2^2=4$, randomly chosen $a_2=82$ *and* $b_2=53$)

1. $h_2(k)=((82k+53)\%\ 87)\%\ 4$
2. Compute for each key:
   - $h_2(8)=((82\times8+53)\%\ 87)\%\ 4=(709\%\ 87)\%\ 4=13\%4=1$
   - $h_2(84)=((82\times84+53)\%\ 87)\%\ 4=(6941\%\ 87)\%\ 4=68\%4=0$

3.Bucket S_4 (*Keys:* $\{75\}$, $m_4=1^2=1$, randomly chosen $a_4=0$ *and* $b_2=0$)

1. $h_4(k)=((0k+0)\%\ 87)\%\ 1$
2. Compute for a key:
   - $h_4(75)=((0\times75+0)\%\ 87)\%\ 1=(0\%\ 87)\%\ 1=0\%1=0$

4.Bucket S_5 (*Keys:* $\{22\}$, $m_5=1^2=1$, randomly chosen $a_5=0$ *and* $b_5=0$)

1. $h_5(k)=((0k+0)\%\ 87)\%\ 1$
2. Compute for a key:
   - $h_5(22)=((0\times22+0)\%\ 87)\%\ 1=(0\%\ 87)\%\ 1=0\%1=0$

Perfect Hashing Example

Secondary Hash Tables Construction

5. Bucket S_6 (*Keys:* $\{13\}$, $m_6=1^2=1$, randomly chosen $a_6=0$ *and* $b_6=0$)

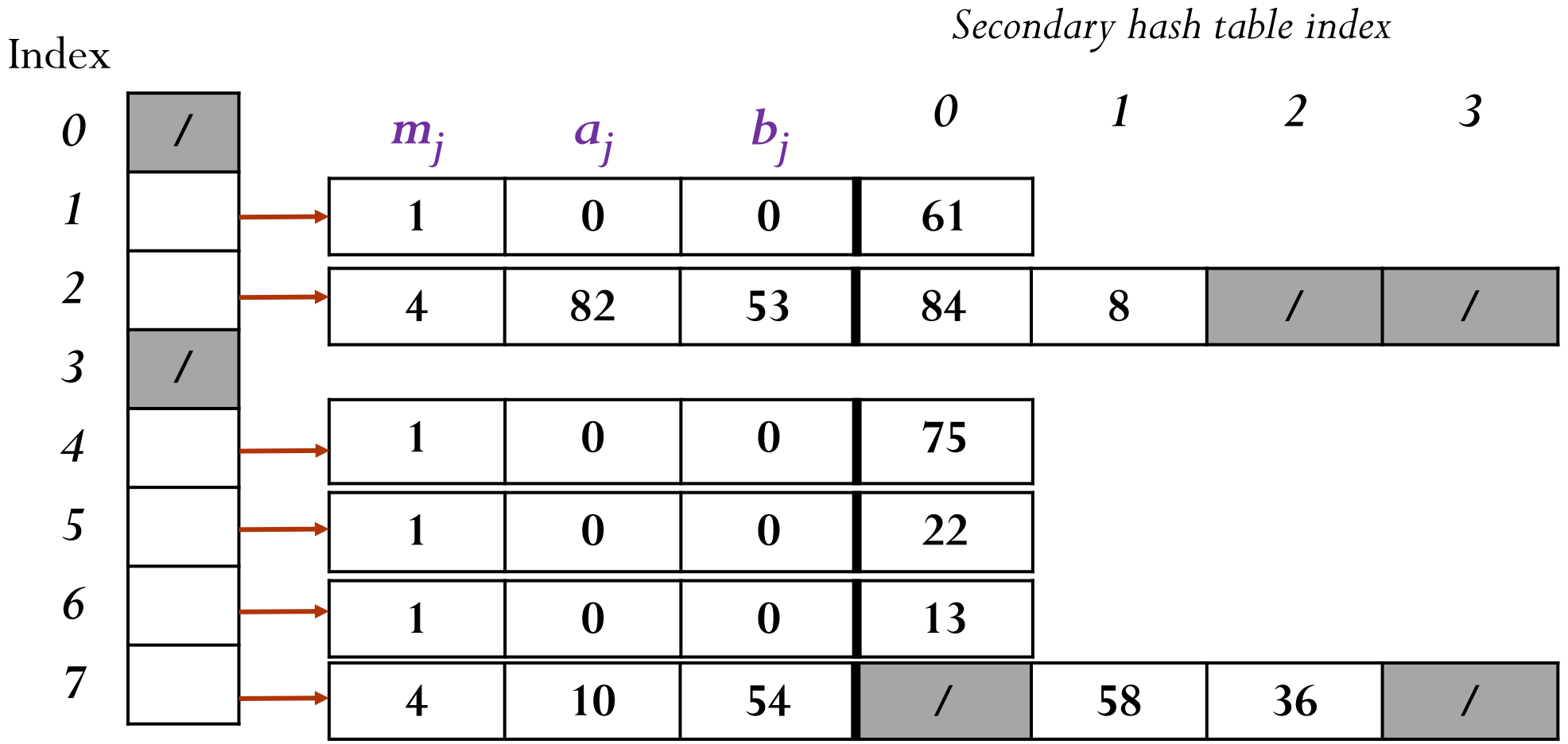
1. $h_6(k)=((0k+0)\%\ 87)\%\ 1$
2. Compute for a key:
   - $h_6(13)=((0\times13+0)\%\ 87)\%\ 1=(0\%\ 87)\%\ 1=0\%1=0$

6. Bucket S_7 (*Keys:* $\{36,58\}$, $m_7=2^2=4$, randomly chosen $a_7=10$ *and* $b_7=54$)

1. $h_7(k)=((10k+54)\%\ 87)\%\ 4$
2. Compute for each key:
   - $h_7(36)=((10\times36+54)\%\ 87)\%\ 4=(414\%\ 87)\%\ 4=66\%4=2$
   - $h_7(58)=((10\times58+54)\%\ 87)\%\ 4=(634\%\ 87)\%\ 4=25\%4=1$

Perfect Hashing Example

Final Hash Table (try it out) with Perfect Hashing



No S_j keys hash to the same sub-slots.-ALL DONE

Try it out (Very Simple)

Q2. Store the following keys, $K = \{12, 14, 17, 21, 24, 27, 31, 34, 37\}$ into hash table using perfect hashing. Use primary hash function $h(k) = k \% m$ where $m = 10$ and secondary hash function $h_j(k) = k \% m_j$ where $m_j = n_j^2$

Solution:

Keys: {12, 14, 17, 21, 24, 27, 31, 34, 37 }

Primary hash function *h(key) = key % 10*

First Level hashing

Keys	h(key)	Hash Value	Bucket Count
12	12%10	2	1
14	14%10	4	1
17	17%10	7	1
21	21%10	1	1
24	24%10	4	2
27	27%10	7	2
31	31%10	1	2
34	34%10	4	3
37	37%10	7	3

Primary Hash Table

Bucket	Keys
0	/
1	{21,31}
2	{12}
3	/
4	{14,24,34}
5	/
6	/
7	{17,27,37}
8	/
9	/

Secondary Hash Table Construction

Bucket 1: $m_1 = n_1^2 = 2^2 = 4$

h1(21) = 21%4 = 1

h1(31) = 31%4 = 3

Bucket 2: $m_2 = n_2^2 = 1^2 = 1$

h2(12) = 12%1 = 0

Bucket 4: $m_4 = n_4^2 = 3^2 = 9$

h4(14) = 14%9 = 5

h4(24) = 24%9 = 6

h4(34) = 34%9 = 7

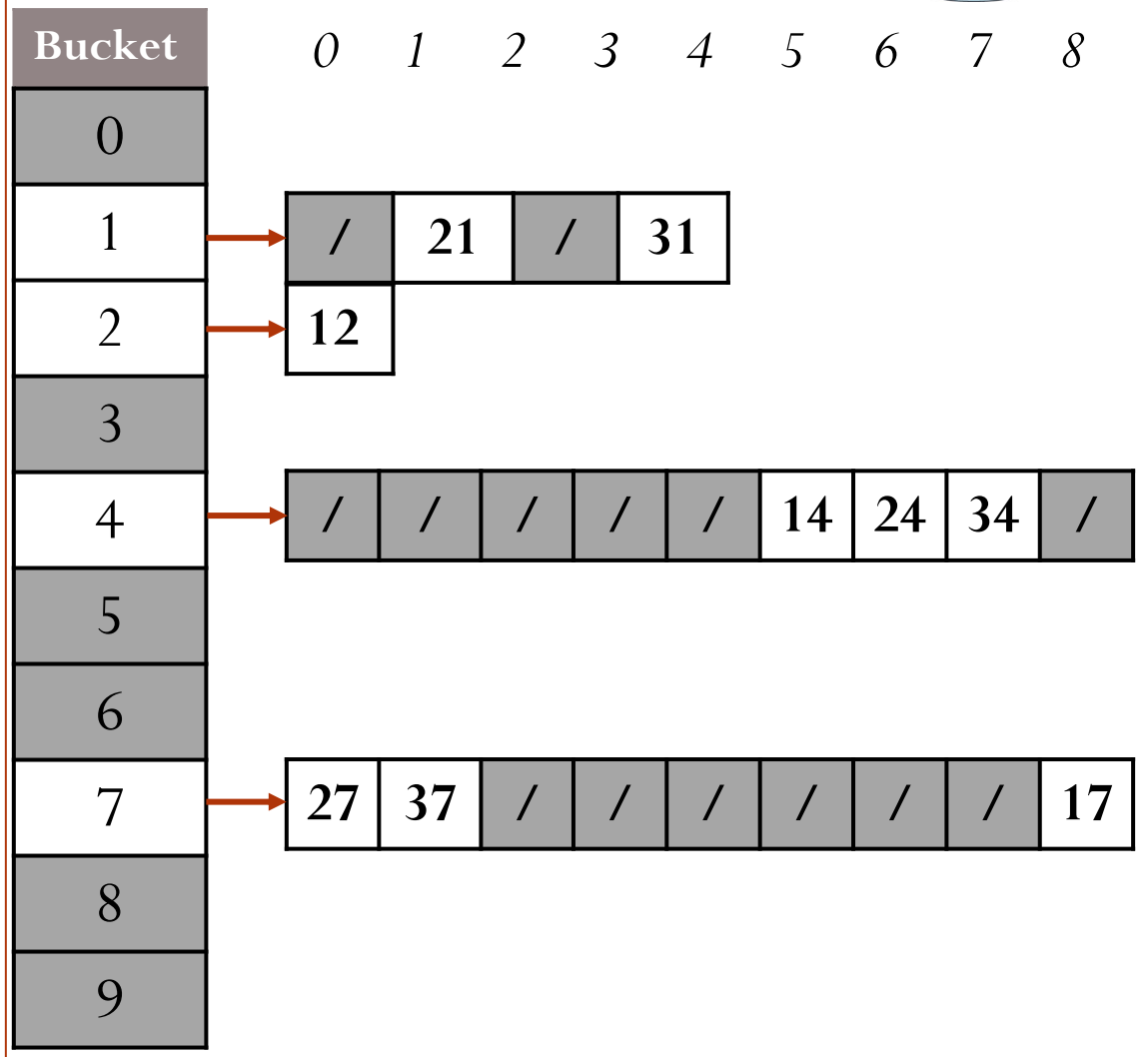
Bucket 7: $m_7 = n_7^2 = 3^2 = 9$

h7(17) = 17%9 = 8

h7(27) = 27%9 = 0

h7(37) = 37%9 = 1

Final Hash Table



Important points of Perfect Hashing

- When $n_j = m_j = 1$, we don't really need a hash function for slot j; when we choose a hash function $h_{a,b}(k) = ((ak + b) \% p) \% m_j$ for such a slot, we just use $a = b = 0$.
- If we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions, then the probability of there being any collisions is less than $1/2$.
- If we store n keys in a hash table of size $m=n$ using a hash function h randomly chosen from a universal class of hash functions, then $E\left[\sum_{j=0}^{m-1} {n_j}^2\right] < 2n$, where n_j is the number of keys hashing to slot j .
- If we store n keys in a hash table of size $m=n$ using a hash function h randomly chosen from a universal class of hash functions and we set the size of each secondary hash table to $m_j=n_j^2$ for $j= 0, 1, \ldots, m-1$, then the expected amount of storage required for all secondary hash tables in a perfect hashing scheme is less than $2n$.
- If we store n keys in a hash table of size $m=n$ using a hash function h randomly chosen from a universal class of hash functions and we set the size of each secondary hash table to $m_j=n_j^2$ for $j= 0, 1, \ldots, m-1$, then the probability that the total storage used for secondary hash tables exceeds $4n$ is less than $1/2$.

Cuckoo Hashing

Cuckoo Hashing

- Cuckoo hashing is a hash table scheme using two hash tables T_1 and T_2 each with r buckets with independent hash functions h_1 and h_2 each mapping a universe **U** to bucket locations $\{0,1,2,\ldots..,r-1\}$.
- A key x can be stored in exactly one of the locations $T_1[h_1(x)]$ and $T_2[h_2(x)]$.
- A lookup operation in Cuckoo Hashing examines both locations $T_1[h_1(x)]$ and $T_2[h_2(x)]$ and succeeds if the key **x** is stored in either location.
- Cuckoo hashing is a hashing technique that guarantees $O(1)$ worst-case lookup time by using two hash tables and two hash functions.
- Each key is stored in one of the two tables, and if a collision occurs, the existing key is "evicted" and relocated to its alternate position in the other table.
- This process continues until all keys are placed or a maximum number of displacements is reached (indicating a failure).

Cuckoo Hashing

How Cuckoo hashing works?

1. **Insertion:**
   - A key k is hashed using two hash functions: $h_1(k)$ and $h_2(k)$, which determine two possible locations for k.
   - If the first position $h_1(k)$ is free, place k there.
   - Otherwise, evict the existing key and move it to its alternative location $h_2(k)$, repeating the process if necessary.
2. **Collision Handling (Rehashing):**
   - If an insertion cycle continues for too long (i.e., looping occurs), the table is **rehashed** with new hash functions.
3. **Lookup:**
   - Check both possible locations $h_1(k)$ and $h_2(k)$.
   - If found in either, return success; otherwise, return failure.
4. **Deletion:**
   - Remove the key from either $h_1(k)$ or $h_2(k)$.

Cuckoo Hashing Example

Example: Store the given Keys, $K=\{8,22,36,75,61,13,84,58\}$ into the hash table using cuckoo hashing. Use the following hash functions: $h_1(k)=k \% 8$ and $h_2(k)=(k/8)\% 8$.

Step 1: Initialize two empty hash table with table size 8.

Index →	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
T_1	/	/	/	/	/	/	/	/
T_2	/	/	/	/	/	/	/	/

Step 2 : Insertion (*Calculate hash values for keys arrived in an order*)

$h_1(8) = 8 \% 8 = 0$

Insert 8 at index 0 of *Table 1(T_1)*

Index →	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
T_1	8	/	/	/	/	/	/	/
T_2	/	/	/	/	/	/	/	/

Cuckoo Hashing Example

$h_1(22) = 22 \% 8 = 6$

Insert 22 at index 6 of *Table 1(T_1)*

Index →	0	1	2	3	4	5	6	7
T_1	8	/	/	/	/	/	22	/
T_2	/	/	/	/	/	/	/	/

$h_1(36) = 36 \% 8 = 4$

Insert 36 at index 4 of *Table 1(T_1)*

Index →	0	1	2	3	4	5	6	7
T_1	8	/	/	/	36	/	22	/
T_2	/	/	/	/	/	/	/	/

Cuckoo Hashing Example

$h_1(75) = 75 \% 8 = 3$

Insert 75 at index 3 of *Table 1(T_1)*

Index →	0	1	2	3	4	5	6	7
T_1	8	/	/	75	36	/	22	/
T_2	/	/	/	/	/	/	/	/

$h_1(61) = 61 \% 8 = 5$

Insert 61 at index 5 of *Table 1(T_1)*

Index →	0	1	2	3	4	5	6	7
T_1	8	/	/	75	36	61	22	/
T_2	/	/	/	/	/	/	/	/

Cuckoo Hashing Example

$h_1(13) = 13 \% 8 = 5$

$T_1[5]$ is occupied by 61. Evict 61 and place 13 in $T_1[5]$.

Relocate 61 to T_2: $h_2(61) = (61/8)\%8 = 7\%8 = 7$ *(Insert 61 at index 7 of T_2)*

Index →	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
T_1	8	/	/	75	36	13	22	/
T_2	/	/	/	/	/	/	/	61

$h_1(84) = 84 \% 8 = 4$

$T_1[4]$ is occupied by 36. Evict 36 and place 84 in $T_1[4]$.

Relocate 36 to T_2: $h_2(36) = (36/8)\%8 = 4\%8 = 4$ *(Insert 36 at index 4 of T_2)*

Index →	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
T_1	8	/	/	75	84	13	22	/
T_2	/	/	/	/	36	/	/	61

Cuckoo Hashing Example

$h_1(58) = 58 \% 8 = 2$

Insert 58 at index 2 of *Table 1* (T_1)

Index →	0	1	2	3	4	5	6	7
T_1	8	/	58	75	84	13	22	/
T_2	/	/	/	/	36	/	/	61

	Stored Normally using $h_1(k)$ in T_1
	Original key is evicted and relocated to T_2
	Relocated to T_2

All keys are successfully inserted without cycles or failures. Lookup and deletion operations can now be performed in $O(1)$ time.

Cuckoo Hashing Example

Try it out

Q2. Store the following keys, $K = \{20, 33, 6, 45, 61, 11, 231, 90, 101\}$ into the hash table using cuckoo hashing. Use hash function $h_1(k) = k \% 11$ and $h_2(k) = k \% 13$. Use both hash table size 15.

Key (k)	h1(k) =k%11	h2(k) =k%13	Final Placement
20	9		Insert 20 at index 9 of T1
33	0	7	Insert 33 at index 0 of T1
6	6	6	Insert 6 at index 6 of T1
45	1		Insert 45 at index 1 of T1
61	6		Collison, evict 6 and Insert 61 at index 6 of T1 and relocate 6 into index 6 of T2
11	0	11	Collison, evict 33 and insert 11 at index 0 of T1 and relocate 33 into index 7 of T2
231	0		Collison, evict 11 and insert 231 at index 0 of T1 and relocate 11 into index 11 of T2
90	2	12	Insert 90 at index 2 of T1
101	2		Collison, evict 90 and insert 101 at index 2 of T1 and relocate 90 into index 12 of T2

Final hash Tables

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T1	231	45	101	-	-	-	61	-	-	20	-	-	-	-	-
T2	-	-	-	-	-	-	6	33	-	-	-	11	90	-	-

4.4. Hashing in Distributed Systems

4.5. Case Studies in GPUs and FPGAs

Student Presentations

Hashing in Distributed Systems

Hashing in Distributed Systems

- A distributed system is a network that consists of autonomous computers that are connected using a distribution middleware. They help in sharing different resources and capabilities to provide users with a single and integrated coherent network.
- One of the ways hashing can be implemented in a distributed system is by taking hash Modulo of a number of nodes.

Web Caching

- ***Problem***: if the page is requested over and over again, it's wasteful to repeatedly download it from the server.
- ***Solution***: use a Web cache
  - which stores a local copy of recently visited pages
  - When a URL is requested, local cache is checked first.
  - If the page is in local cache, send it directly to the browser *no need to contact the original server*
  - If it is not in the local cache, it is downloaded from the suitable server and the result is sent to both browser and local cache for future use
- Caching is good
  - much faster response time
  - Fewer request to far away servers- less network traffic, less congestion in the queues at network switches

Hashing in Distributed Systems

Where to place Web Cache?

1. **Individual web cache:** At each end user machine
2. **Shared web cache :** Implement a web cache that is shared by many users
   - **Problem**:
     - aggregated cache might be very large
     - So it might not fit on a single machine
- **Solution**:
  - **IDEA**: cache over multiple machines
  - implement a shared cache at a large scale by spreading the cache over multiple machines
  - Suppose if the the shared cache is spread over 100 machines- Where should you look for a cached copy of the Web page?
  - ***Bad Solution***: poll all 100 caches for a copy (infeasible)
  - ***Good Solution*** - **Hashing** (mapping from URLs to caches)

Hashing in Distributed Systems

Mapping URLs to Caches using Simple Hashing

- Suppose there are caches named $\{0,1,2,3...,n-1\}$ and x is the URL of a web page then store the web page x at the cache server named $h(x) \bmod n$
- $h(x)$ is probably something like a 32-bit value, representing an integer (2^{32}) that is very bigger than n; so we apply the "$\bmod n$" operation to recover the name of one of the caches.
- **Problem**:
  - Caches are not static
  - A new Web cache server can be added i.e. $n+1$
  - Or an existing can fail or loose connection i.e. $n-1$
  - Web cache servers are changing over time
- What will happen if n changes in $h(x) \bmod n$?
  - For an object $h(x) \bmod n \neq h(x) \bmod n + 1$
  - So changing n forces almost all objects to relocate (rehashing)- which is costly for large data
  - This is a disaster for applications where n constantly changes

Hashing in Distributed Systems

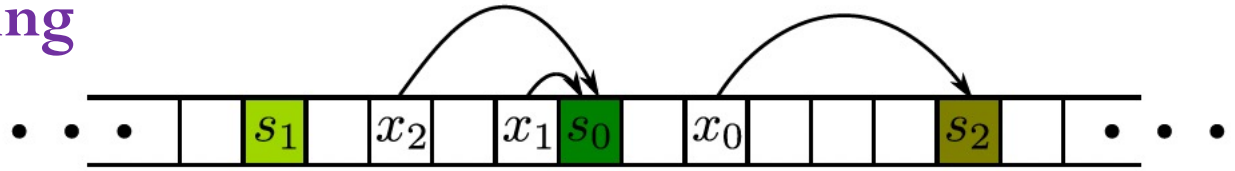
Solution: Consistent Hashing

- Goal: almost all objects stay assigned to the same cache even as the number ***n*** of caches changes
- Key Idea:
  - Apply Hashing for both
    - names of objects (URLs) ***x*** (as before) – ***h(x)***
    - names of cache servers ***s*** (in addition) - ***h(s)***
    - Both need to be hashed to the same range.

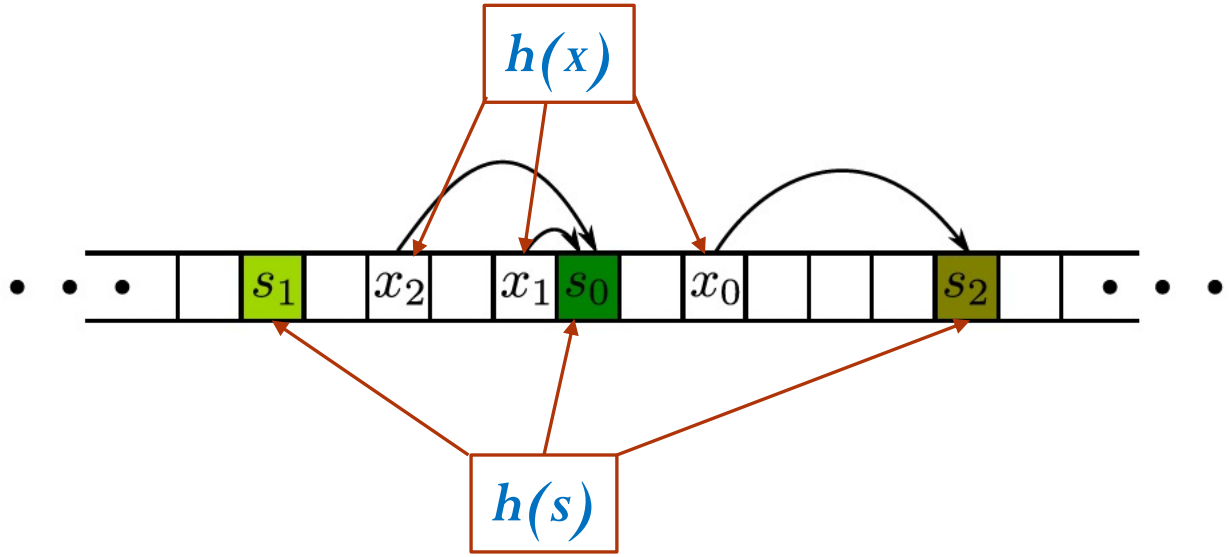
- Which objects are assigned to which caches
  - Given an object ***x*** that hashes to the bucket ***h(x)***
  - We scan buckets to the right of ***h(x)*** until we find a bucket ***h(s)*** to which the name of some cache ***s*** hashes. (We wrap around the array, if need be.)
  - We then designate ***s*** as the cache responsible for the object ***x***.

Hashing in Distributed Systems

Consistent hashing



Each element of the array above is a bucket of the hash table.
Each object **x** is assigned to the first cache server **s** on its right.



Each element of the array above is a bucket of the hash table.
Each object **x** is assigned to the first cache server **s** on its right.

Hashing in Distributed Systems

Consistent hashing

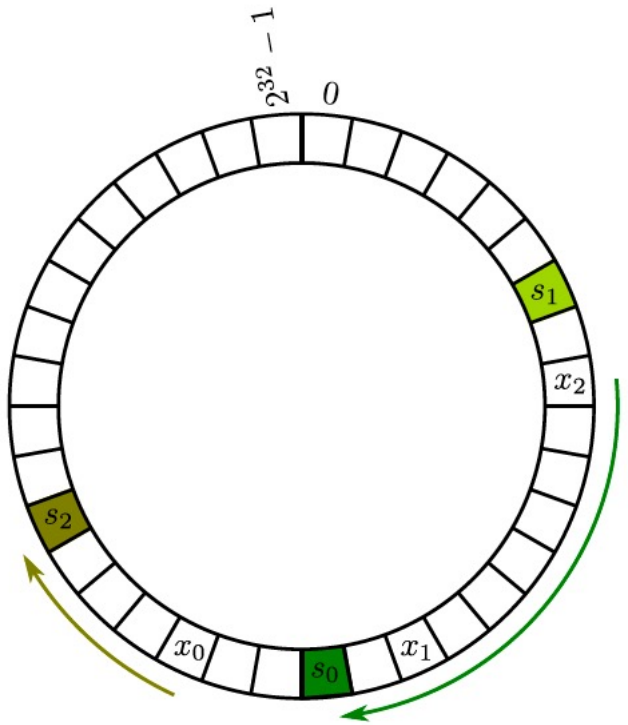


Fig:We glue 0 and $2^{32}-1$ together, so that objects are instead assigned to the cache server that is closest in the clockwise direction. This solves the problem of the last object being to the right of the last cache.

- n caches partition the circle into n segments, with each cache responsible for all objects in one of these segments (**1/n**).

Consistent hashing Properties

- **Property 1:** The expected load on each of the ***n*** cache servers is exactly a ***1/n*** fraction of the objects.
- **Property 2**:
  - Adding the ***n^{th}*** cache causes only a ***1/n*** fraction of the objects to relocate (in best case- it occurs if ***h(s)*** evenly distribute ***s***)
  - clearly the objects on the new cache have to move from where they were before.

Hashing in Distributed Systems

Consistent hashing (Adding a new cache)

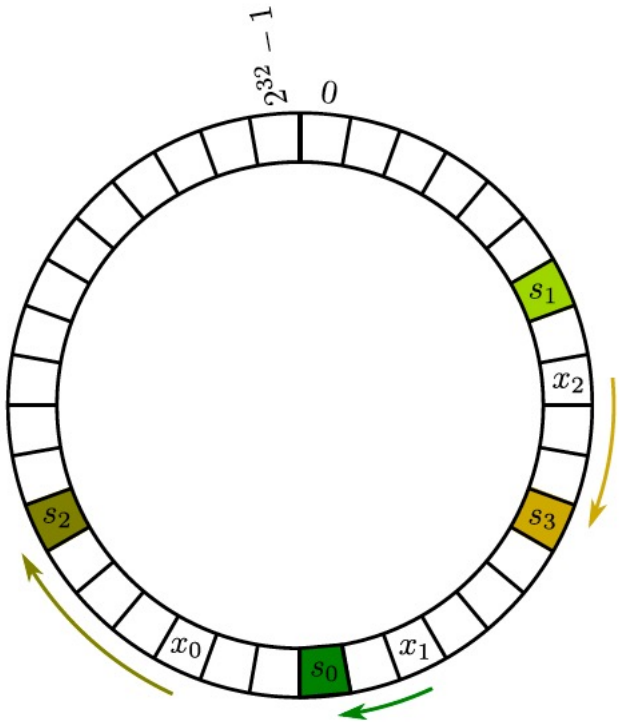


Fig: Adding a new cache server s_3.

- On average, adding n^{th} cache may not move whole $1/n$ fraction of the objects
- Only x_2 is moved from s_0 to s_3 . The x_1 is still belongs s_0

Consistent hashing (Removing a cache)

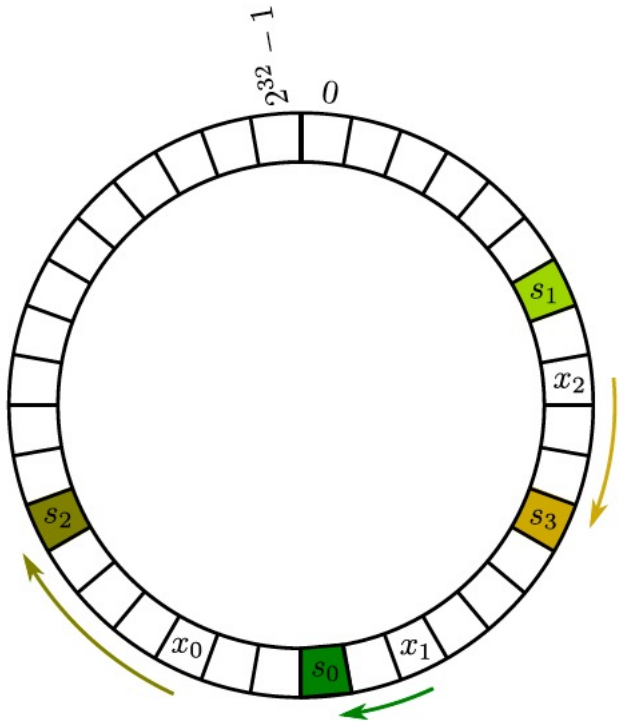


Fig: Removing a cache server s_3.

- Removing a cache server say s_3, First assign all the objects (for which s_3 is responsible) to the successor cache server, s_0 in our example and remove s_3

Hashing in Distributed Systems

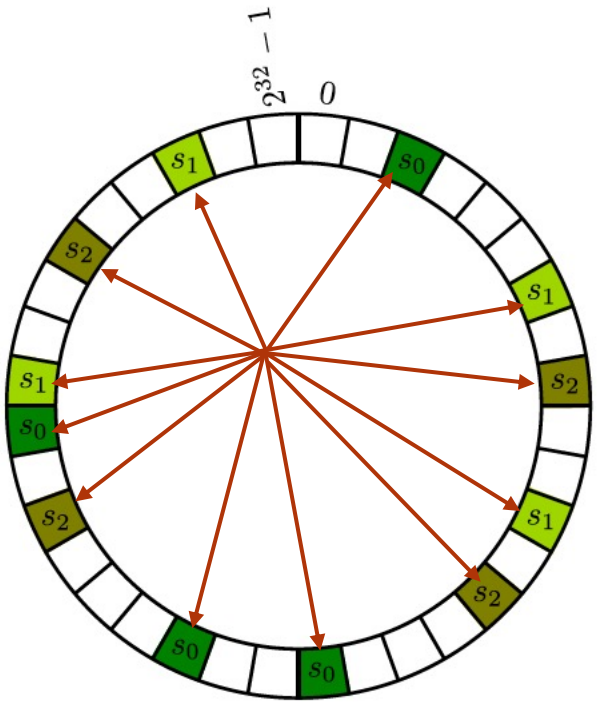
Consistent hashing (Lookup and Insert)

- Given an object x,
- we use efficient successor operation to lookup or insert
- successor operation scans rightward/clockwise for s
  - stops at s where $h(s) \geq h(x)$
- How to make *successor* operation?
  - Use a data structure to store cache names
  - Which data structure can make operation faster?
    - Hash table ? *-not good- it doesn't maintain order*
    - A heap? *-not good- it maintains only for partial order (min or max at a time)*
    - A Binary Search Tree? *maintains order of elements but must be balanced* e.g. Red Black tree
    - Finding the cache responsible for storing a given object x takes time $O(\log n)$, where n is the number of caches.

Hashing in Distributed Systems

Consistent hashing (Reducing the variance)

- Expected load of each cache server s is a $1/n$ fraction of the objects but in real the load of each cache will vary.
- Picking n random points on the circle, very unlikely to get a perfect partition of the circle into equal-sized segments.
- An easy way to reduce this variance is to make k "virtual copies" of each cache s, implemented by hashing its name with k different hash functions to get $h_1(s), \ldots, h_k(s)$.
- For example, with cache servers $\{0, 1, 2\}$ and $k = 4$, we choose **12** points on the circle: **4** labeled "**0**", **4** labeled "**1**", and **4** labeled "**2**".
- Objects are assigned as before using from $h(x)$
- We scan x rightward/clockwise until we encounter one of the hash values of some cache s, and **s** is responsible for storing **x**



Hashing in Distributed Systems

Consistent hashing (Reducing the variance)

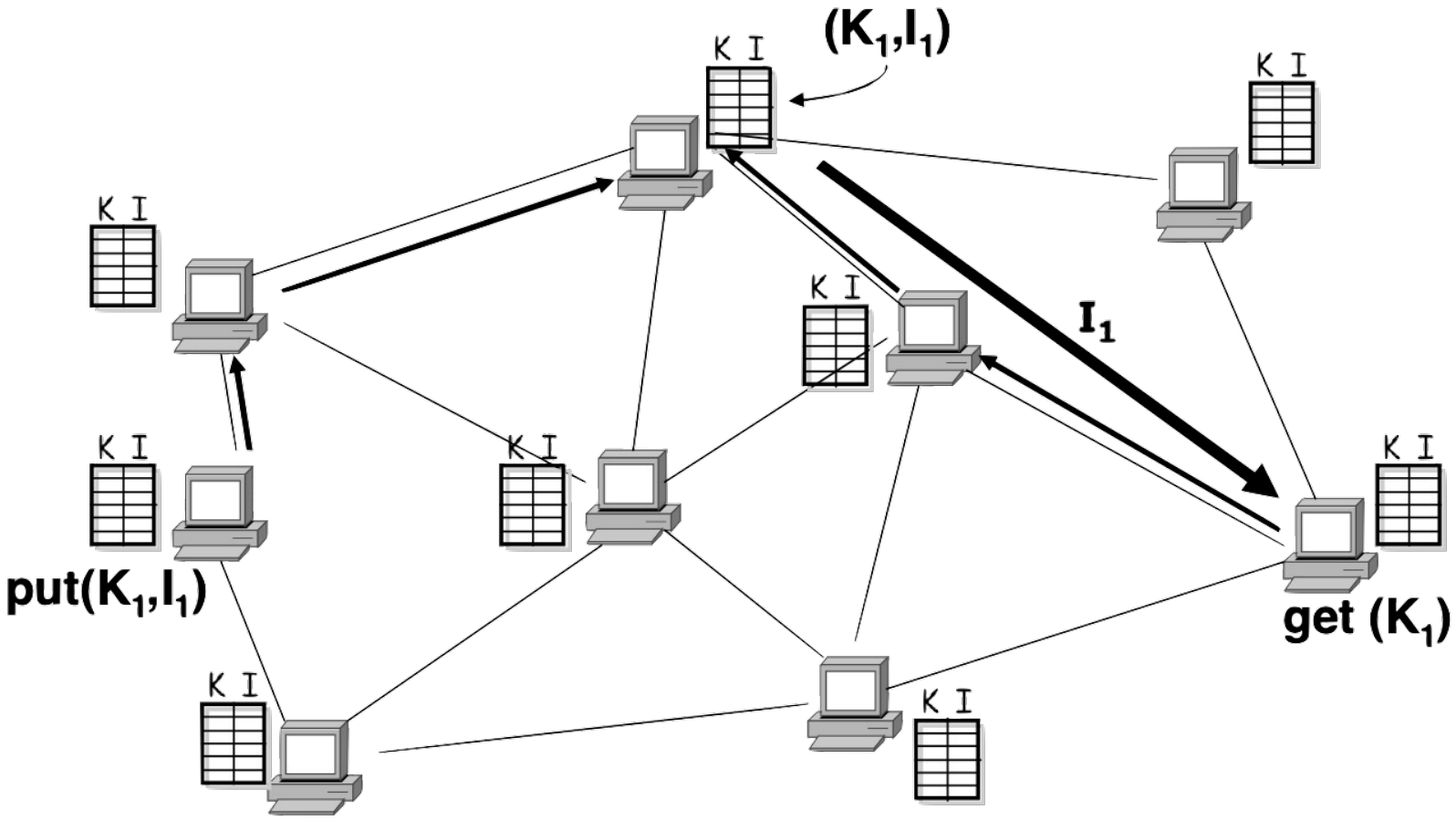
- Still expected load = $1/kn$ for each s_k
- Still one cache may get more objects than expected
- Choosing $k \approx log_2 n$ is large enough to obtain reasonably balanced loads
- Virtual copies are also useful for dealing with heterogeneous caches that have different capacities.
- The sensible approach is to make the number of virtual copies of a cache server proportional to the server's capacity
- for example, if one cache is twice as big as another, it should have twice as many virtual copies

Distributed Hash System

- Distributed hash tables(DHT) are decentralized, so all nodes form the collective system without any centralized coordination. They are generally fault-tolerant because data is replicated across multiple nodes. Distributed hash tables can scale for large volumes of data across many nodes.
- DHS is a decentralized system and uses a **distributed hash table** (DHT) to store/retrieve data efficiently across multiple nodes in a network.
- It is commonly used in peer-to-peer (P2P) networks, distributed storage systems, blockchain technologies where data is stored as ***(key, value)*** pairs
- The ***key*** is hashed to determine the node responsible for storing the (*key*, *value*)
- Popular Distributed Hash Systems
  - Chord
  - Kademlia
  - Pastry
  - Tapestry

Distributed Hash System

- Hash Table Interface: and *put*(*key*, *item*) *get*(*key*)



Distributed Hash System

Case Studies

- **BitTorrent and distributed hash table**
  - BitTorrent is a peer-to-peer (P2P) file-sharing protocol that allows users to download and upload files without relying on a central server. One of the key technologies that make BitTorrent decentralized is the Distributed Hash Table (DHT), which enables trackerless torrents.

- **Amazon DynamoDB and Cord/Kademila**
  - DynamoDB is inspired by Amazon's original Dynamo system, which is a DHT-like key-value store. DynamoDB is not a pure DHT, but it borrows many principles from DHTs, especially from Amazon Dynamo. DynamoDB partitions data using consistent hashing. AWS manages virtual nodes to balance load, just like DHT overlays (e.g., Chord, Kademlia).

4.5. Case Studies in GPUs and FPGAs

Student Presentations

Hashing in GPU

- Hashing on a GPU (Graphics Processing Unit) can be significantly faster than on a CPU (Central Processing Unit) due to the parallel processing capabilities of GPUs.
- GPUs are designed to handle thousands of threads simultaneously, making them well-suited for tasks that can be parallelized, such as hashing.

Why GPUs for Hashing?

- Parallelism: GPUs consist of thousands of cores that can execute multiple hashing operations simultaneously.
- High Throughput: Suitable for applications that require large-scale hashing computations.
- Optimized Memory Access: GPU memory bandwidth is high, which benefits hash table lookups and cryptographic hashing.

Applications of GPU Hashing

- Cryptographic Hashing: Used in blockchain mining (e.g., Bitcoin, Ethereum).
- Data Deduplication: Storing unique data using hash-based indexing.
- Approximate Matching: Hashing is used in similarity searches and machine learning.
- Bioinformatics: DNA sequence alignment and other biological data analysis.
- Network Security: Hash-based intrusion detection and packet filtering.

Hashing in GPU

Types of Hashing Implemented on GPUs

A. Cryptographic Hashing

- Used for password storage, digital signatures, and blockchain mining. Examples: SHA-256 (Bitcoin mining), Ethash (Ethereum mining), BLAKE2, Keccak, MD5 (for security applications)

B. Hash Tables on GPU

- Used for fast key-value storage and retrieval.
  - Cuckoo Hashing: Resolves collisions using multiple hash functions.
  - Linear Probing: Places elements in the next available slot if a collision occurs.
  - Perfect Hashing: Constructs a minimal collision-free hash table.

C. Approximate Hashing

- SimHash: Used for near-duplicate detection in web searches.
- Locality-Sensitive Hashing (LSH): Helps in approximate nearest-neighbor searches.

Hashing in GPU

GPU-Optimized Hashing Libraries

- CUDA Thrust: NVIDIA's high-level parallel algorithms library for GPU-based hashing.
- cuDPP (CUDA Data Parallel Primitives): Provides efficient hash table implementations.
- GPUMurmurHash: A GPU version of MurmurHash for fast non-cryptographic hashing.
- OpenCL-Based Hashing: Can be used for cross-platform hashing implementations.

Performance Considerations

- Memory Bottleneck: Hash tables require frequent memory access; optimizations like shared memory and caching improve performance.
- Load Balancing: Uneven distribution of hash values can create workload imbalances.
- Collision Handling: Efficient methods like double hashing and cuckoo hashing are required for performance optimization.
- Atomic Operations: Synchronization issues arise when multiple threads update hash tables, requiring atomic operations.

Hashing in GPU

Future Trends

- AI-Accelerated Hashing: Using machine learning to optimize hash function performance.
- FPGA vs. GPU Hashing: Some applications explore FPGAs for lower power consumption.
- Homomorphic Hashing: Secure computing techniques for privacy-preserving applications.

Approximate hashing

- Approximate hashing refers to techniques used to compute hash values in such a way that they allow for efficient comparisons and searches, even when the data being compared isn't exactly identical.
- Useful in applications where exact matching is too computationally expensive or impractical, such as in large datasets or noisy data environments.

Hashing in GPU

Key Concepts in Approximate Hashing

1. **Near-Identical Comparisons**:
   - Approximate hashing techniques allow us to determine whether two pieces of data are similar (rather than identical) by comparing their hash values.
   - The idea is that similar data will produce "close" or "similar" hash values, making it easier to identify near-duplicates without having to perform expensive exact comparisons.
2. **Tolerance for Small Differences**:
   1. Unlike standard hashing, where any change in the input leads to a completely different hash, approximate hashing is designed to produce similar hash values for inputs that are slightly different.
   2. It is helpful in applications like text search, image comparison, or even error-tolerant indexing in databases.

Hashing in GPU

Common Techniques in Approximate Hashing

1. **Locality-Sensitive Hashing (LSH)**:
   - The idea behind LSH is to map similar data points into the same "bucket" with high probability while keeping dissimilar points in different buckets. This drastically reduces the number of comparisons needed in tasks like nearest-neighbor searches.
   - LSH works by creating hash functions that preserve the "locality" or proximity of data points in a specific way. For instance:
     - In text, it might map similar words to the same bucket.
     - In images, it might map similar pixel patterns to the same bucket.
2. **SimHash**:
   1. SimHash is a technique used to detect near-duplicate documents or items in large datasets.
   2. The idea is to hash features (such as shingles or fingerprints) of the data and then produce a compact hash value that reflects the content. Minor differences in input data will lead to hash values that are still "close," enabling approximate matches.
   3. Commonly used in web search engines for document similarity.

Hashing in GPU

Applications of Approximate Hashing

1. **Duplicate Detection**: Identifying near-duplicate documents, images, or files. For example, social media platforms or search engines can use approximate hashing to detect similar content without needing to perform exact matches.
2. **Plagiarism Detection**: Tools for plagiarism detection in text or code can use approximate hashing to find documents or code snippets that are similar but not identical.
3. **Data Deduplication**: Approximate hashing can help in storage systems by finding and removing near-duplicate data, thus saving space.
4. **Image Comparison**: In image processing, approximate hashing is used to find similar images (e.g., finding duplicates or similar images in large datasets). Hashing techniques like **dHash** (difference hash) or **pHash** (perceptual hash) are commonly used for this.
5. **Search and Recommendation Systems**: Approximate hashing can improve the efficiency of recommendation systems by quickly finding items that are similar to the user's preferences.

Hashing in GPU

Example: Near-Duplicate Text Detection Using SimHash

Let's take two sample sentences:

- **Text 1:** "I love machine learning"
- **Text 2:** "I adore machine learning"

1. Feature Extraction:

- For **Text 1** ("I love machine learning"):
  - Words: ["I", "love", "machine", "learning"]
- For **Text 2** ("I adore machine learning"):
  - Words: ["I", "adore", "machine", "learning"]

2. Hash Each Feature:

- We will hash each word into a 4-bit binary value for simplicity.
- Let's assume the hash function for each word returns:

- **Text 1**:
  - "I" -> 0101
  - "love" -> 1000
  - "machine" -> 1110
  - "learning" -> 1101
- **Text 2**:
  - "I" -> 0101
  - "adore" -> 1011
  - "machine" -> 1110
  - "learning" -> 1101

Hashing in GPU

3. Combine the Hashes:

- Now, we'll combine these hashes for both texts. For each position in the binary hash (4 bits in our case), we sum the corresponding bits from all the words.
- **For Text 1**:
  - Sum the bits position-wise:
    - 1st position: 0 + 1 + 1 + 1 = 3
    - 2nd position: 1 + 0 + 1 + 1 = 3
    - 3rd position: 0 + 0 + 1 + 0 = 1
    - 4th position: 1 + 0 + 0 + 1 = 2
- **For Text 2**:
  - Sum the bits position-wise:
    - 1st position: 0 + 1 + 1 + 1 = 3
    - 2nd position: 1 + 0 + 1 + 1 = 3
    - 3rd position: 0 + 1 + 1 + 0 = 2
    - 4th position: 1 + 1 + 0 + 1 = 3
- After summing the bits, if the sum for a position is greater than 0, the resulting bit for that position is 1. If it's 0 or equal to 0, it's 0.
- So, the final SimHash for **Text 1** and **Text 2** will be:
  - Final SimHash (after comparing each sum with zero): 1111
  - Final SimHash (after comparing each sum with zero): 1111

Hashing in GPU

4. Comparing the Hashes:

Now, let's compare the SimHashes of **Text 1** and **Text 2**:

Text 1 SimHash: 1111

Text 2 SimHash: 1111

We can calculate the **Hamming distance** (number of differing bits) between the two SimHashes:

1111 vs. 1111: There are no differing bits, so the **Hamming distance** is **0**.

Conclusion:

- Since the **Hamming distance** is **0**, the two texts are exactly identical based on their SimHashes.
- Text 1 ("I love machine learning") and Text 2 ("I adore machine learning") are highly similar (only one word differs), but the SimHash method identifies them as similar because it focuses on structural similarities rather than exact matches.

Hashing in FPGA

- **Hashing in FPGA** involves implementing hash functions using hardware logic to achieve **high-speed and parallel computation**.
- FPGAs are commonly used for cryptographic hashing, database indexing, and networking applications where performance and low latency are critical.

Why FPGA for Hashing?

Compared to CPUs and GPUs, FPGAs offer the following advantages for hashing operations:

- **Parallel Processing:** Multiple hash computations can be performed simultaneously.
- **Low Latency:** Dedicated logic circuits minimize delay compared to software implementations.
- **Customizability:** Hash functions can be optimized for specific applications.
- **Energy Efficiency:** Consumes less power per hash computation compared to GPUs.
- **Scalability:** Can implement multiple hash pipelines for increased throughput.

Hashing in FPGA

Applications of Hashing in FPGA

1. **Cryptographic Hashing**
   - SHA-1, SHA-256, SHA-3, MD5, Blake2, etc.
   - Used in **blockchain mining**, **digital signatures**, **password hashing**.
2. **Network Packet Processing**
   - Hash-based packet filtering in routers/firewalls.
   - Used for **IP lookup**, **load balancing**, and **deep packet inspection** **(DPI)**.
3. **Database and Caching**
   - Hash tables for **fast key-value lookups**.
   - Used in **database indexing**, **deduplication**, and **content-addressable storage**.
4. **Pattern Matching & Machine Learning**
   - Approximate hashing for **image/video recognition**.
   - MinHash, SimHash for **near-duplicate detection**.
5. **Data Integrity Checking**
   - CRC (Cyclic Redundancy Check) and hash-based checksums.
   - Used in **storage systems**, **RAID controllers**, and **secure file transfers**.

Hashing in FPGA

Example: Implementing SHA-256 Hashing in FPGA

SHA-256 Overview

- SHA-256 is a cryptographic hash function that generates a **256-bit hash**.
- It processes input data in **512-bit blocks** through a **series of rounds**.

FPGA Implementation Strategy

1. **Data Input Buffer** → Load input message block into FPGA memory.
2. **Parallel Hash Pipelines** → Each SHA-256 round is mapped to a separate FPGA logic block.
3. **State Registers** → Maintain intermediate hash values across rounds.
4. **Final Hash Output** → Produce a 256-bit hashed value.

Use Case Example: FPGA in Blockchain Mining

- Bitcoin and Ethereum mining require high-speed **SHA-256 (Bitcoin)** or **Keccak (Ethereum)** hashing.
- FPGA miners are more efficient than CPUs but less powerful than GPUs/ASICs.
- FPGA-based **Proof of Work (PoW)** hashing systems balance speed and power efficiency.

Hashing in FPGA

Summary

- **FPGA-based hashing** is useful for cryptography, networking, and database acceleration.
- **Parallel pipelines** in FPGAs enable **faster hash computation** than CPUs.
- **Verilog/VHDL** implementations of SHA-256, CRC, and other hash functions are commonly used.
- **Optimization techniques** like **pipelining**, **loop unrolling**, and **parallel cores** improve performance.
- **FPGAs are a great choice** when **power efficiency and low-latency hashing** are required.

References:

1. **Hardware Accelerated Hashing:** A. A. Fairouz, M. Abusultan, V. V. Fedorov and S. P. Khatri, "Hardware Acceleration of Hash Operations in Modern Microprocessors," in *IEEE Transactions on Computers*, vol. 70, no. 9, pp. 1412-1426, 1 Sept. 2021, doi: 10.1109/TC.2020.3010855.
2. **Perfect Hashing**: Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. & Stein, C. (2001), *Introduction to Algorithms* , The MIT Press.
3. **Perfect Hashing**: https://www.cs.otago.ac.nz/cosc242/pdf/L11.pdf
4. **Cuckoo Hashing**: Mark Allen Weiss. 1995. Data structures and algorithm analysis (2nd ed.). Benjamin-Cummings Publishing Co., Inc., USA.
5. **Consistent Hashing**: https://web.stanford.edu/class/cs168/l/l1.pdf
6. Various resources Like books, Lecture slides from different universities, Web Links, AI tools etc.

*** End of Chapter 4 ***