COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Vectorized Bloom Filters for Advanced SIMD Processors

Orestis Polychroniou
Kenneth A. Ross

Bloom filters

- Introduction
  - Original version [Bloom 1970]
    - Represents a “set of items”
    - Answers: “Does item X belong to the set ?”
  - Supports 2 operations
    - Insert an item in the set
    - Check if an item exists in the set
  - Probabilistic data structure
    - Allows false positives

Bloom filters

- Description
  - The data structure
    - A bitmap (an array of bits) of m bits
    - A number of hash functions
  - Insert an item in the set
    - Compute hash functions h(x,m), g(x,m), …
    - Set bits h(x,m), g(x,m), …
  - Search an item in the set
    - Test bits h(x,m), g(x,m), …

Bloom filters

- Errors
  - False negatives are not possible
    - If item x in set: h(x,m), g(x,m), … are all set
  - False positives are possible
    - h(x,m), g(x,m), … may be set by other items
    - 1 bit not set: $1-1/m$
    - k bits not set: $(1-1/m)^k$
    - k bits not set with n items in the filter: $(1-1/m)^{kn}$
    - 1 target bit is set: $1 - (1-1/m)^{kn}$
    - k target bits are set: $[1 - (1-1/m)^{kn}]^k$

Bloom filters in Databases

- Semi-Joins
  - Evaluate selections
    - Select tuples from table R if R.y > 5
    - Select tuples from table S if S.y < 3
  - Truncate join inputs using Bloom filters
    - Discard R tuples if R.x not in the S.x set
    - Discard S tuples if S.x not in the R.x set
  - Join remaining tuples
    - Filter tuples that the Bloom filters missed

The query:

```
select *
from R, S
where R.x = S.x
and R.y > 5
and S.y < 3
```

Bloom filters in Databases

- In parallel / distributed databases
  - Filter data to reduce network traffic
    - Network << RAM
    - Probing the Bloom filter > send over the network
    - Broadcast the filters —> small cost
- In main-memory database execution
  - Filter data as early as possible to reduce the working set
    - Filter before partitioning
    - If after: Bloom filter probing > hash table probing
    - Bloom filter fits in the cache often

Implementation

- Scalar implementation
  - Iterate over the hash functions / bit-tests
    - 1 access & bit-test / time
    - 1 hash function / time
  - Good performance —> short-circuit
    - Bit-test fail —> stop inner loop
    - Most keys fail early
  - Bad performance —> short-circuit
    - Branching logic —> branch mis-predictions & pipeline bubbles

Implementation

- Scalar implementation

```
for (o = i = 0 ; i != tuples ; ++i) {
  key = keys[i];                          // read the key
  for (f = 0 ; f != functions ; ++f) {    // iterate over functions
    h = hash[f](key);                     // compute the hash function
    if (bit_test(bitmap, h) == 0)         // perform bit-test (x86 instruction)
      goto failure;                       // early abort if bit-test fails
  }
  rids_out[o] = rids[i];                  // copy the payload to output
  keys_out[o++] = key;                    // write the key to output
failure:;                                 // jump here if not qualified
}
```

- Use multiplicative hashing
  - 1 multiplication
  - Universal family
  - Pair-wise independent functions easy

Implementation

- Scalar implementation

```
for (o = i = 0 ; i != tuples ; ++i) {
  key = keys[i];
  h = hash_1(key);                          // 1st function
  if (bit_test(bitmap, h) == 0) goto failure;
  h = hash_2(key);                          // 2nd function
  if (bit_test(bitmap, h) == 0) goto failure;
  [...]                                     // more functions unrolled
  rids_out[o] = rids[i];
  keys_out[o++] = key;
failure:;
}
```

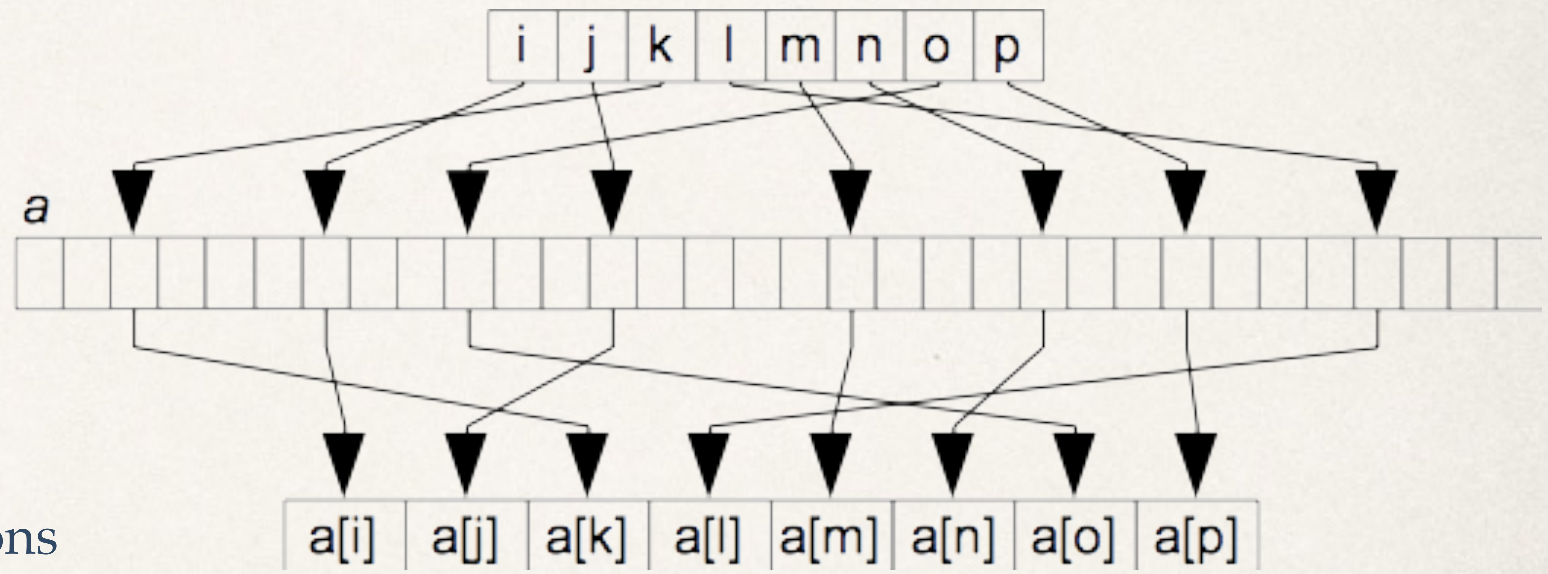
  - How much can be done ?
    - Unroll hash functions
    - Separate branches (prediction states) per function
    - Better branch prediction (hopefully)

SIMD in Databases

- SIMD on query execution
  - General usage
    - Scan, aggregation, index search [Zhou et.al. 2002]
  - For sorting / compressing
    - Comb-sort [Inoue et al. 2007]
    - Merge-sort using bitonic merging [Chhugani et al. 2008]
    - Range partitioning [Polychroniou et al. 2014]
    - Dictionary (de-)compression [Willhalm et al. 2009]
  - For indexing
    - Tree index search [Kim et al. 2010]
    - Hash table probing using multi-key buckets [Ross 2006]

Implementation

- SIMD loads
  - Sequential
    - 128/256/512 sequential bits
    - Align —> better performance
    - Mask reads
  - Fragmented
    - 32/64 bits from multiple locations
    - Indexes in another SIMD register
    - Loaded values packed in SIMD
    - Since Intel Haswell (2009)



Implementation

- SIMD without gathers
  - Scalar accesses
    - 256-bit load = 32-bit load
    - Pack in less space
    - Tree node accesses [Kim et.al. 2009]
    - Multi-key hash buckets [Ross 2006]
  - Fragmented accesses
    - Extract index from SIMD to scalar
    - Load each item individually
    - Pack values in SIMD

```
// extract indexes
i1 = _mm256_cvtsi128_si64(index);
i2 = _mm256_cvtsi128_si64(
       _mm256_permute4x64_epi64(index, 1));
i3 = _mm256_cvtsi128_si64(
       _mm256_permute4x64_epi64(index, 2));
i4 = _mm256_cvtsi128_si64(
       _mm256_permute4x64_epi64(index, 3));

// load values one at a time
v1 = _mm_load_epi64(&data[i1]);
v2 = _mm_load_epi64(&data[i2]);
v3 = _mm_load_epi64(&data[i3]);
v4 = _mm_load_epi64(&data[i4]);

// pack values
v12 = _mm256_unpacklo_epi64(v1, v2);
v34 = _mm256_unpacklo_epi64(v3, v4);
value = _mm256_permute2x128_si256(v12,
                                  v34, 64);
```

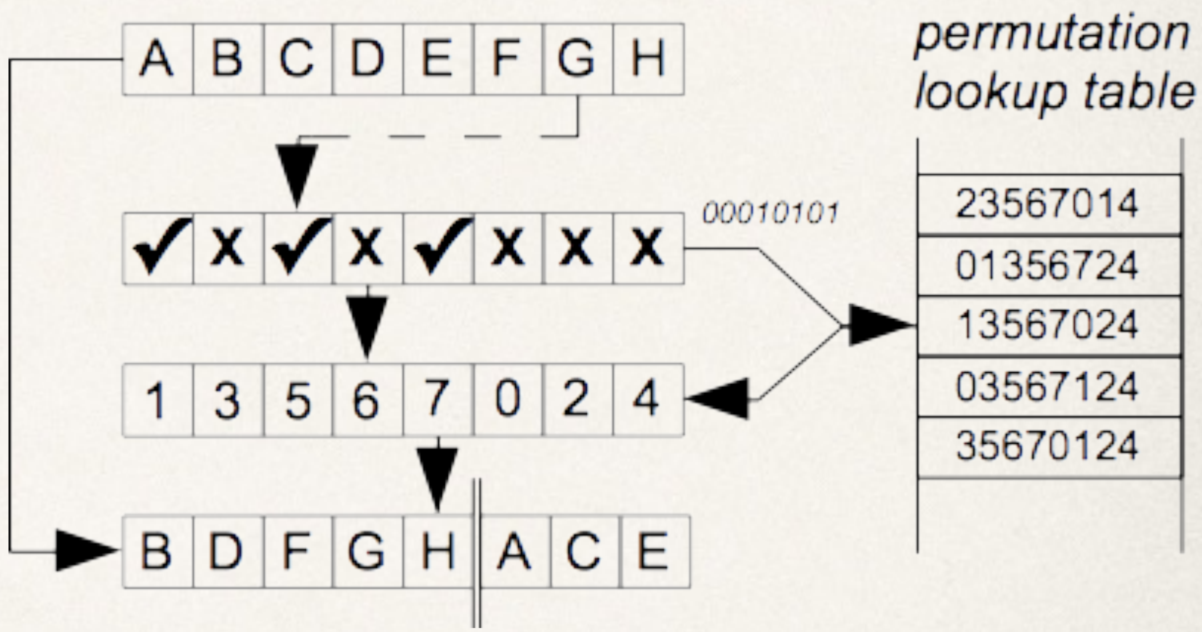
Implementation

- Using SIMD for Bloom filters
  - Vectorizing hashing / access / bit-test
    - Multiplicative hash in SIMD
    - 32-bit gather to access the bitmap on hash div 32
    - Mask with 1 bit shifted using hash mod 32
  - “How” to vectorize >1 functions ?
    - k=1 —> similar to selection scan
    - Maintain short-circuit
    - Avoid branching
    - Minimize loads/stores

```
// multiplicative hashing
hash = _mm256_mullo_epi32(key, factor);
hash = _mm256_srli_epi32(hash, shift);

// bit-test
index = _mm256_srli_epi32(hash, 5);
bit = _mm256_and_si256(hash, mask_31);
data = _mm256_i32gather_epi32(bitmap, index, 4);
bit = _mm256_sllv_epi32(mask_1, bit);
data = _mm256_and_epi32(data, bit);
aborts = _mm256_cmpeq_epi32(data, mask_0);
```

Implementation

- SIMD 2-way partitioning
  - Using SIMD permutations
    - Register to register "gather"
    - "Pull"-based shuffling
  - Using boolean result bitmap as an index
    - Get boolean results —> extract bitmap
    - Load permutation mask
    - Permute vector to "true" and "false"
    - W SIMD lanes = 2^W permutation mask
    - Best stored in W * 2^W bytes —> L1 for 8-way SIMD




```
// load 8-way permutation mask
bitmap = _mm256_movemask_ps(aborts);
mask = _mm_load_epi64(&perm_table[bitmap]);
mask = _mm256_cvtepi8_epi32(mask);

// permute keys & rids
key = _mm256_permutevar8x32_epi32(key, mask);
rid = _mm256_permutevar8x32_epi32(rid, mask);
```

Implementation

- Conditional control flow transformation
  - Maintain short-circuit logic
    - Never do multiple bit-tests for the same key
    - First bit-test fails —> second bit-test wasted
    - **Process a different input key per lane**
  - Arbitrary hash function per lane
    - Maintain function indexes (per lane)
    - Any hash function (per lane)
    - Function index = k —> tuple qualifies !
    - “Gather” hash functions from register (not L1)

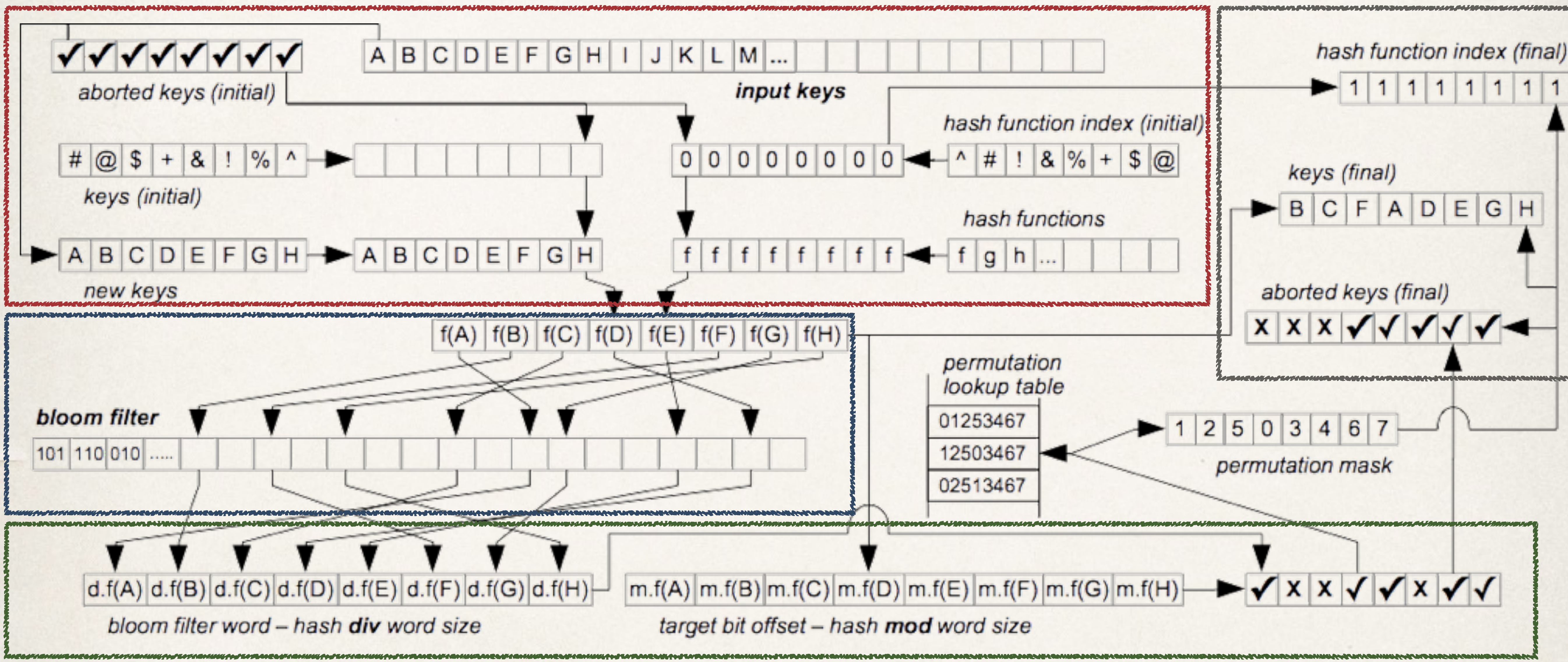
```
// choose hash function per key
factor = _mm256_permutevar8x32_epi32(factors,
                                      fun);
// increment function index
fun = _mm256_add_epi32(fun, mask_1);
done = _mm256_cmpeq_epi32(fun, mask_k);

// multiplicative hashing
hash = _mm256_mullo_epi32(key, factor);
hash = _mm256_srli_epi32(hash, shift);
```

Implementation

- Conditional control flow transformation
  - Dynamic input reading
    - Recycle lanes that failed a bit-test
    - Permute SIMD vector in two parts
    - Refill aborted part of the vector
    - Advance input pointer
    - Word-aligned access
  - Dynamic output writing
    - SIMD permute —> write qualifiers
    - Advance output pointer

```
// read new keys & payloads
new_key = _mm256_maskload_epi32(keys, aborts);
new_val = _mm256_maskload_epi32(vals, aborts);

// clear aborted data
key = _mm256_andnot_si256(aborts, key);
rid = _mm256_andnot_si256(aborts, rid);
fun = _mm256_andnot_si256(aborts, fun);

// mix old with new items
key = _mm256_or_si256(key, new_key);
rid = _mm256_or_si256(rid, new_rid);

// perform bit-tests and permute data
[…]
bitmap = […]

// advance input pointers by counting bits
keys += _mm_popcnt_u64(bitmap);
rids += _mm_popcnt_u64(bitmap);
```

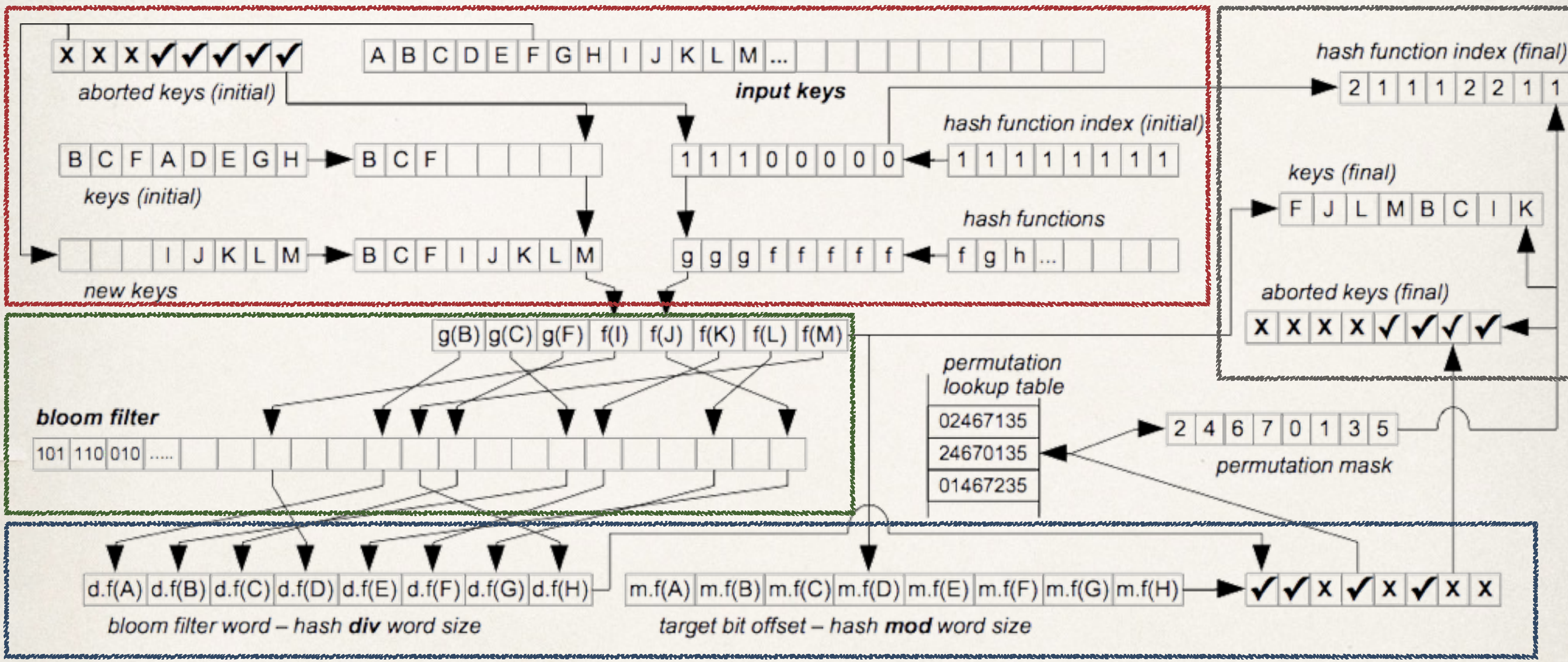
Example



- First loop
  - 32-bit keys, no payloads, no output code

1) Input & hashing
2) Bitmap access
3) Bit-testing
4) Permutations

Example



- Second loop
  - 32-bit keys, no payloads, no output code

1) Input & hashing
2) Bitmap access
3) Bit-testing
4) Permutations

Implementation

- Writing the output
  - Use branching
    - Low selectivity —> rarely taken
    - Skipped otherwise
  - Filter data
    - SIMD permute
    - Store sequentially
    - Qualifiers “aborted”
    - Advance output pointers
    - Same as selection filtering

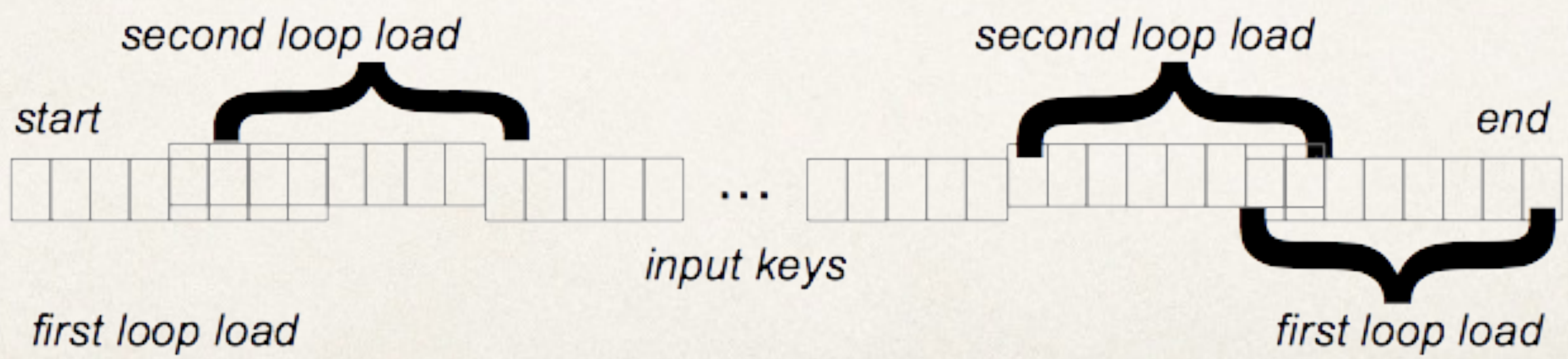
```
// any qualifiers ?
done = _mm256_cmpeq_epi32(fun, functions);
done = _mm256_andnot_si256(aborts, done);
if (!_mm256_testz_si256(done, done)) {

  // load permutation mask
  bitmap = _mm256_movemask_ps(done);
  mask = _mm256_loadl_epi64(&perm_table[bitmap]);
  mask = _mm256_cvtepi8_epi32(mask);

  // permute data and mask
  key_out = _mm256_permutevar8x32_epi32(key, mask);
  rid_out = _mm256_permutevar8x32_epi32(key, mask);
  done = _mm256_permutevar8x32_epi32(done, mask);

  // write qualifiers to output
  _mm256_maskstore_epi32(keys_out, done);
  _mm256_maskstore_epi32(rids_out, done);

  // update output pointer by counting bits
  keys_out += _mm_popcnt_u64(done);
  rids_out += _mm_popcnt_u64(done);
}
```

Implementation

- Loop unrolling
  - In general
    - Interleave instructions to increase IPC
    - Crucial for in-order CPUs
    - (Should be) irrelevant in out-of-order CPUs
    - Can still improve performance
    - Limited by number of registers to hold “state”
  - For Bloom filters
    - Dynamic input reading —> naive loop unrolling does not work
    - Read the input from non-overlapping locations

Implementation

- Loop unrolling
  - In Bloom filter probing
    - Read the input from non-overlapping locations
    - Simplest way: low —> high & high —> low
    - Allows for 2-way loop unrolling
    - Stop when the two pointers meet

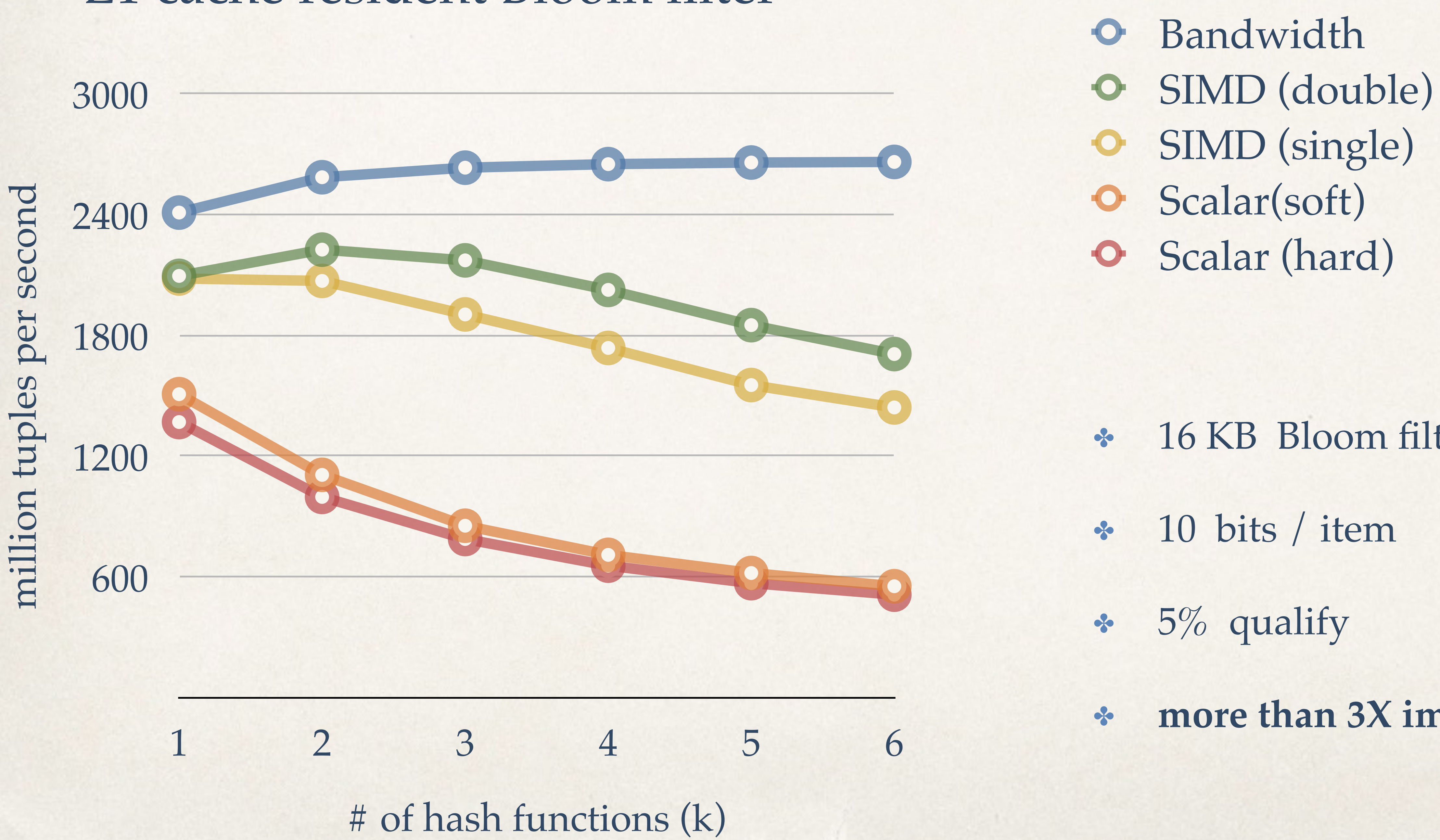


Experiments

- Experimental setup
  - Hardware platform & software setting
    - 1 Intel Xeon E3-1675 v3 CPU @ 3.5 GHz with 4-cores & 2-way SMT
    - 32 GB of DDR3 RAM @ 1600 MHz
    - Running 8 threads & shared Bloom filter
    - Using 32-bit keys & 32-bit payloads
  - Figures
    - Scalar soft: standard scalar implementation
    - Scalar hard: scalar implementation with unrolled branches
    - SIMD single: standard SIMD implementation
    - SIMD double: SIMD implementation with unrolled loop

Experiments

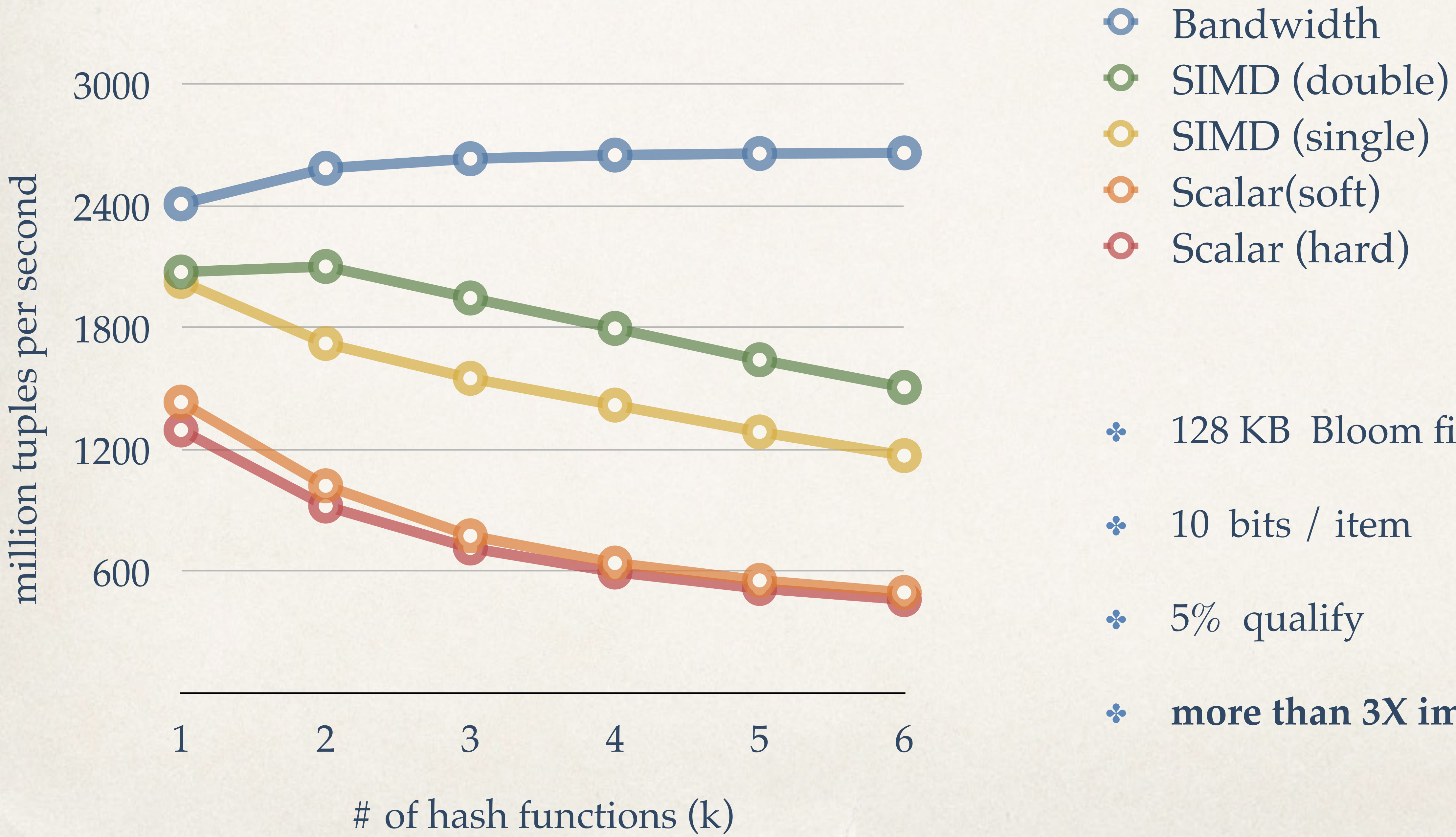
- L1 cache resident Bloom filter



- 16 KB Bloom filter
- 10 bits / item
- 5% qualify
- **more than 3X improvement !**

Experiments

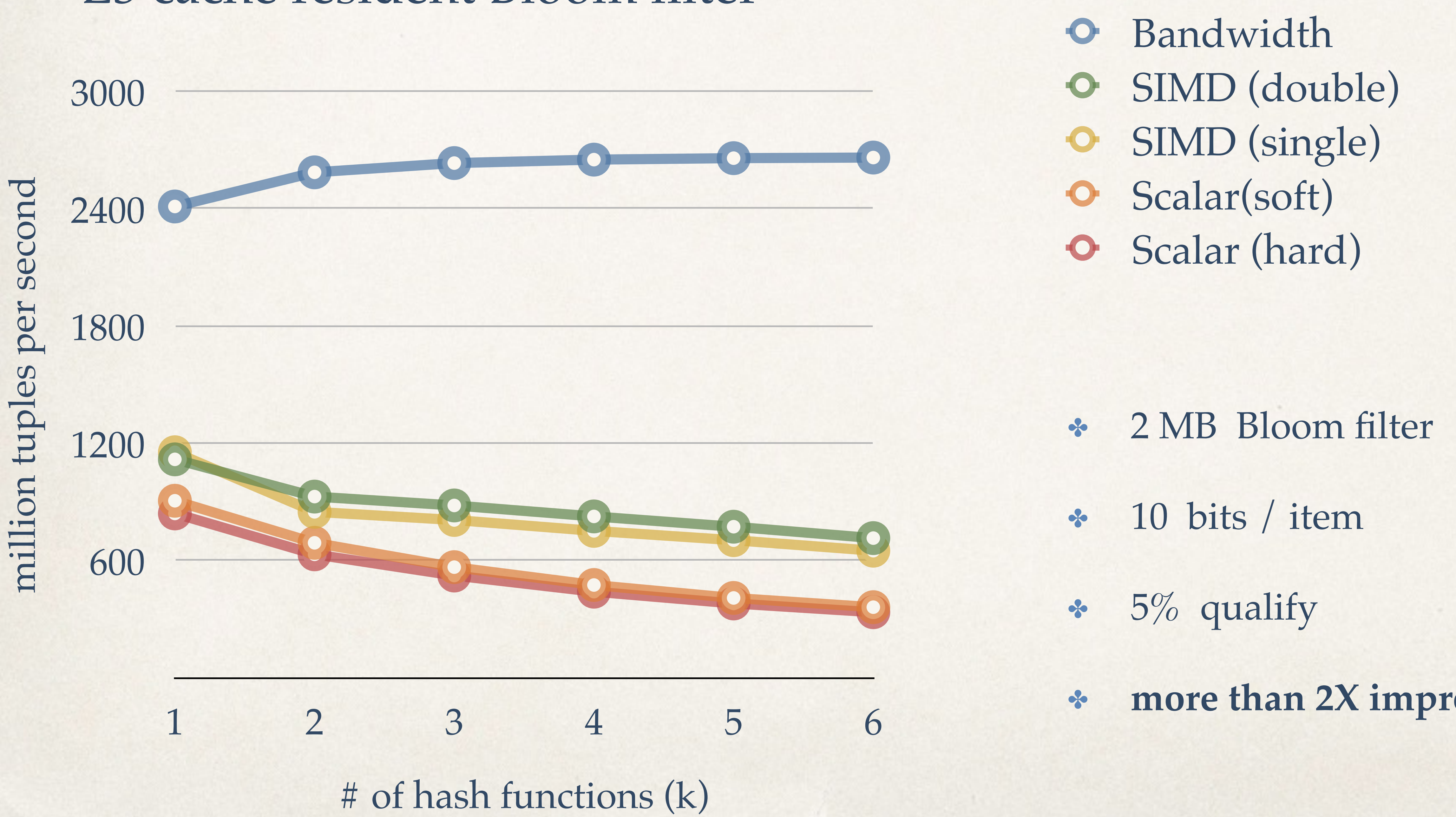
- L2 cache resident Bloom filter



- 128 KB Bloom filter
- 10 bits / item
- 5% qualify
- **more than 3X improvement !**

Experiments

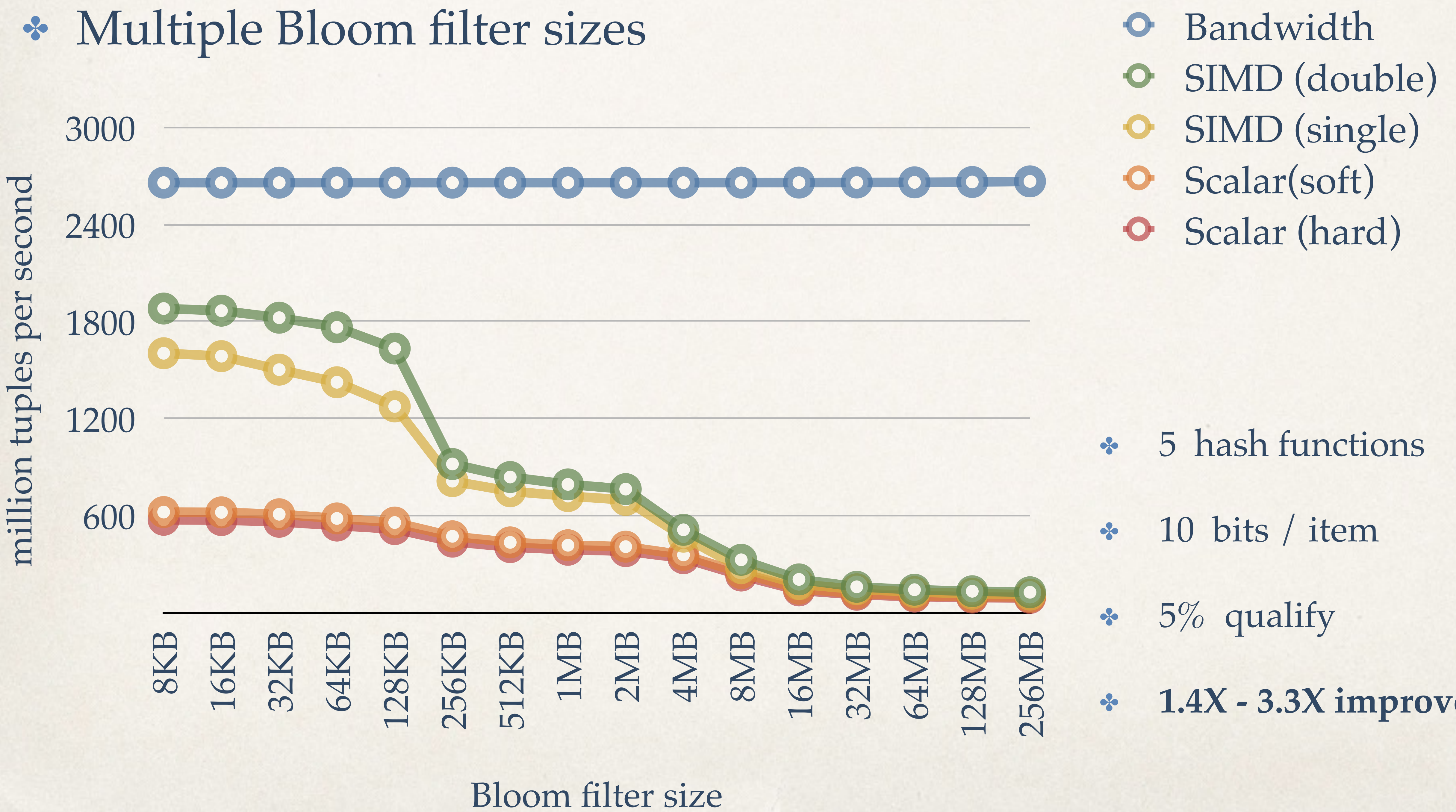
- L3 cache resident Bloom filter



- 2 MB Bloom filter
- 10 bits / item
- 5% qualify
- **more than 2X improvement !**

Experiments

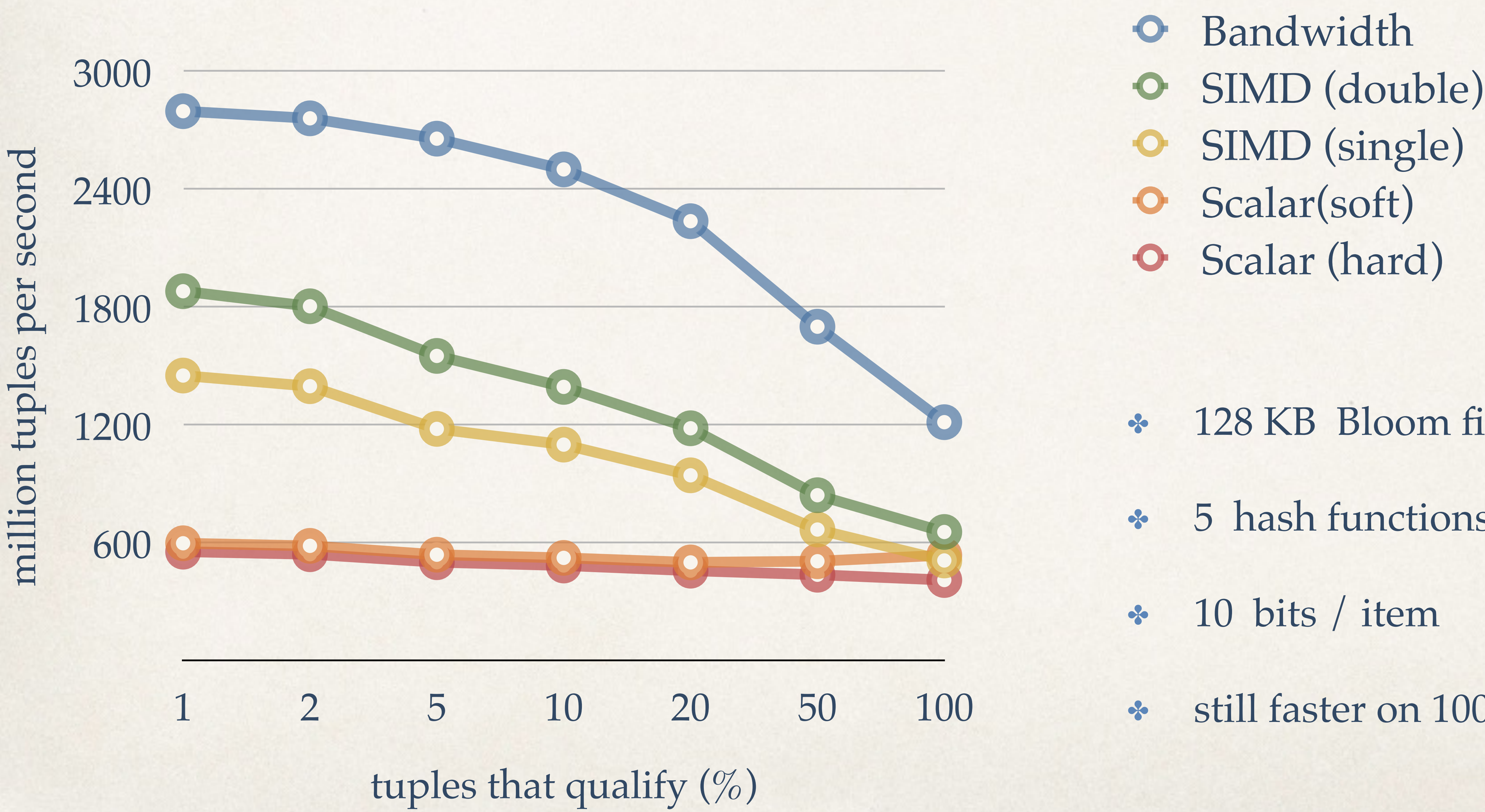
- Multiple Bloom filter sizes



- 5 hash functions
- 10 bits / item
- 5% qualify
- **1.4X - 3.3X improvement !**

Experiments

- Multiple selectivities



- 128 KB Bloom filter (L2)
- 5 hash functions
- 10 bits / item
- still faster on 100% selectivity

Conclusions

- Vectorized Bloom filters
  - Implementation
    - Access data non-sequentially in SIMD
    - Eliminate conditional control flow
    - Maintain short-circuit
    - Non-trivial loop unrolling
    - **Re-usable** techniques for other structures
  - Performance
    - More than **3X** faster when cache-resident
    - Still faster when operating off the cache or all tuples qualify

Questions