Shepard - A Fast Exact Match Short Read Aligner

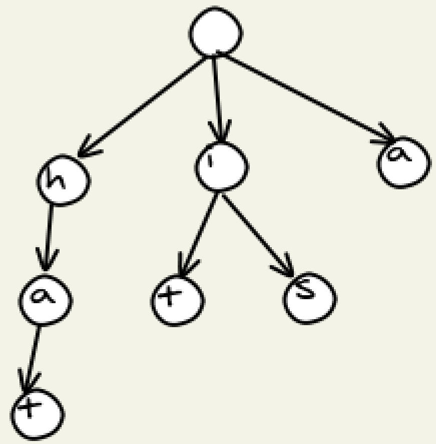
Chad Nelson, Kevin Townsend, Bhavani Rao, Phillip Jones, Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA, USA
(cnel711, ktown, bhavani, phjones, zambreno) @iastate.edu

Overview

- The Challenge
- What is Shepard?
  - Software for Minimal Perfect Hash Table Creation
  - FPGA hardware pipeline for fast Hash Table lookups
- How fast is it?
- What's next?

The Challenge

- Align millions of short DNA sequences to a reference genome.
- Current aligners (simplified):



- Our Solution:
  - One giant hash table

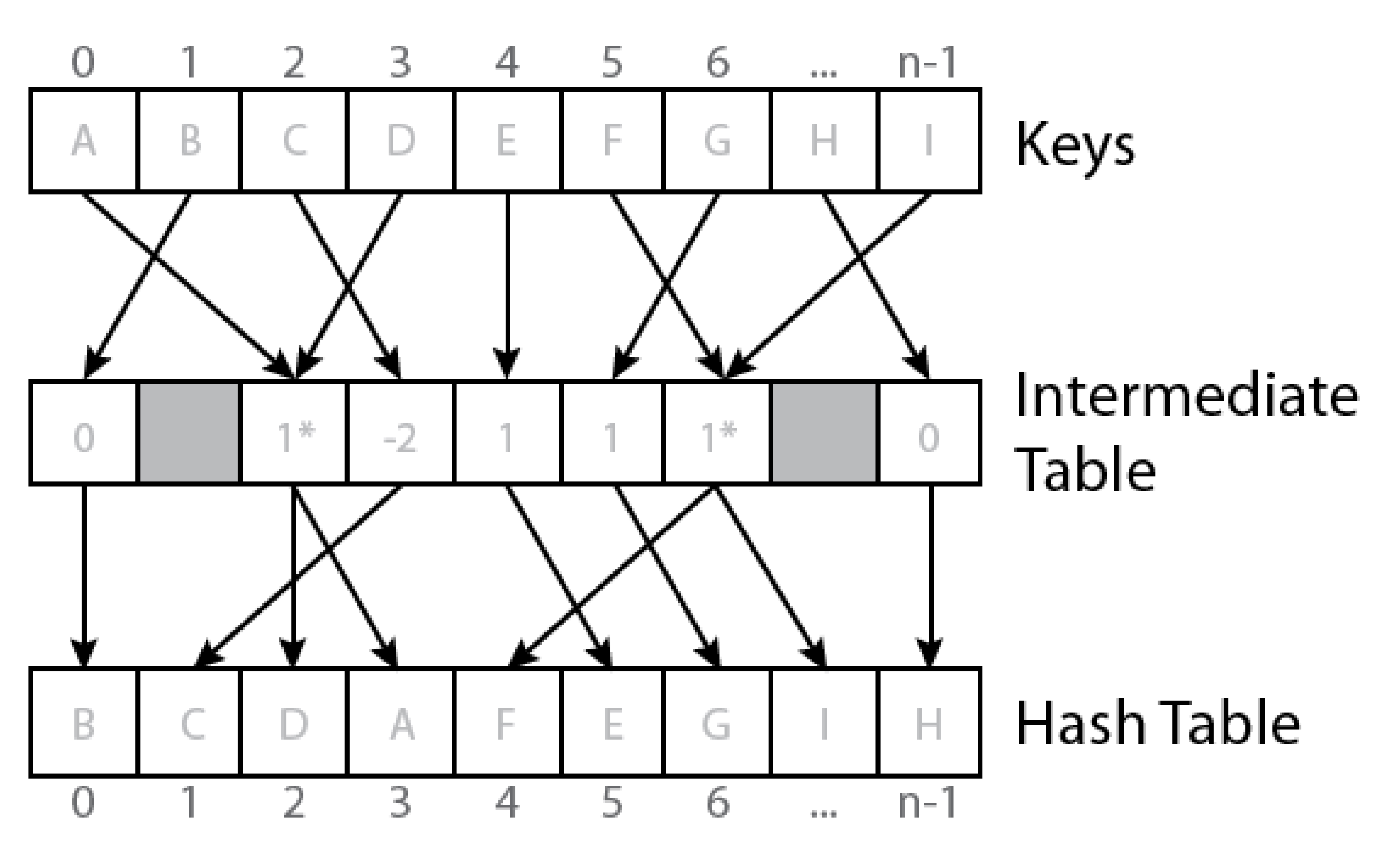
What is a MPH?

- MPH = Minimal Perfect Hash
- Given a **fixed set of keys**…
  - Perfect Hashes have no collisions
  - Minimal Perfect Hashes contain no empty buckets (they are memory efficient)
- For large sets of keys, you can create a general MPH in O(n) time.
  - General MPH algorithms require using an intermediate table.
  - **The intermediate table is the MPH.**

Minimal Perfect Hash Function Example

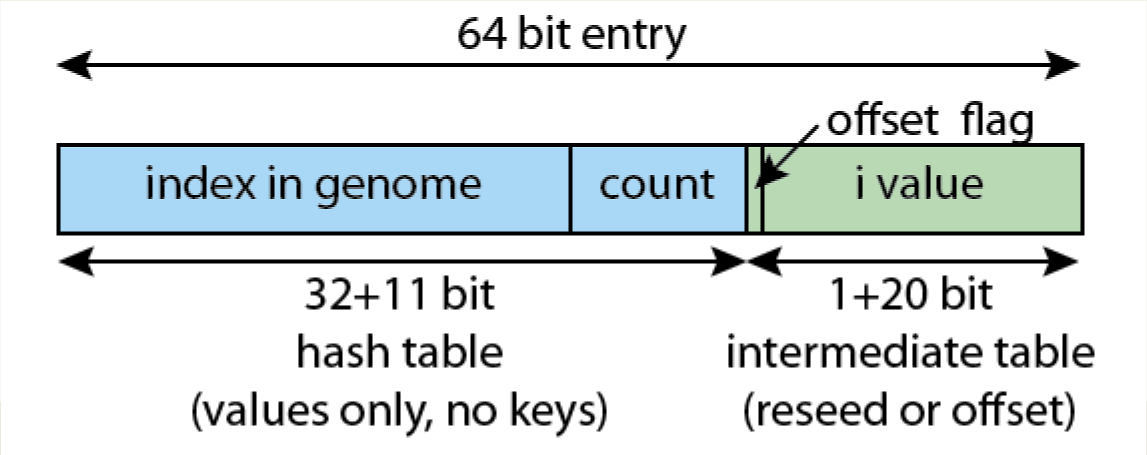
- Hash the key once to retrieve a few bits of information stored in the intermediate table
- Do one of the following:
  - Rehash with the new seed from the intermediate table (any buckets that had a collision would be rehashed*)
  - Add the offset from the intermediate table to the initial index.
- The MPH is created by choosing values for the intermediate table so that a given set of keys will not collide!

Minimal Perfect Hash Example



Shepard's Software

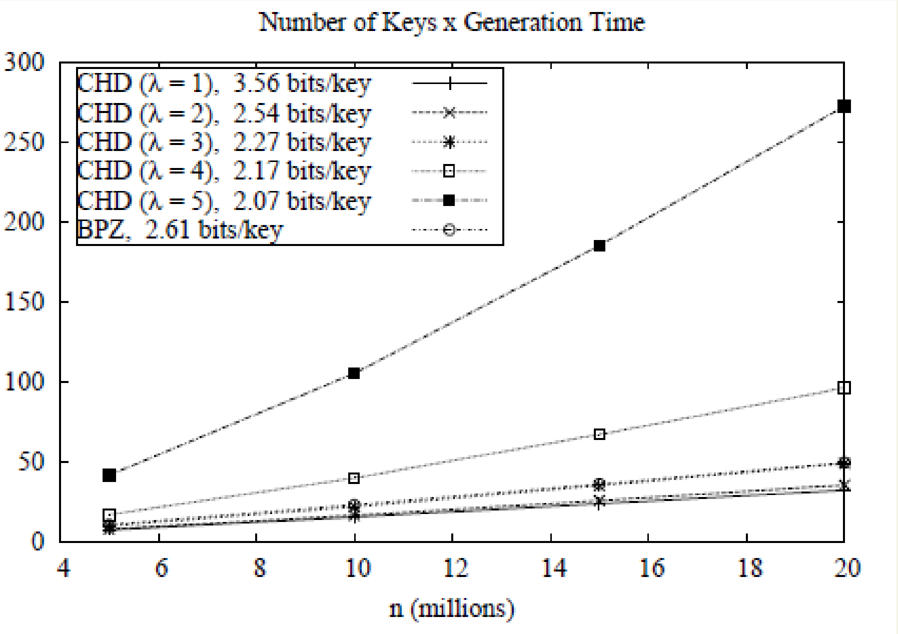
- Differences between our algorithm and other general purpose MPH algorithms (CHD)
  - CHD compresses the intermediate table down to between 2 and 4 bits per entry
  - Shepard does not compress (no need); if entries are 8 byte aligned, there are 20 bits per entry available for the intermediate table



Speed of Shepard's Hash Table Construction

- Current MPH creation using C code (CHD)
  - 800,000 entries / sec
    *only 20 million entries

- Shepard MPH C code
  - 300,000 entries / sec
    *using 2.8 billion entries
  - For 2.8 billion entries, takes ~2.5 hours



(d) $\alpha = 1.00$, Space lower bound $= 1.44$.

Graph from: Hash, displace, and compress
Djamal Belazzougui et al

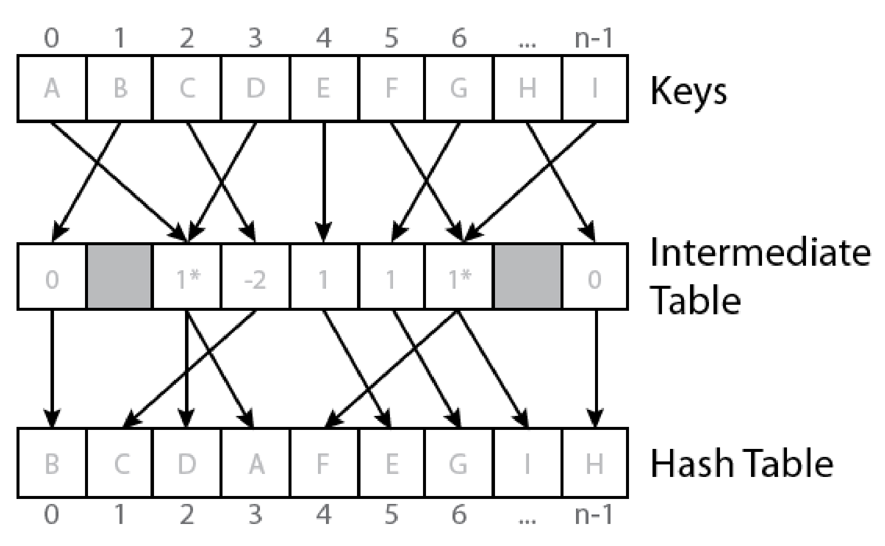
Future Work

- Hardware Pipeline for Hash Table Creation
  - The 6 stages for hash table creation are simple for-loops
  - Assuming:
    - ~40 memory operations per entry in construction
    - Application is memory bound
    - Convey HC-1 can perform 10 billion memory ops / sec
  - We could create the hash table in **about 12 seconds**, more than 250x faster than CHD
  - Concurrency Issues

Software Implementation

Algorithm 1 Pseudocode of the Shepard Pipeline

procedure ALIGN($reads, genome, hashtable, results$)
    for ($i = 0; i < length(reads); i++$) **do**
        $r \leftarrow reads[i]$; ▷ Stage 1
        $h \leftarrow hash(r, seed = 0)$; ▷ Stage 2
        $ivalue \leftarrow intermediateTable[h]$;
        if ($ivalue$ is an offset) **then** ▷ Stage 3
            $index \leftarrow hash(r, seed = 0) + ivalue$;
        else
            $index \leftarrow hash(r, seed = ivalue)$;
        end if
        $entry \leftarrow hashtable[index]$;
        $check \leftarrow genome[entry.index]$; ▷ Stage 4
        if ($r == check$) **then** ▷ Stage 5
            $results[i] \leftarrow entry$;
        else
            $results[i] \leftarrow NULL$;
        end if
    end for
end procedure



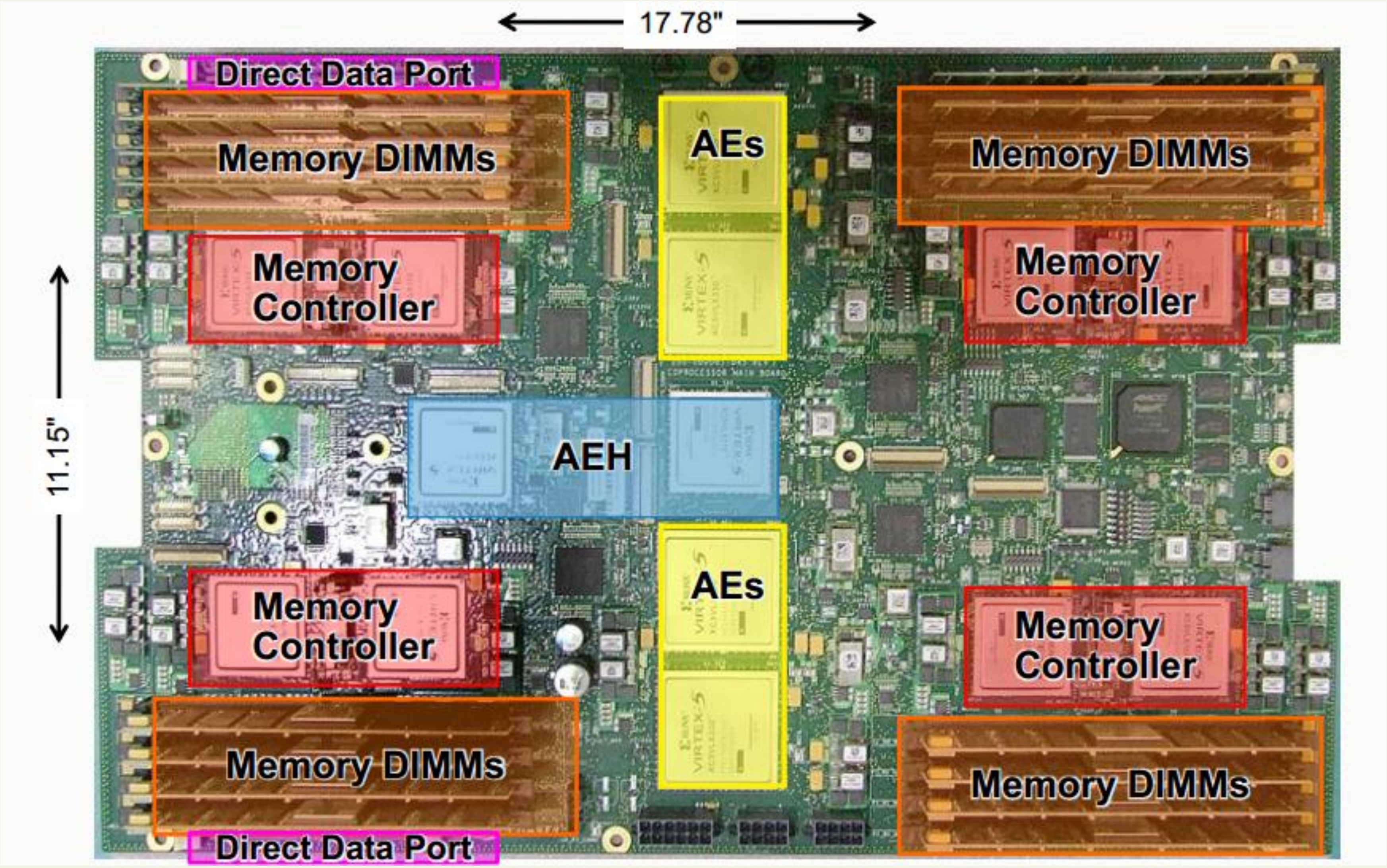
Software Implementation

- Speed
  - Single threaded:
    - 1,000,000 reads / second (using total program execution time)
  - Performance increases when using multiple threads

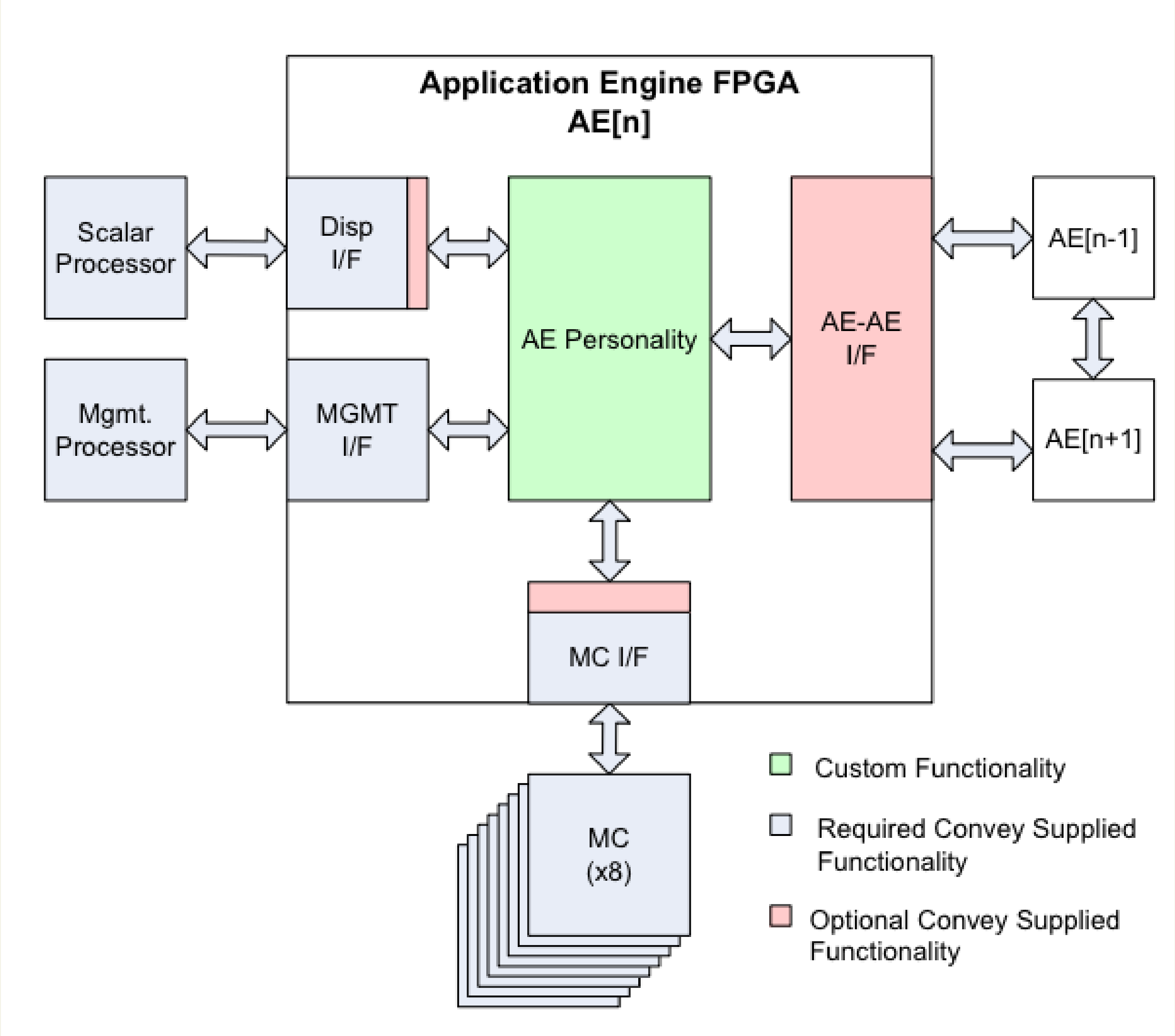
PERFORMANCE COMPARISON OF POPULAR SHORT READ ALIGNERS.

Tool	Platform	Speed (reads/s)	% Aligned	Memory (GB)
MAQ [4]	CPU	50	93.2	1.2
SOAP	CPU	70	93.8	14.7
SOAP2 [5]	CPU	2,000	93.6	5.4
Bowtie [6]	CPU	2,500	91.7	2.3
SOAP3 [9]	GPU	6,000,000	96.8	3.2

The Convey HC-1



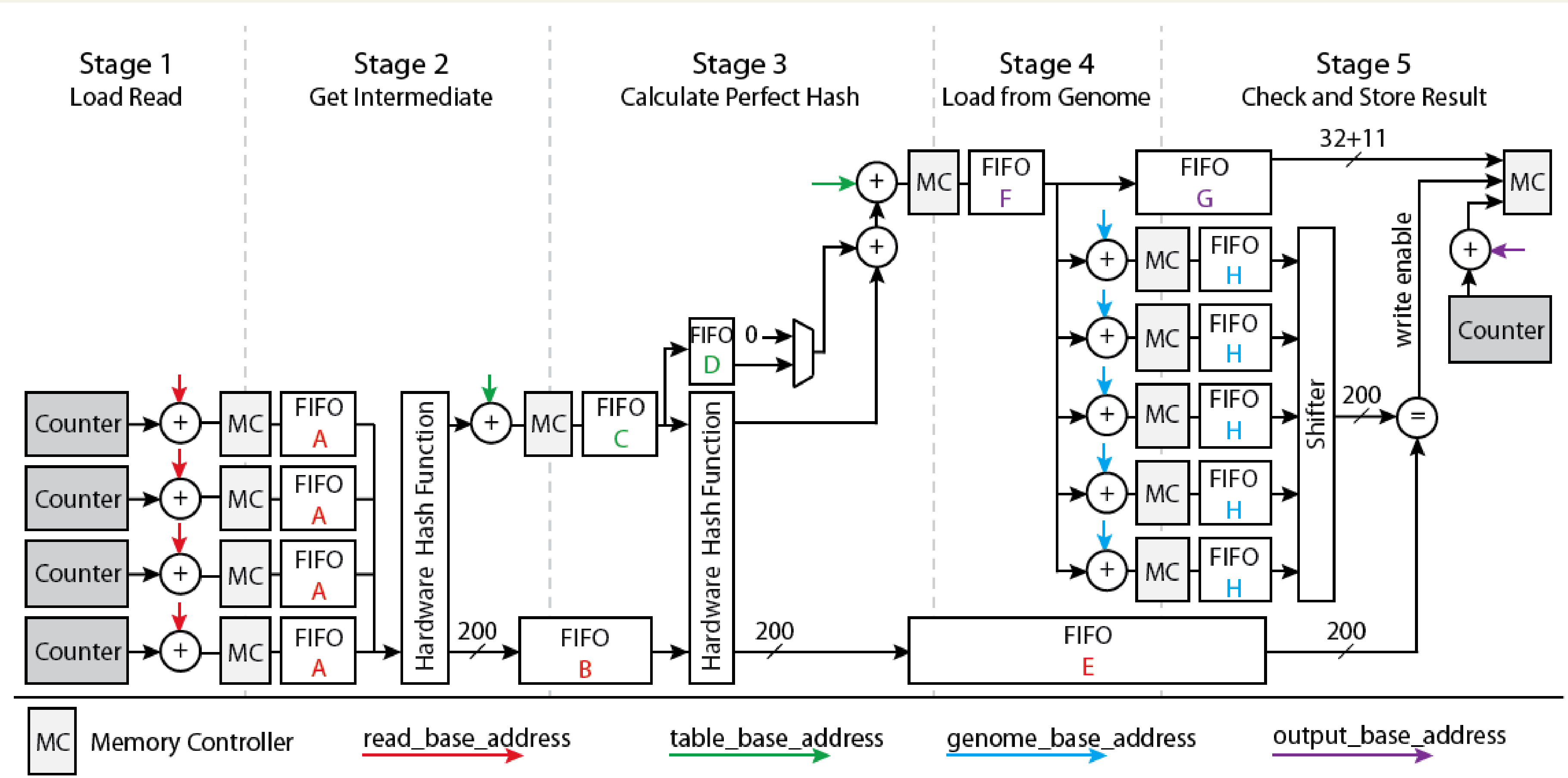
The Convey HC-1



Convey Custom AE Development Process

- It's simple!
- Take your original program, pull out a kernel written in C/C++
- Turn the kernel into a CAE instruction (define ISA)
- Write a software simulator of the program
  - Emulates custom hardware
  - Validates memory accesses
  - Validates AEG registers
- Write hardware description in Verilog or VHDL
  - Use the software simulator to test your design (quick)
  - Build a bitfile (slow)

Hardware Implementation

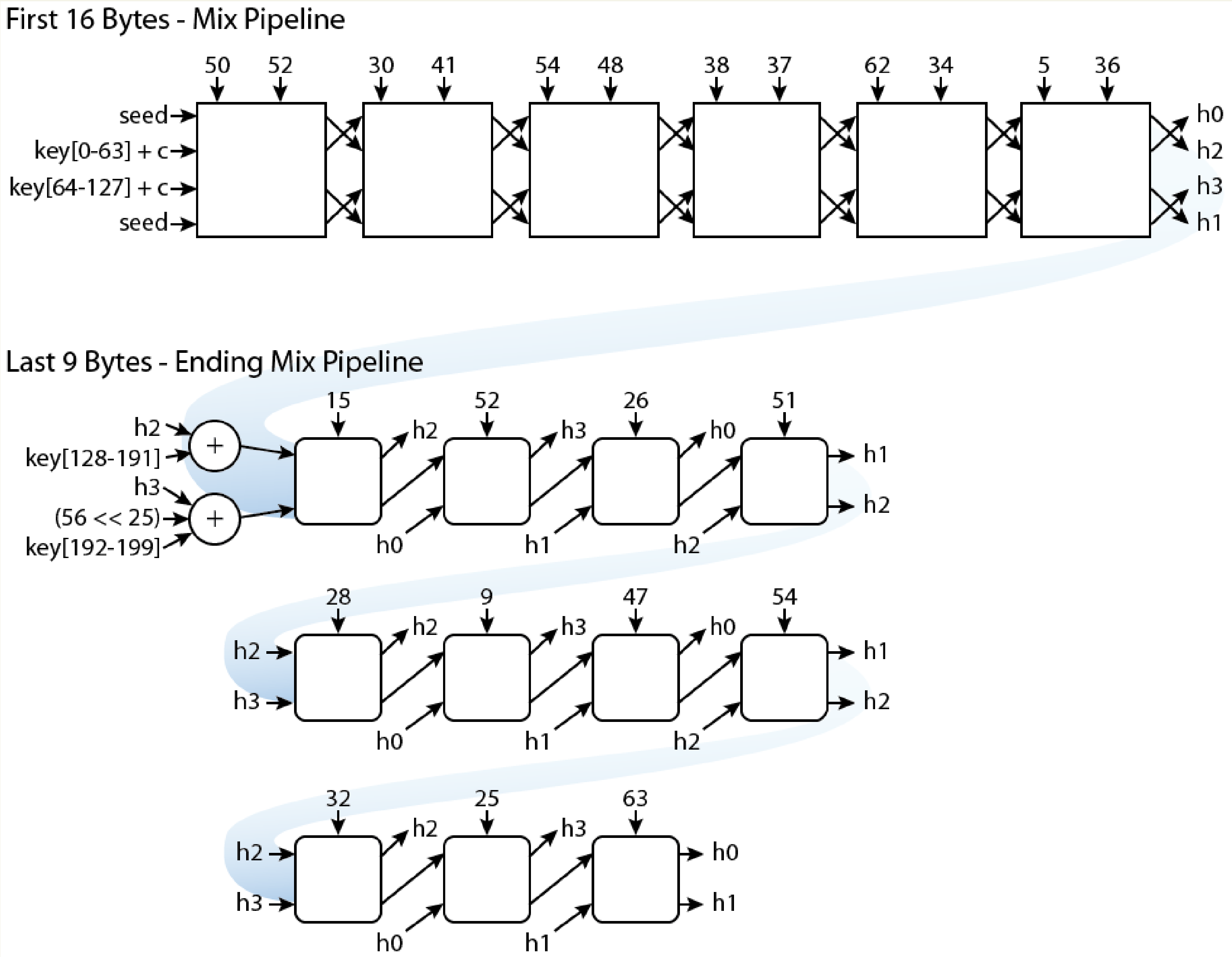


Hash Function Pipeline

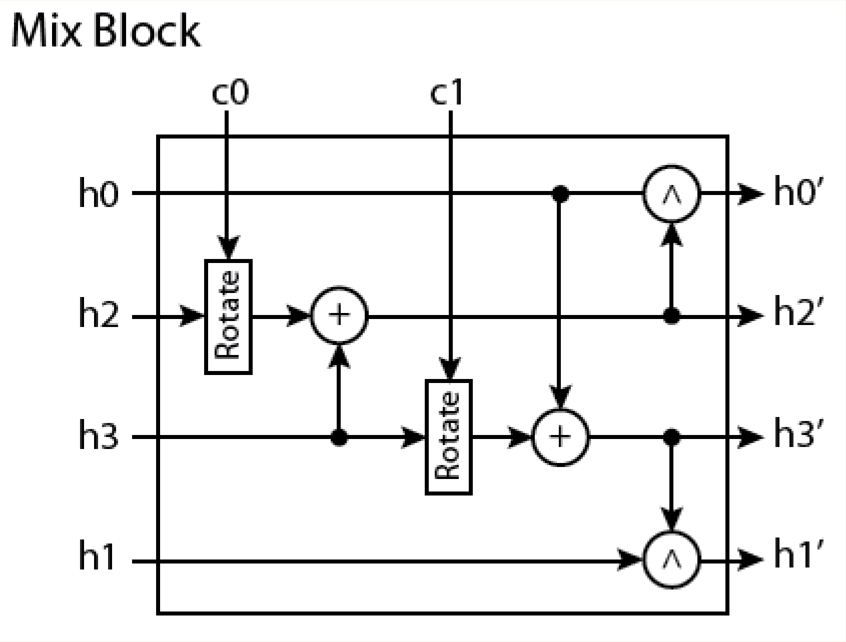
- Jenkin’s Spooky Hash
- Fixed the pipeline specifically for 100 base pair reads (25 bytes)

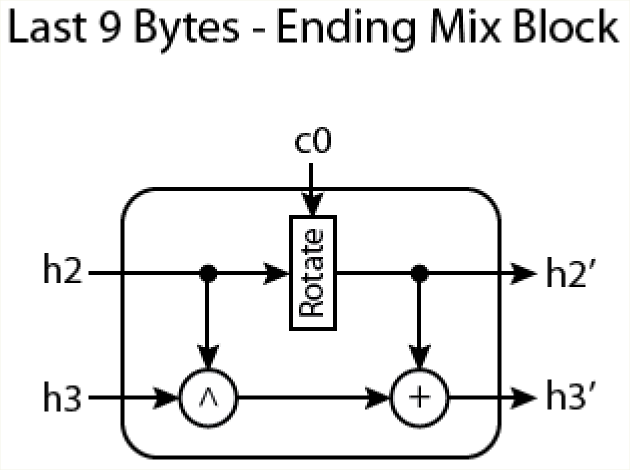
- Hash function consists of rotations, additions, and XOR

Hash Function Pipeline Implementation



Hash Function Pipeline Implementation





Speed

- 350,000,000 reads per second

PERFORMANCE COMPARISON OF POPULAR SHORT READ ALIGNERS.

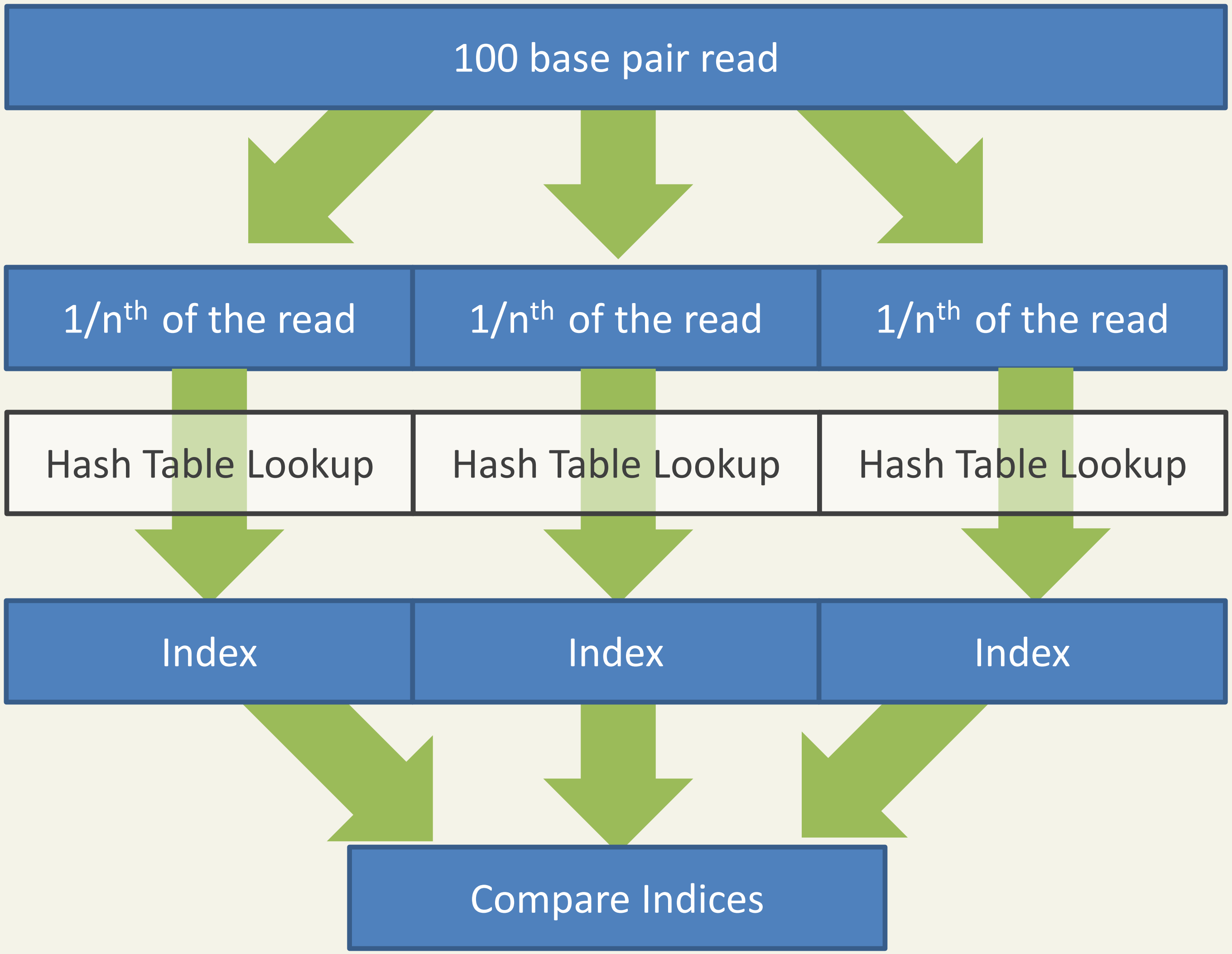
Tool	Platform	Speed (reads/s)	% Aligned	Memory (GB)
MAQ [4]	CPU	50	93.2	1.2
SOAP	CPU	70	93.8	14.7
SOAP2 [5]	CPU	2,000	93.6	5.4
Bowtie [6]	CPU	2,500	91.7	2.3
SOAP3 [9]	GPU	6,000,000	96.8	3.2
Shepard	FPGA	350,000,000	25.2	23.3

- Exact matches only!
- *% aligned* depends on the quality of the reference genome and the read data
- Had we used better read data (such as the data used in the SOAP3 paper), the *% aligned* would be as high as 60.3%

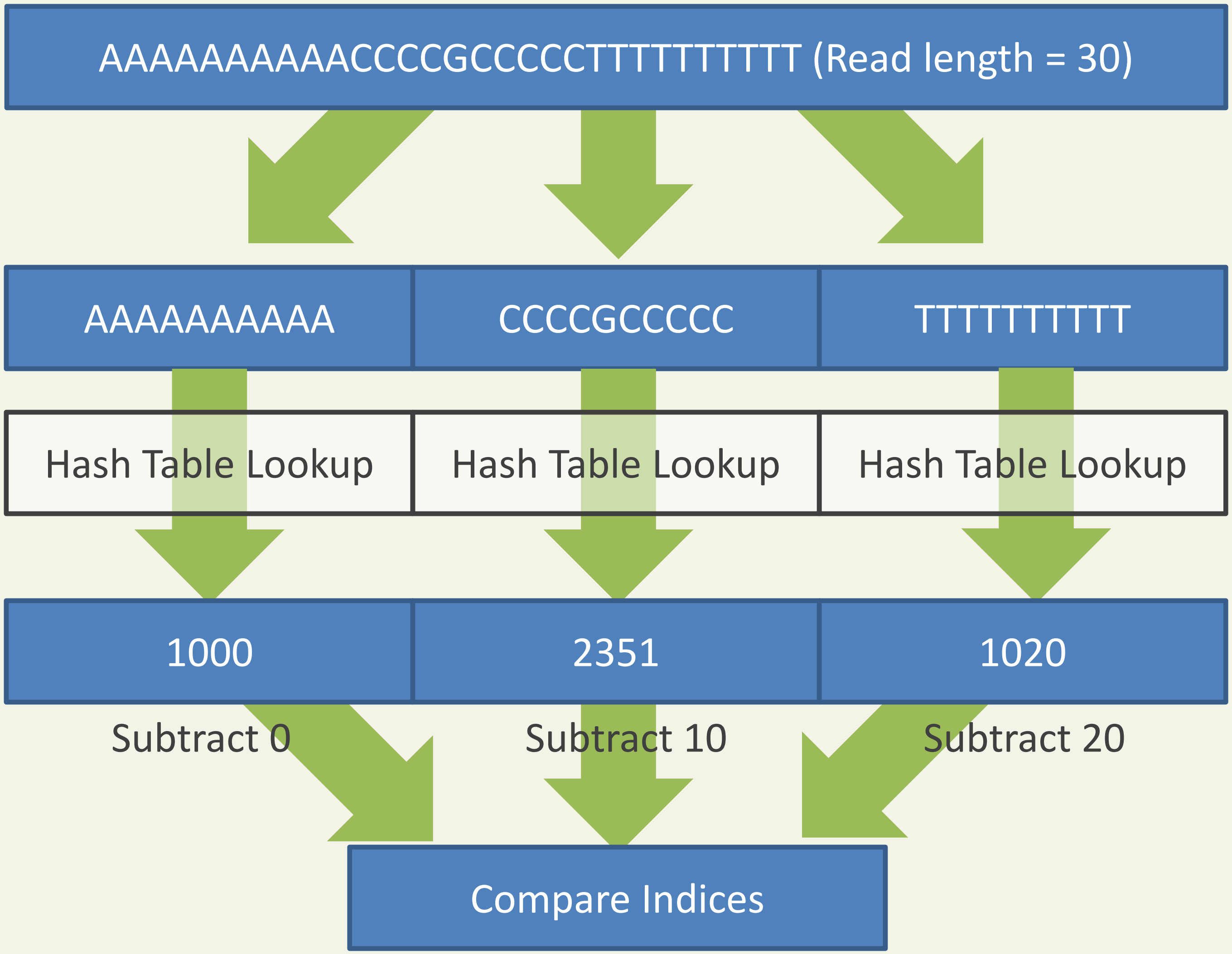
Future Work

- Let's update the pipeline to allow mismatches!
- The idea:
  - Split the read into **n** parts. Use a hash table to lookup the index of these parts.
  - Compare the indices of the parts. If any of the indices match, we can compare the read to the genome to see how many mismatches occurred!
- The time cost lies in the extra 2***n** memory operations that must be performed.
  - Our original design only used 12 of the 16 memory controllers, so if n=3, we would incur no time penalty!
  - We get a "free lunch" by using all of the available resources.

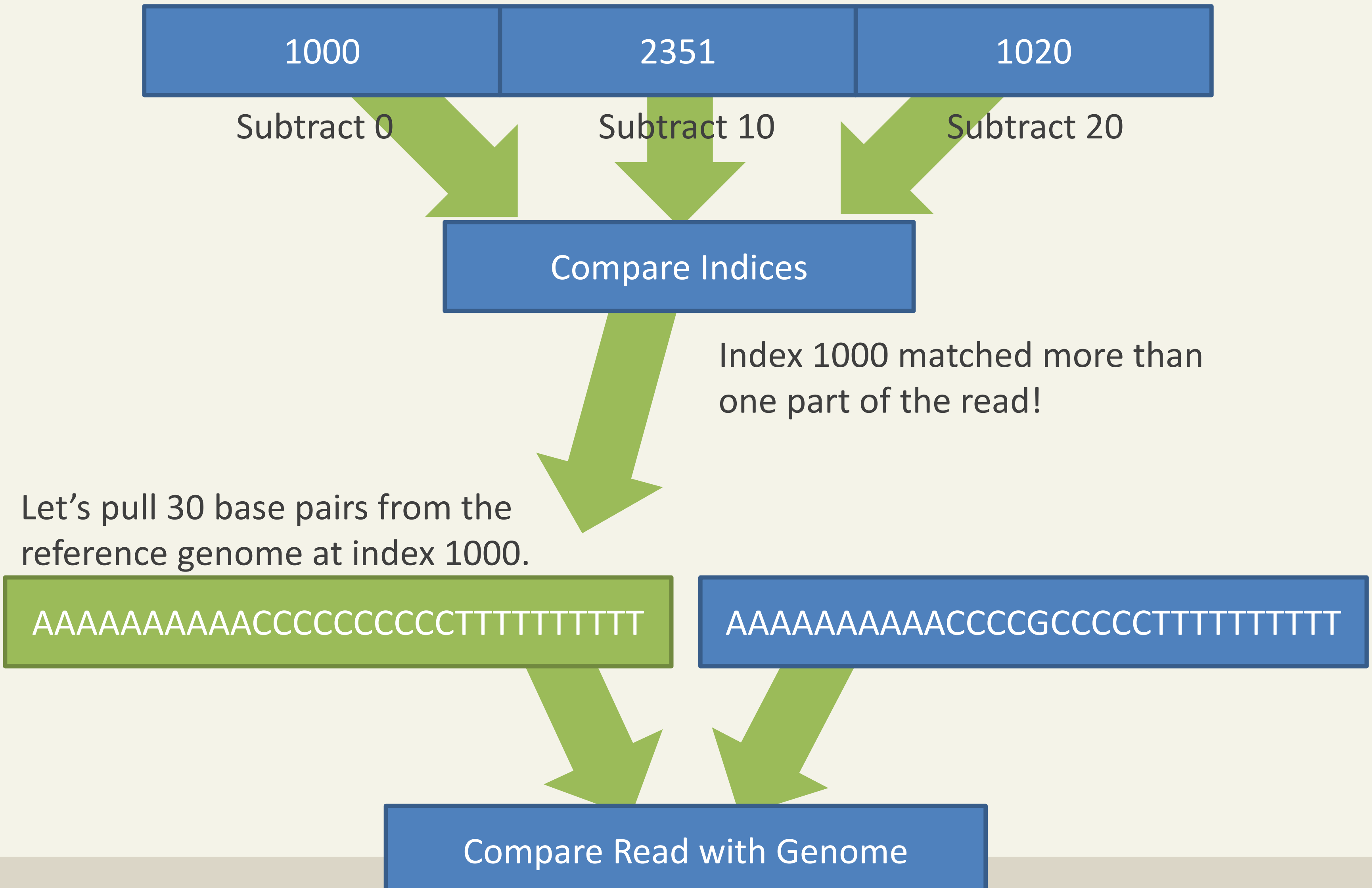
Future Work


100 base pair read
1/nth of the read
1/nth of the read
1/nth of the read
Hash Table Lookup
Hash Table Lookup
Hash Table Lookup
Index
Index
Index
Compare Indices

Future Work Example



Future Work Example



Future Work Example

- The comparison of the genome and a read is fairly simple in hardware for mismatches:
  - XOR the genome with the read
  - Count the number of 1's. If all 0's, it's an exact match.

- It becomes slightly more complex for insertions/deletions, but the same approach can be taken for comparison

Future Work Risks

- What affect will the presence of duplicates have on the alignment %?
  - At 100 base pairs, 2.36% of the genome is a duplicate of itself.
  - At 36 base pairs, 11% of the genome is a duplicate of itself.

- Let’s test in software and find out!

Conclusion (1/2)

- We implemented our own software to create MPH table:
  - Our Speed: 300,000 entries / sec
  - Other software (CHD): 800,000 entries / sec

- Future Work
  - Increase the speed to 250,000,000 entries / sec
  - This would allow us to make the pre-processing step part of the alignment process!
  - Instead of using a generic human reference genome, people may be able to use the DNA sequence of a blood family member as a reference in order to increase the percentage of exact matches.
  - The multi-port cache is useful IP for future projects utilizing the Convey HC-1 (allows atomic read-write)

Conclusion (2/2)

- We implemented our own hardware for read alignment:
  - Speed: 350,000,000 reads / sec
  - Alignment: 25%
  - About 60x speedup over SOAP3 (GPU)
  - Over 100,000x speedup over Bowtie (CPU)
  - Con: exact match only

- Future Work
  - Increase the alignment percentage by gaining the ability to detect mismatches, insertions, and deletions using the hashing approach.
  - This would make the project applicable to real-world sequence alignment. We can alter the pipeline without loosing too much speed.

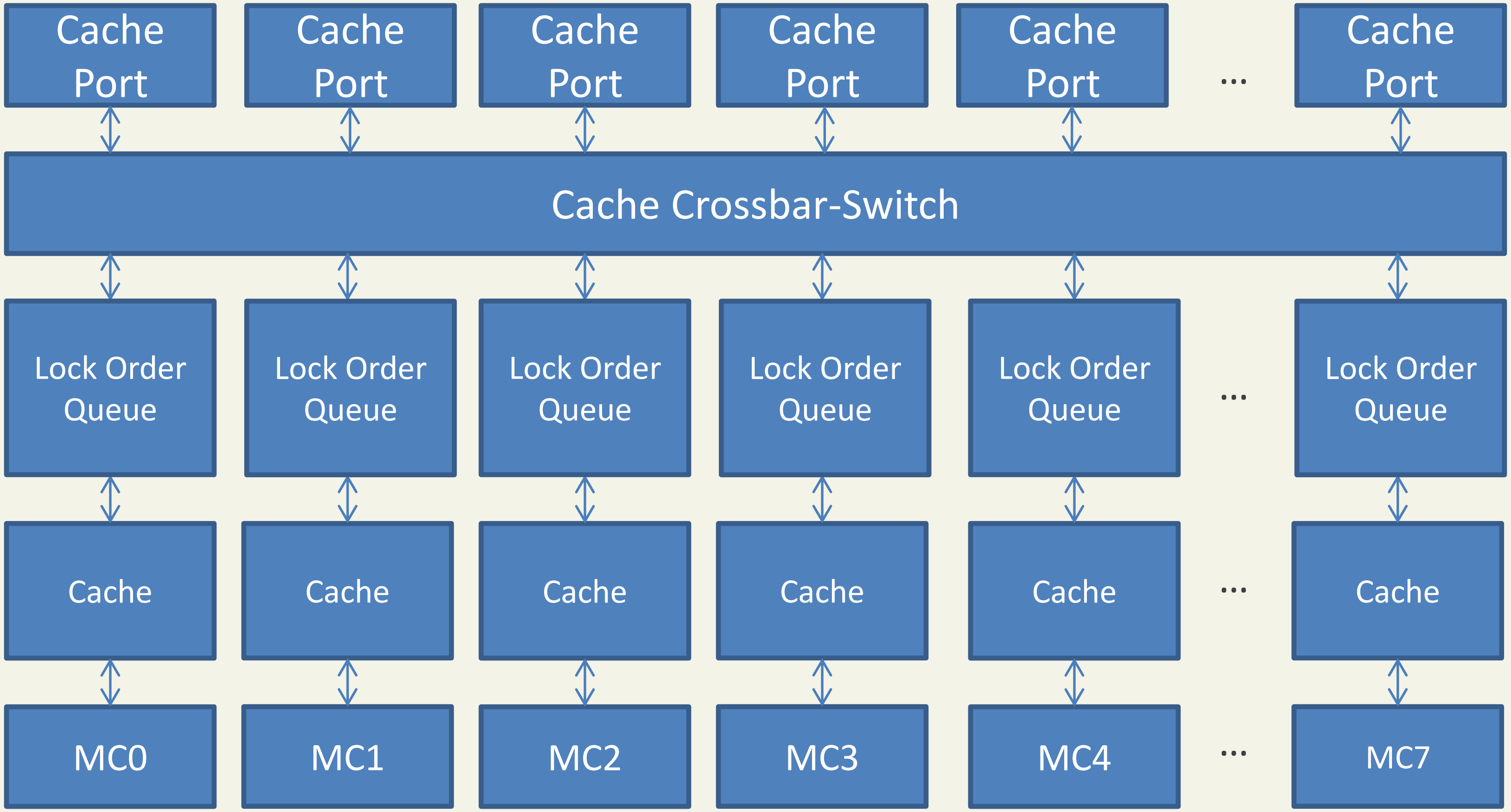
Questions

Future Work

Future Work - Concurrency Solution

- Divide the hash table and keys in four, one for each of the four FPGAs (AEs) on the Convey HC-1
  - Use the first two bits of the key to separate the hash table
  - Need to keep track of the size of each hash table, though they should be roughly the same size
  - This gets rid of AE to AE concurrency issues by separating the problem
- For stages 2, 4, and 5, implement a cache on each AE for handling the rare case of a concurrency issues.

Future Work - The Cache



Future Work - The Cache

- 16 Cache ports
- Use the Write Complete interface
- No crossbar, no read order queue, no strong order queue
- Write back on replace
- The cache would need the ability to Lock certain cache lines to a given Cache Port. Store the lock with the cache line.

- Lock Order Queue
  - If a request for a memory address comes in, and the cache is locked, the request waits until it is unlocked

- The cache will provide an **infrastructure** that we can use more generally in other projects utilizing the Convey HC-1. It allows strongly-ordered memory operations across multiple Memory Controllers.

Construction Algorithm Visualized!

Algorithm

- Stage 1 (create unique)
  – Create a list of all N of your <key, value> pairs to be added to the hash table
- Stage 2
  – Hash every key into a bucket to see which spaces in the hash table will have collisions
  – This amounts to having an array of size N initialized to all zeros. Then, you hash every key and increment the value at that index.
- Stage 2a
  – Count the number of buckets of each size. Since the max bucket size is small (< 20), this can be done during Stage 2.

Algorithm (Continued)

- Stage 3 (sorting)
  - Sort the keys by bucket size (largest to smallest)
  - Using the collision count values from stage 2 and the size of each bucket, this can be done in O(N) time.
- Stage 4 (reseed big buckets)
  - For each bucket size >= 2, starting with the largest, reseed the bucket
  - This involves initializing an bit array with all 0's.
  - For each bucket (contains 2 or more keys), rehash all the keys from the bucket with a new seed. Check the bit array that all keys have a spot in the final array
  - Store the new seed in the intermediate table.

Algorithm (Continued)

- Stage 5 (displace singular buckets)
  - For each bucket size = 1, find the next available empty space in the bit array
  - Place the key in this position, recording the offset from the original location in the intermediate table
- Stage 6 (add values to hash table)
  - The intermediate table is your a MPHF
  - Simply place the values in the final output table.

Example

- We are going to create a MPH Table for the following keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar

Stage 2

- Keys:
- Hash(seed=0)

Bucket Size	Number
0	10
1	0
2	0

Counts
0
0
0
0
0
0
0
0
0
0

Stage 2

- Keys:
  - Armadillo
- Hash(seed=0)
  - 1

Bucket Size	Number
0	9
1	1
2	0

Counts
0
1
0
0
0
0
0
0
0
0

Stage 2

- Keys:
  - Armadillo
  - Bird
- Hash(seed=0)
  - 1
  - 5

Bucket Size	Number
0	8
1	2
2	0

Counts
0
1
0
0
0
1
0
0
0
0

Stage 2

- Keys:
  - Armadillo
  - Bird
  - Cat

- Hash(seed=0)
  - 1
  - 5
  - 4

Bucket Size	Number
0	7
1	3
2	0

Counts
0
1
0
0
1
1
0
0
0
0

Stage 2

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
- Hash(seed=0)
  - 1
  - 5
  - 4
  - 8

Bucket Size	Number
0	6
1	4
2	0

Counts
0
1
0
0
1
1
0
0
1
0

Stage 2

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
- Hash(seed=0)
  - 1
  - 5
  - 4
  - 8
  - 3

Bucket Size	Number
0	5
1	5
2	0

Counts
0
1
0
1
1
1
0
0
1
0

Stage 2

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
- Hash(seed=0)
  - 1
  - 5
  - 4
  - 8
  - 3
  - 1

Bucket Size	Number
0	5
1	4
2	1

Counts
0
2
0
1
1
1
0
0
1
0

Stage 2

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage

- Hash(seed=0)
  - 1
  - 5
  - 4
  - 8
  - 3
  - 1
  - 9

Bucket Size	Number
0	4
1	5
2	1

Counts
0
2
0
1
1
1
0
0
1
1

Stage 2

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse

- Hash(seed=0)
  - 1
  - 5
  - 4
  - 8
  - 3
  - 1
  - 9
  - 5

Bucket Size	Number
0	4
1	4
2	2

Counts
0
2
0
1
1
2
0
0
1
1

Stage 2

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana

- Hash(seed=0)
  - 1
  - 5
  - 4
  - 8
  - 3
  - 1
  - 9
  - 5
  - 0

Bucket Size	Number
0	3
1	5
2	2

Counts
1
2
0
1
1
2
0
0
1
1

Stage 2

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar
- Hash(seed=0)
  - 1
  - 5
  - 4
  - 8
  - 3
  - 1
  - 9
  - 5
  - 0
  - 7

Bucket Size	Number
0	2
1	6
2	2

Counts
1
2
0
1
1
2
0
1
1
1

Stage 2

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar

- Hash(seed=0)
  - 1
  - 5
  - 4
  - 8
  - 3
  - 1
  - 9
  - 5
  - 0
  - 7

Bucket Size	Number
0	2
1	6
2	2

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar

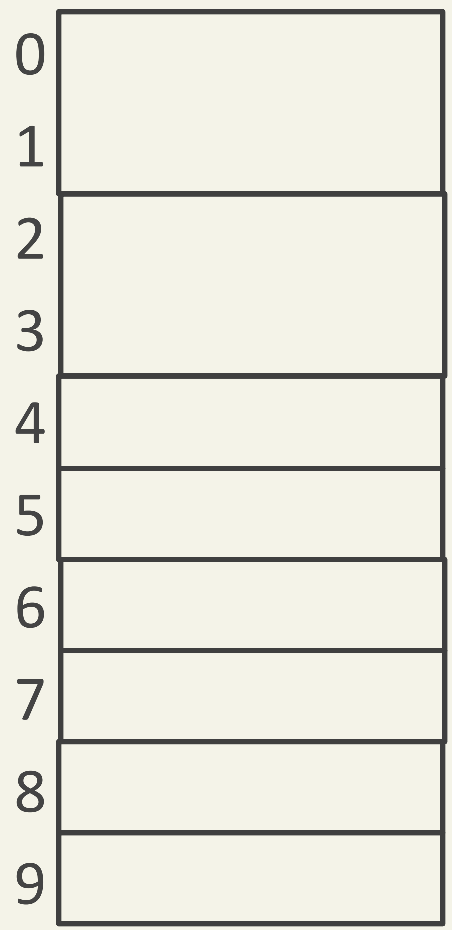
Bucket Size	Number
0	2
1	6
2	2

New Index
-
-
-
-
-
-
-
-
-
-

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar



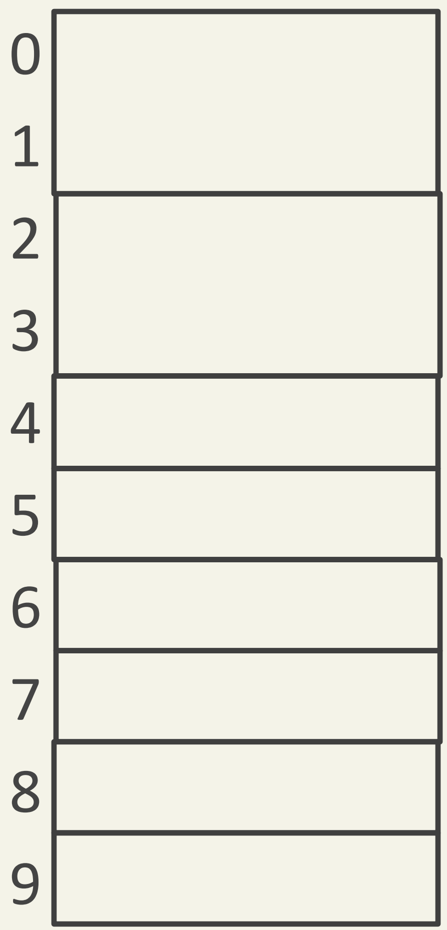
Bucket Size	Number
0	2
1	6
2	2

New Index
4
-
-
-
-
-
-
-
-
-

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar



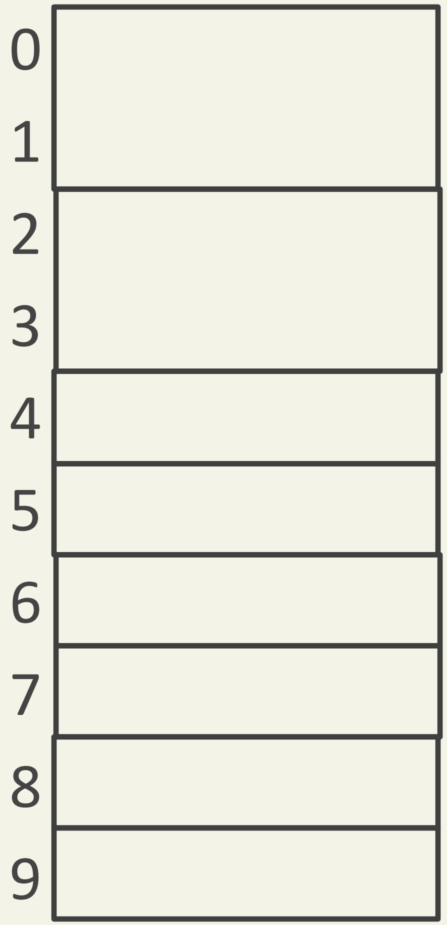
Bucket Size	Number
0	2
1	6
2	2

New Index
4
0
-
-
-
-
-
-
-
-

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar



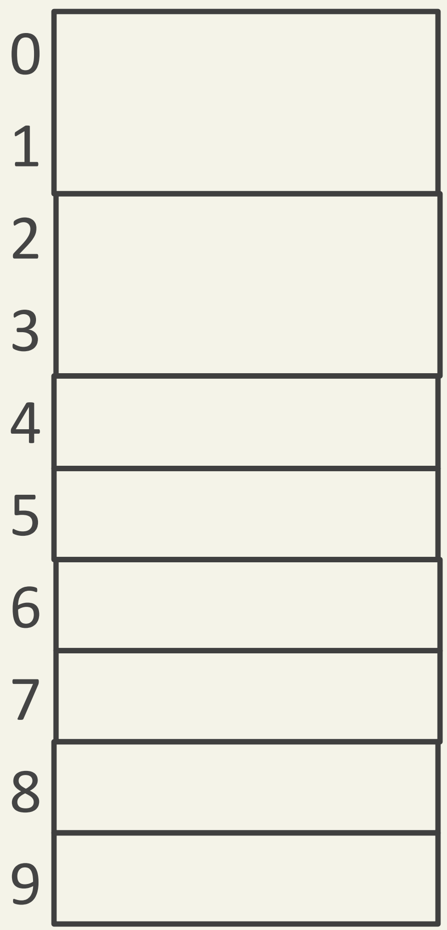
Bucket Size	Number
0	2
1	6
2	2

New Index
4
0
-
5
-
-
-
-
-
-

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar



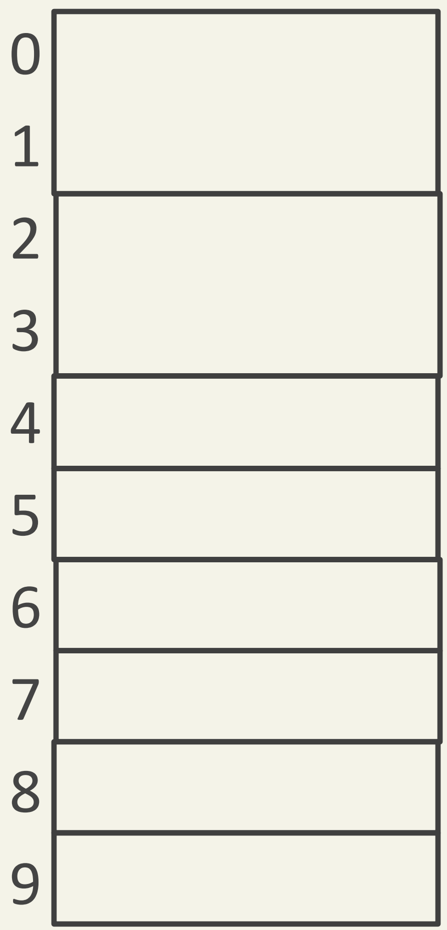
Bucket Size	Number
0	2
1	6
2	2

New Index
4
0
-
5
6
-
-
-
-
-

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar



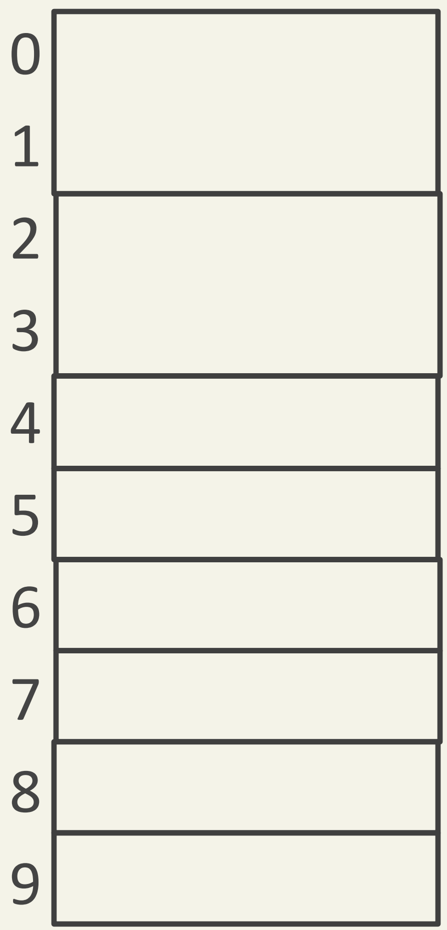
Bucket Size	Number
0	2
1	6
2	2

New Index
4
0
-
5
6
2
-
-
-
-

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar



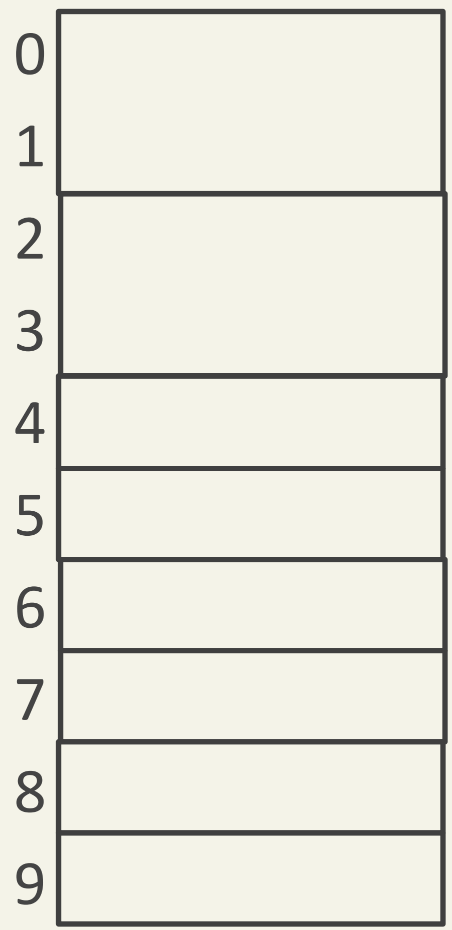
Bucket Size	Number
0	2
1	6
2	2

New Index
4
0
-
5
6
2
-
7
-
-

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar



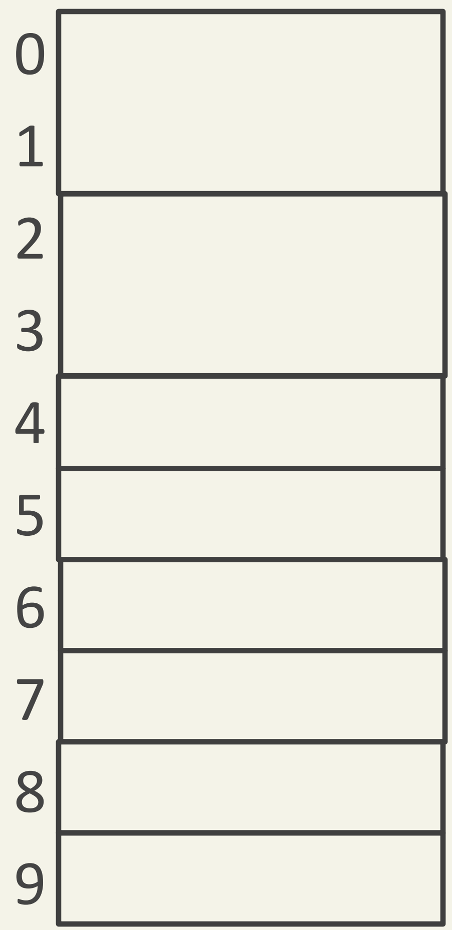
Bucket Size	Number
0	2
1	6
2	2

New Index
4
0
-
5
6
2
-
7
8
-

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - Armadillo
  - Bird
  - Cat
  - Dog
  - Elephant
  - Frog
  - Garage
  - Horse
  - Iguana
  - Jaguar



Bucket Size	Number
0	2
1	6
2	2

New Index
4
0
-
5
6
2
-
7
8
9

Counts
1
2
0
1
1
2
0
1
1
1

Stage 3

- Keys:
  - 1– Armadillo
  - 5– Bird
  - 4– Cat
  - 8– Dog
  - 3– Elephant
  - 1– Frog
  - 9– Garage
  - 5– Horse
  - 0– Iguana
  - 7– Jaguar

- Sorted Keys:
  - Armadillo
  - Frog
  - Bird
  - Horse
  - Iguana
  - Elephant
  - Cat
  - Jaguar
  - Dog
  - Garage

Bucket Size	Number
0	2
1	6
2	2

New Index	Counts
4	1
0	2
-	0
5	1
6	1
2	2
-	0
7	1
8	1
9	1

Stage 4

- From largest to smallest
  - Armadillo – 1 -> 2
  - Frog – 1 -> 7
  - Bird – 5
  - Horse – 5
  - Iguana – 0
  - Elephant – 3
  - Cat – 4
  - Jaguar – 7
  - Dog – 8
  - Garage – 9

Intermediate Table
0
0
0
0
0
0
0
0
0
0

Bit Array
0
0
0
0
0
0
0
0
0
0

Stage 4

- From largest to smallest
  - Armadillo – 1 -> 2
  - Frog – 1 -> 7
  - Bird – 5
  - Horse – 5
  - Iguana – 0
  - Elephant – 3
  - Cat – 4
  - Jaguar – 7
  - Dog – 8
  - Garage – 9

Intermediate Table
0
reseed = 1
0
0
0
0
0
0
0
0

Bit Array
0
0
1
0
0
0
0
1
0
0

Stage 4

- From largest to smallest
  - Armadillo – 1 -> 2
  - Frog – 1 -> 7
  - Bird – 5 -> 1
  - Horse – 5 -> 9
  - Iguana – 0
  - Elephant – 3
  - Cat – 4
  - Jaguar – 7
  - Dog – 8
  - Garage – 9

Intermediate Table
0
reseed = 1
0
0
0
reseed = 1
0
0
0
0

Bit Array
0
1
1
0
0
0
0
1
0
1

Stage 5

- From largest to smallest
  - Armadillo – 1 -> 2
  - Frog – 1 -> 7
  - Bird – 5 -> 1
  - Horse – 5 -> 9
  - Iguana – 0 -> 0
  - Elephant – 3
  - Cat – 4
  - Jaguar – 7
  - Dog – 8
  - Garage – 9

Intermediate Table
offset = 0
reseed = 1
0
0
0
reseed = 1
0
0
0
0

Bit Array
1
1
1
0
0
0
0
1
0
1

Stage 5

- From largest to smallest
  - Armadillo – 1 -> 2
  - Frog – 1 -> 7
  - Bird – 5 -> 1
  - Horse – 5 -> 9
  - Iguana – 0 -> 0
  - Elephant – 3 -> 0
  - Cat – 4
  - Jaguar – 7
  - Dog – 8
  - Garage – 9

Intermediate Table
offset = 0
reseed = 1
0
offset = 0
0
reseed = 1
0
0
0
0

Bit Array
1
1
1
1
0
0
0
1
0
1

Stage 5

- From largest to smallest
  - Armadillo — 1 -> 2
  - Frog — 1 -> 7
  - Bird — 5 -> 1
  - Horse — 5 -> 9
  - Iguana — 0 -> 0
  - Elephant — 3 -> 0
  - Cat — 4 -> 0
  - Jaguar — 7
  - Dog — 8
  - Garage — 9

Intermediate Table
offset = 0
reseed = 1
0
offset = 0
offset = 0
reseed = 1
0
0
0
0

Bit Array
1
1
1
1
1
0
0
1
0
1

Stage 5

- From largest to smallest
  - Armadillo – 1 -> 2
  - Frog – 1 -> 7
  - Bird – 5 -> 1
  - Horse – 5 -> 9
  - Iguana – 0 -> 0
  - Elephant – 3 -> 0
  - Cat – 4 -> 0
  - Jaguar – 7 -> 5
  - Dog – 8
  - Garage – 9

Intermediate Table
offset = 0
reseed = 1
0
offset = 0
offset = 0
reseed = 1
0
offset = -2
0
0

Bit Array
1
1
1
1
1
1
0
1
0
1

Stage 5

- From largest to smallest
  - Armadillo – 1 -> 2
  - Frog – 1 -> 7
  - Bird – 5 -> 1
  - Horse – 5 -> 9
  - Iguana – 0 -> 0
  - Elephant – 3 -> 0
  - Cat – 4 -> 0
  - Jaguar – 7 -> 5
  - Dog – 8 -> 6
  - Garage – 9

Intermediate Table
offset = 0
reseed = 1
0
offset = 0
offset = 0
reseed = 1
0
offset = -2
offset = -2
0

Bit Array
1
1
1
1
1
1
1
1
0
1

Stage 5

- From largest to smallest
  - Armadillo – 1 -> 2
  - Frog – 1 -> 7
  - Bird – 5 -> 1
  - Horse – 5 -> 9
  - Iguana – 0 -> 0
  - Elephant – 3 -> 0
  - Cat – 4 -> 0
  - Jaguar – 7 -> 5
  - Dog – 8 -> 6
  - Garage – 9 -> 8

Intermediate Table
offset = 0
reseed = 1
0
offset = 0
offset = 0
reseed = 1
0
offset = -2
offset = -2
offset = -1

Bit Array
1
1
1
1
1
1
1
1
1
1