Exercise Session 7

Data Processing on Modern Hardware
263-3502-00L — Fall Semester 2012

Cagri Balkesen
cagri.balkesen@inf.ethz.ch

Department of Computer Science ETH Zurich, Switzerland

1 November 2012

The Classic Hash Join

To compute $R \bowtie S$,

1. **Build a hash table** on the "inner" join relation S. } Hash Phase
2. **Scan** the "outer" relation R and **probe** into the hash table for each tuple $r \in R$. } Join Phase

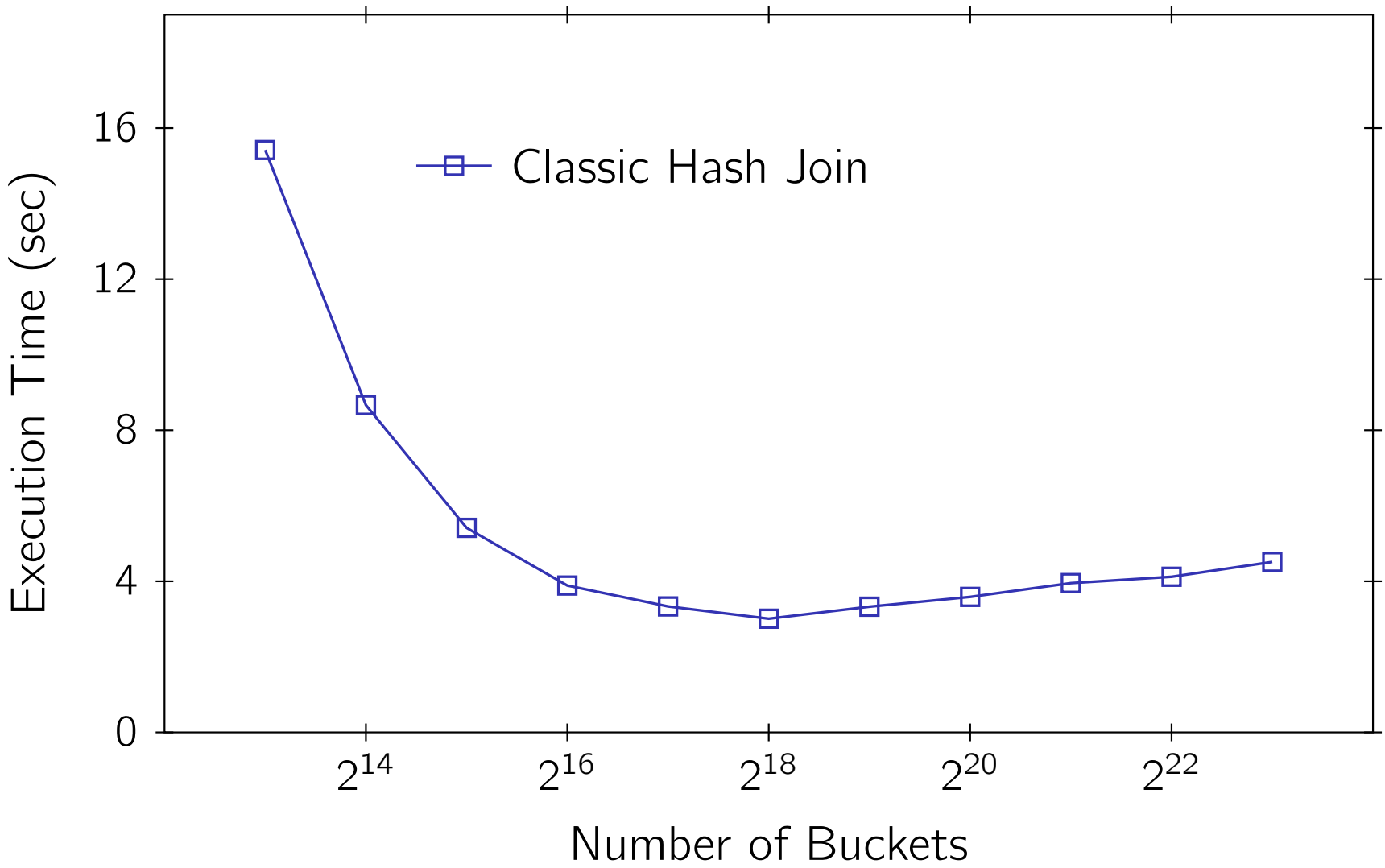
```
1 Function: hash_join(R, S)
  // Hash Phase
2 foreach tuple s ∈ S do
3     insert s into hash table H ;
  // Join Phase
4 foreach tuple r ∈ R do
5     probe H and append matching tuples to result ;
```

Architecture of Test Machine

- Intel(R) Core(TM)2 Duo CPU P9600@2.53GHz
- L1 (Data) : 32K, 8-way set associative
- L1 (Instruction) : 32K, 8-way set associative
- L2 (Shared) : 6144K, 24-way set associative
- DTLB0 : 16 Entries
- DTLB1 : 256 Entries

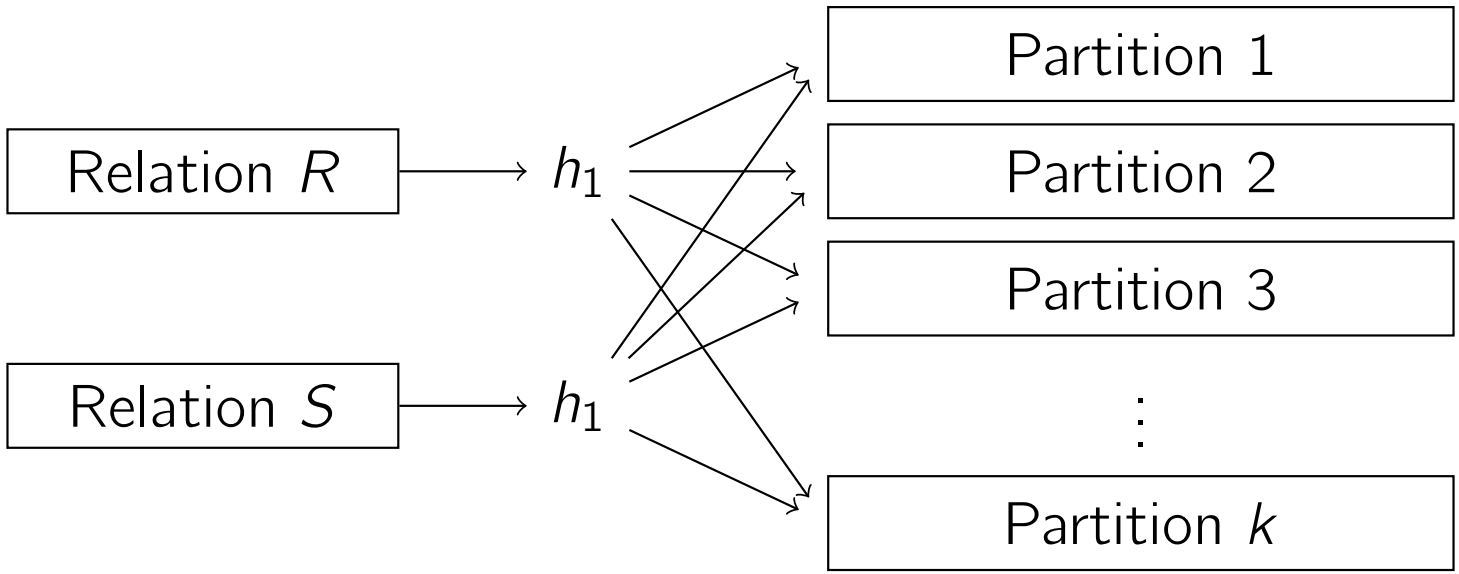
Classic Hash Join → Bucket Size

Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



Partitioned Hash Join

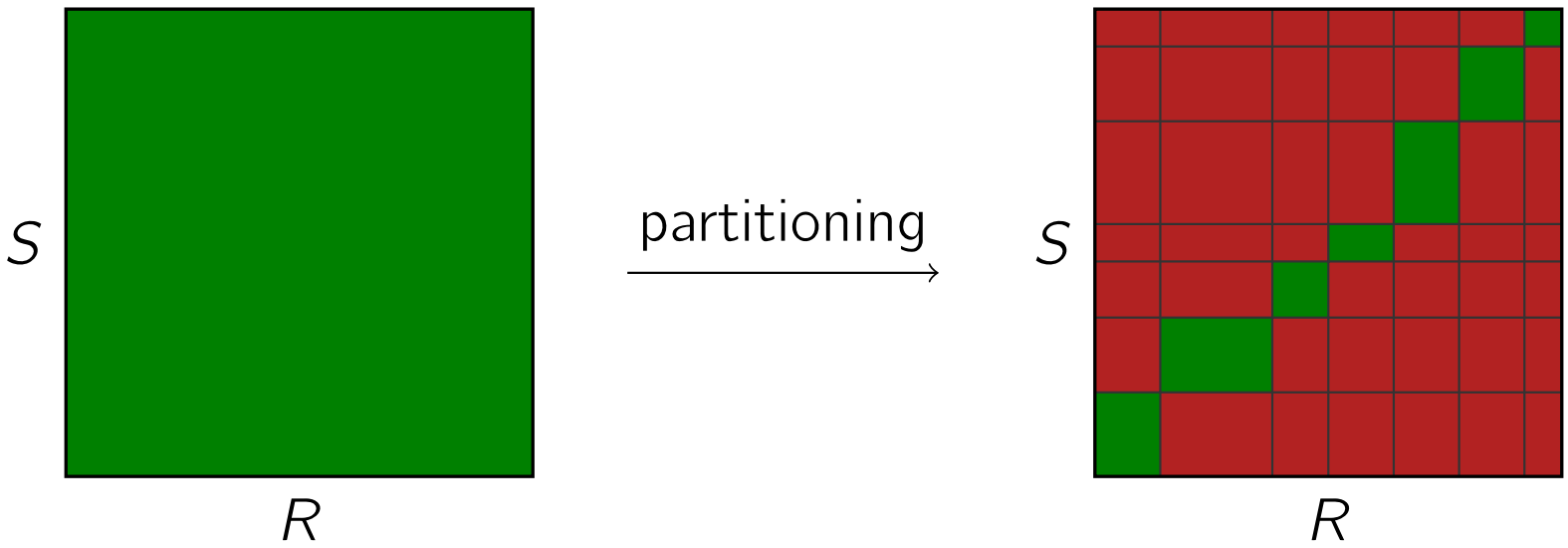
Thus: **Partition** input into **cache-sized chunks**.



- Choose k ($\rightsquigarrow h_1$) such that single partitions **fit into cache**.
- Partition **both** input relations using the **same** hash function h_1.
- Only need to join **within** partitions then.

Partitioned Hash Join

- Potentially matching tuples will end up in corresponding partitions.

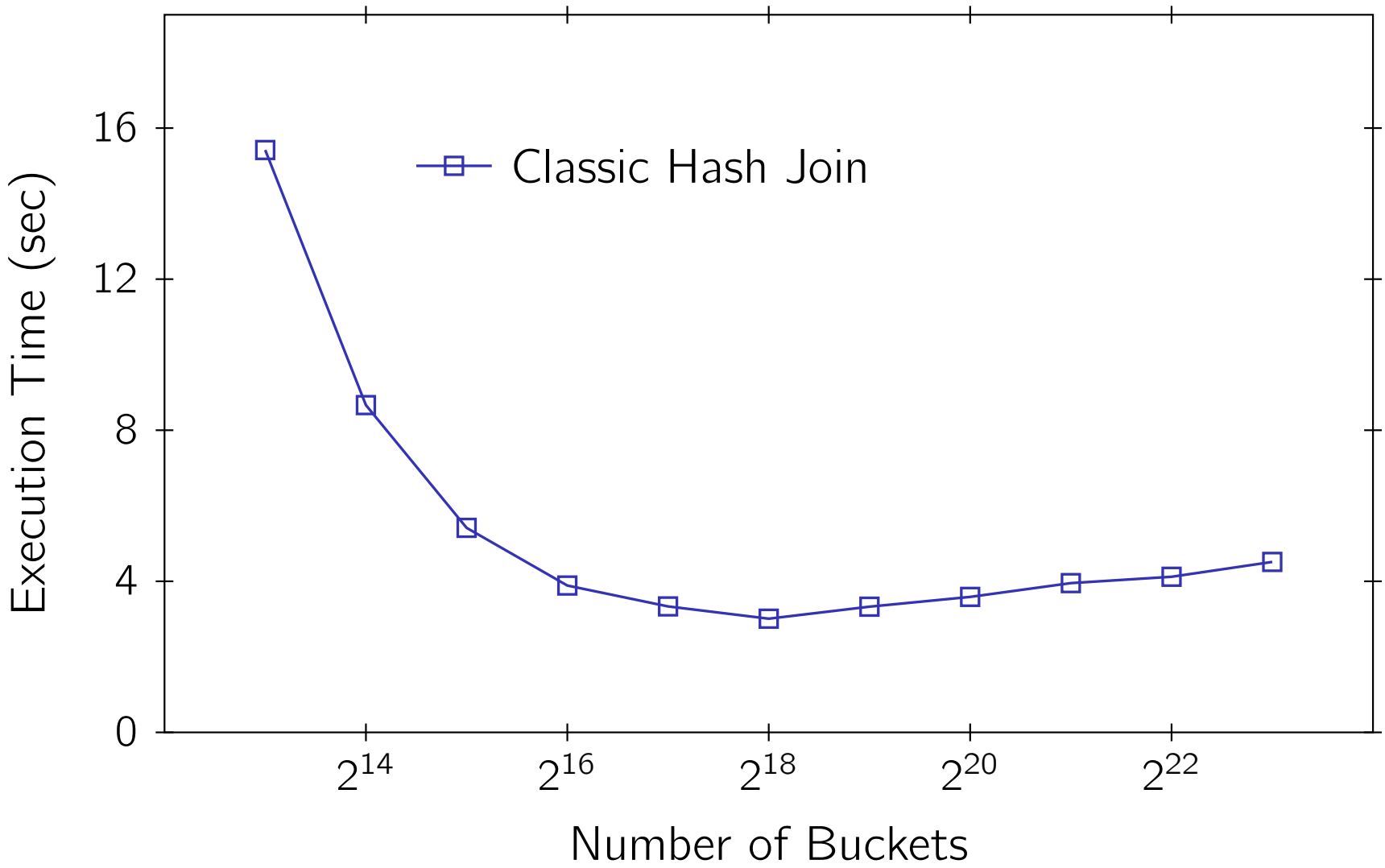


partitioned_hash_join(. . .)

```
create_hashtable(R, &hashtable_R, num_buckets, 0);
create_hashtable(S, &hashtable_S, num_buckets, 0);

for(i=0; i<num_buckets; i++) {
  if((hashtable_R.num_tuples[i] > 0) && (hashtable_S.num_tuples[i] > 0)) {
    partial_R.num_tuples = hashtable_R.num_tuples[i];
    partial_R.tuples = hashtable_R.tuples[i];

    partial_S.num_tuples = hashtable_S.num_tuples[i];
    partial_S.tuples = hashtable_S.tuples[i];

    result += hash_join(&partial_R, &partial_S, partial_R.num_tuples);
    //result += nested_loops_join(&partial_R, &partial_S);
  }
}
```

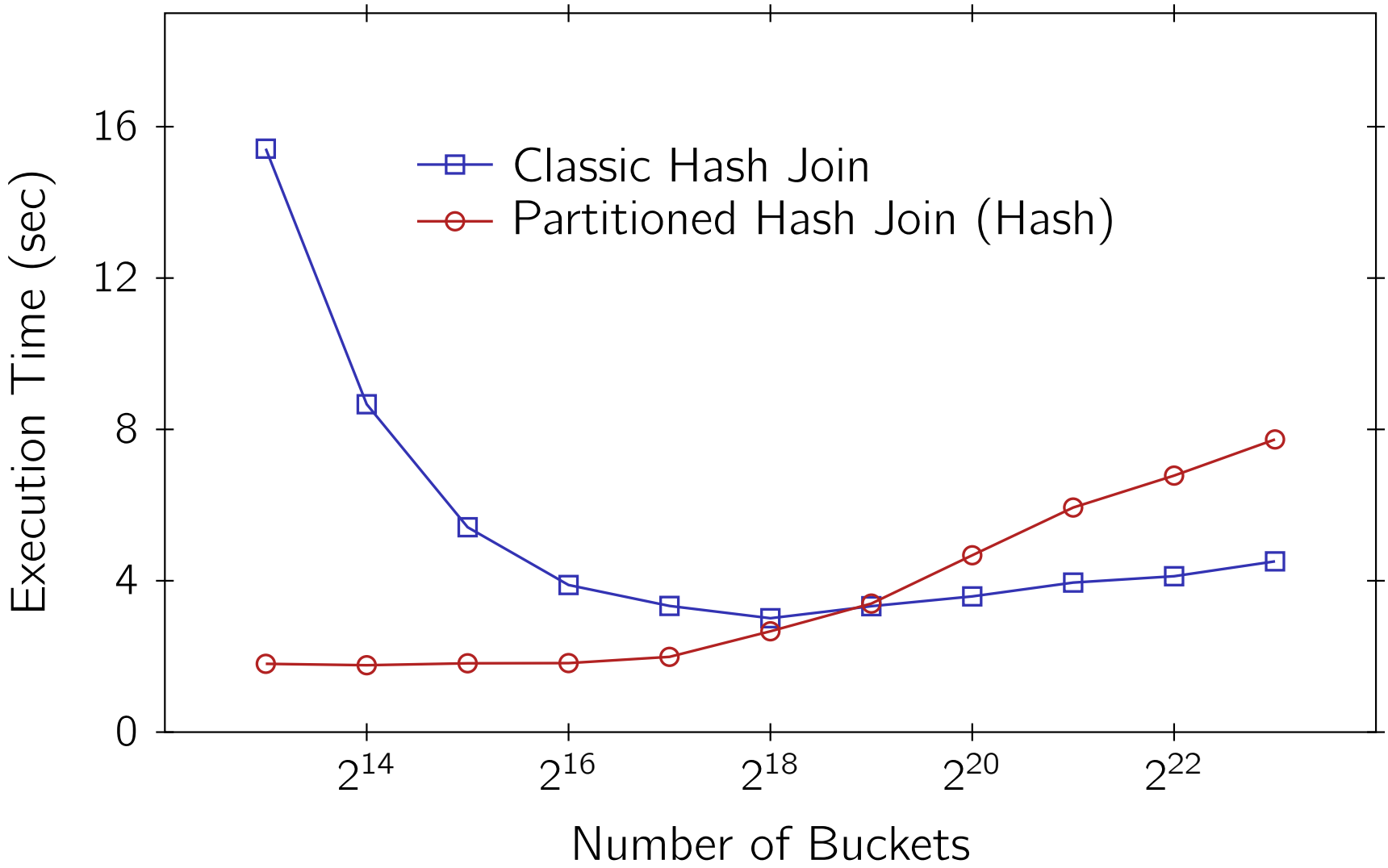
Classic Hash Join vs. Partitioned Hash Join

Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



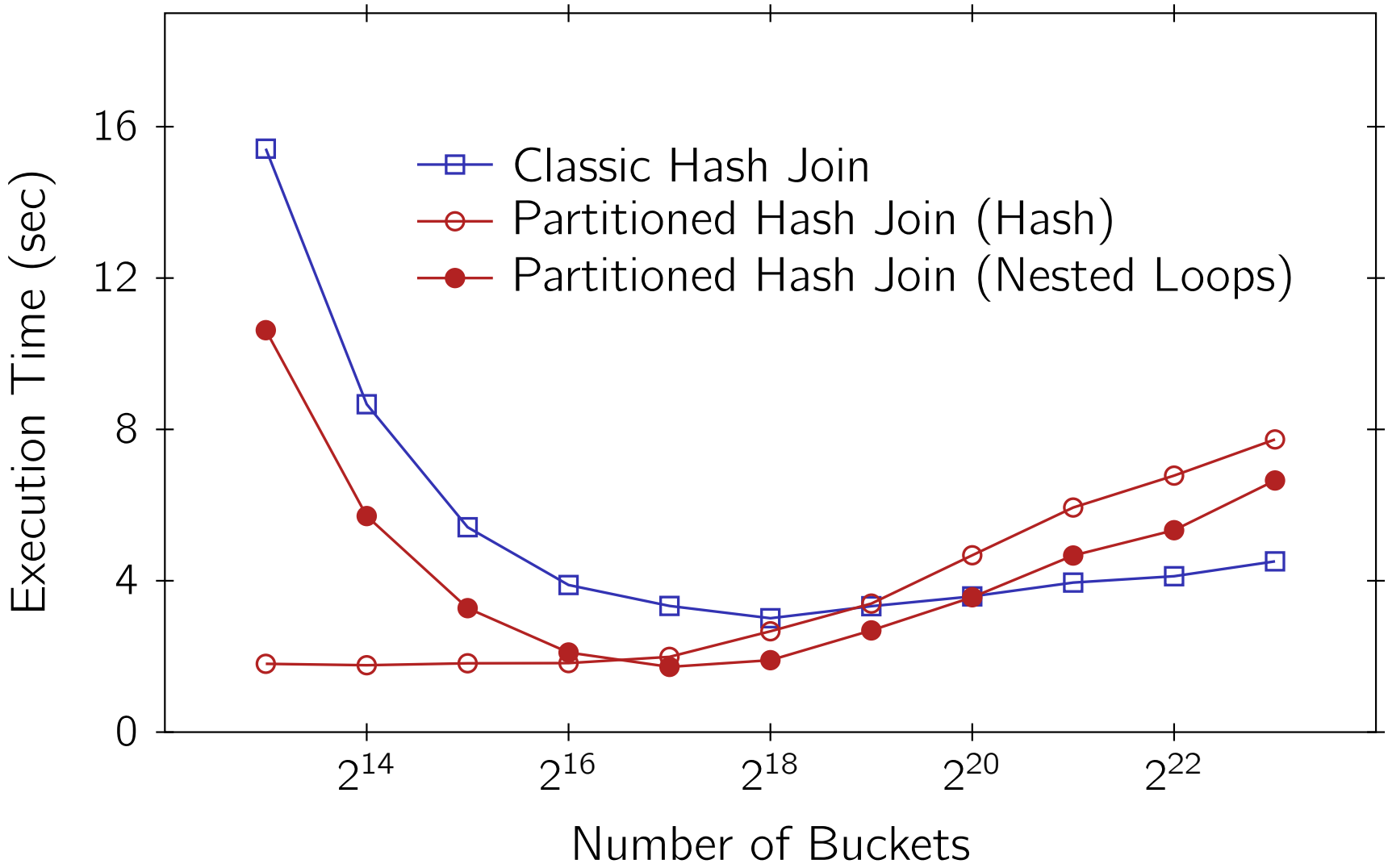
Classic Hash Join vs. Partitioned Hash Join

Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



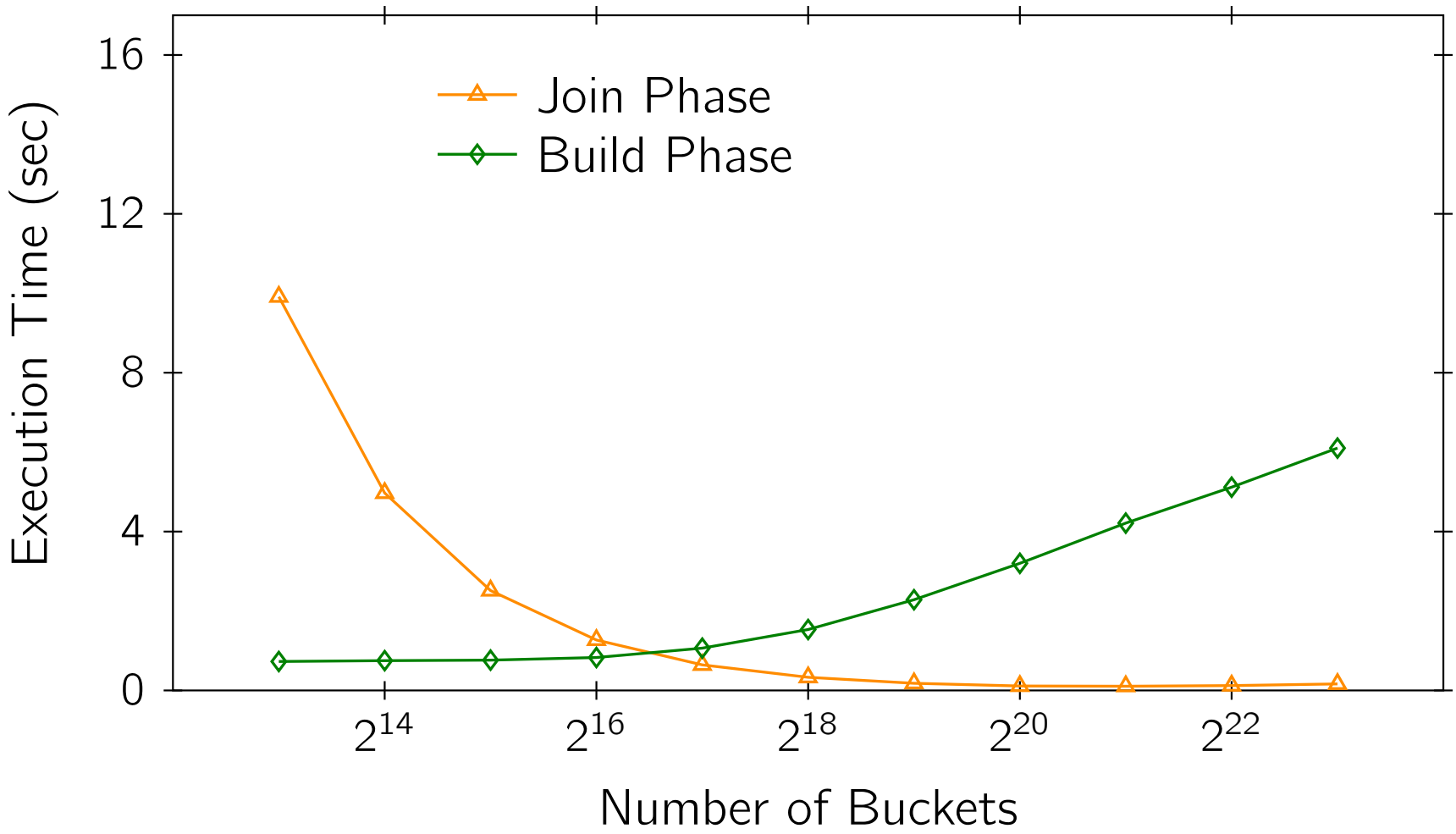
Classic Hash Join vs. Partitioned Hash Join

Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



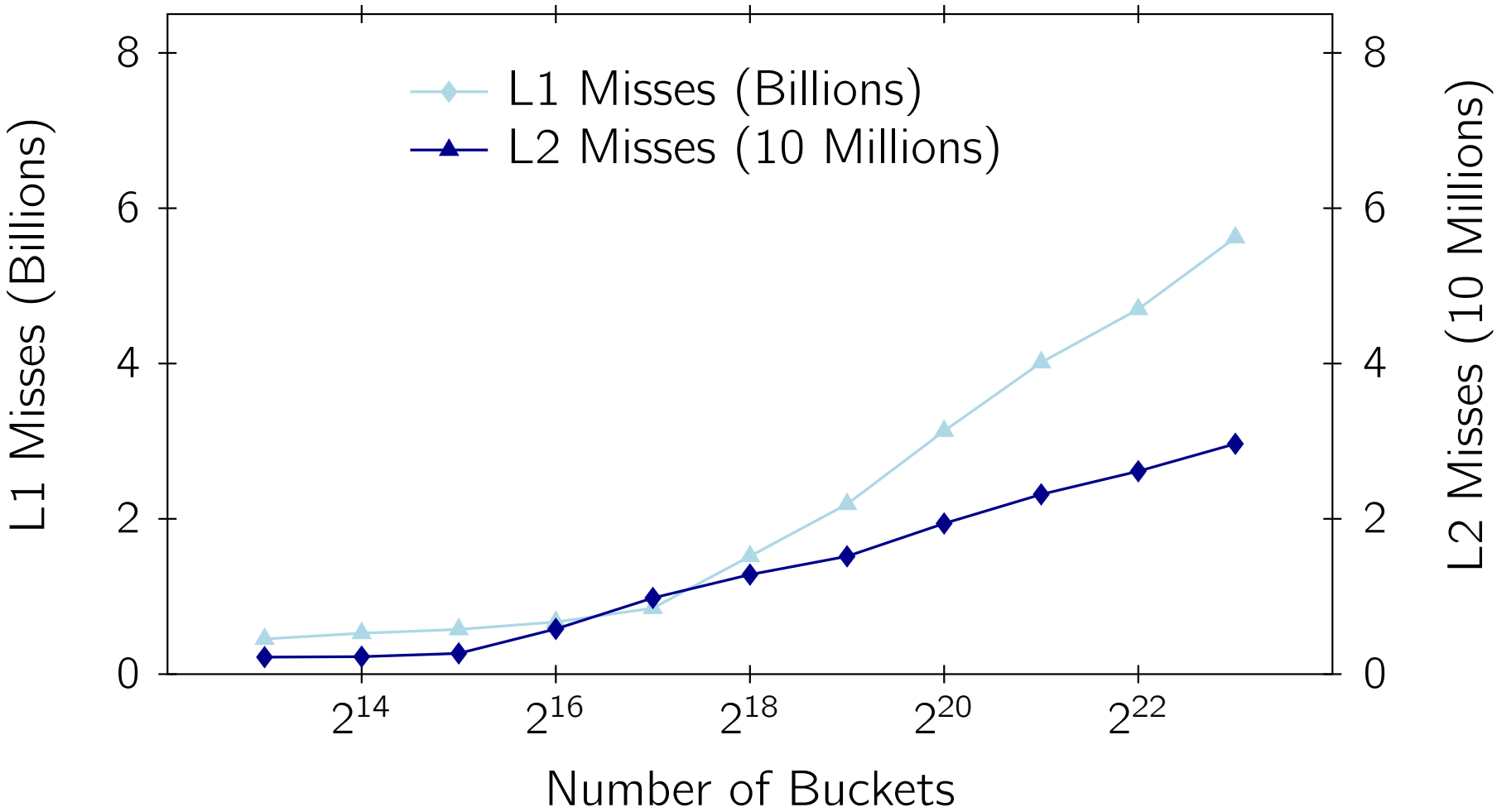
Build Phase vs. Join Phase (Execution Time)

Partitioned Hash Join (Nested Loops)
Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



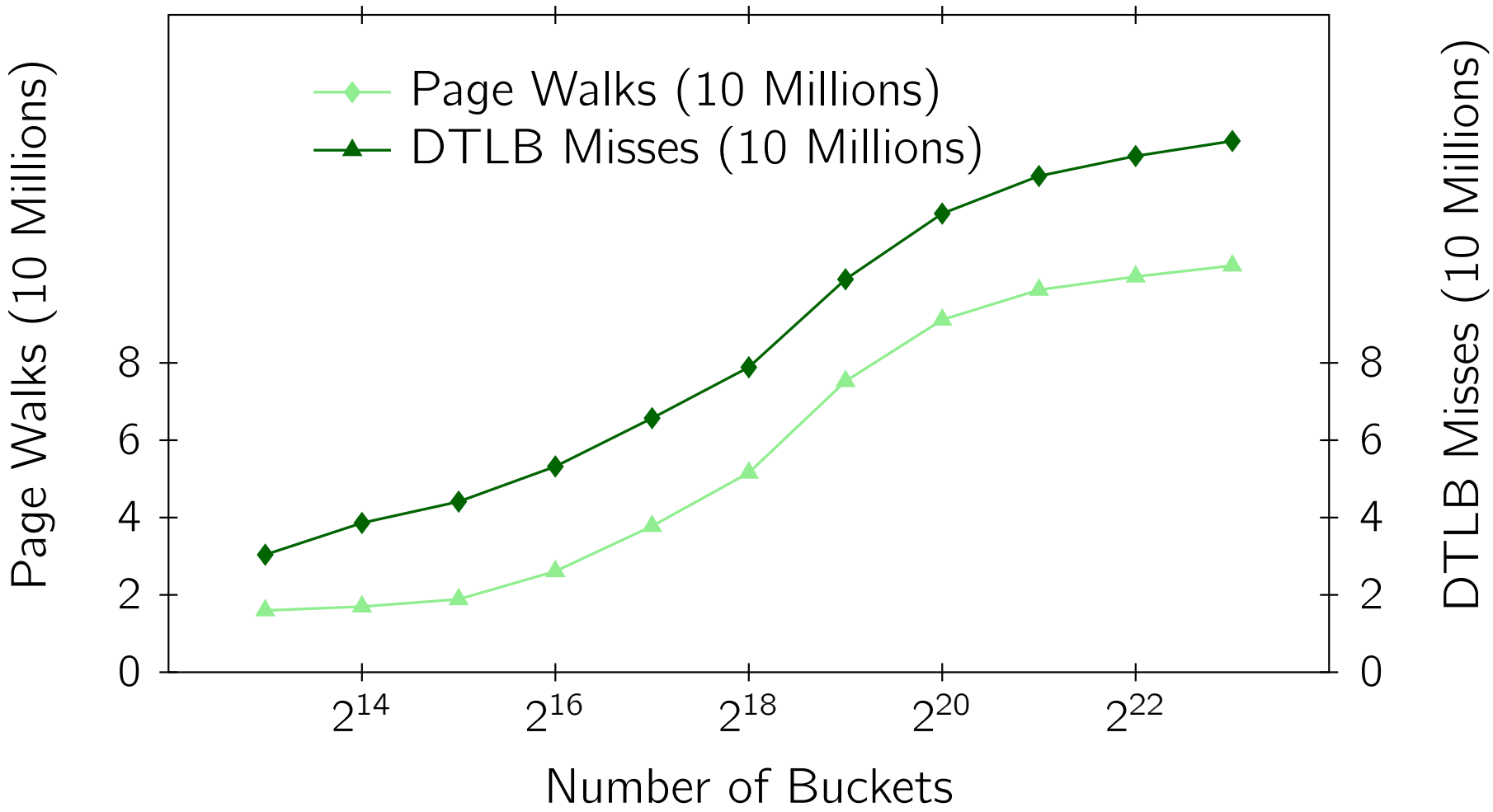
Build Phase: Cache Misses

Partitioned Hash Join (Nested Loops)
Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



Build Phase: TLB Misses

Partitioned Hash Join (Nested Loops)
Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



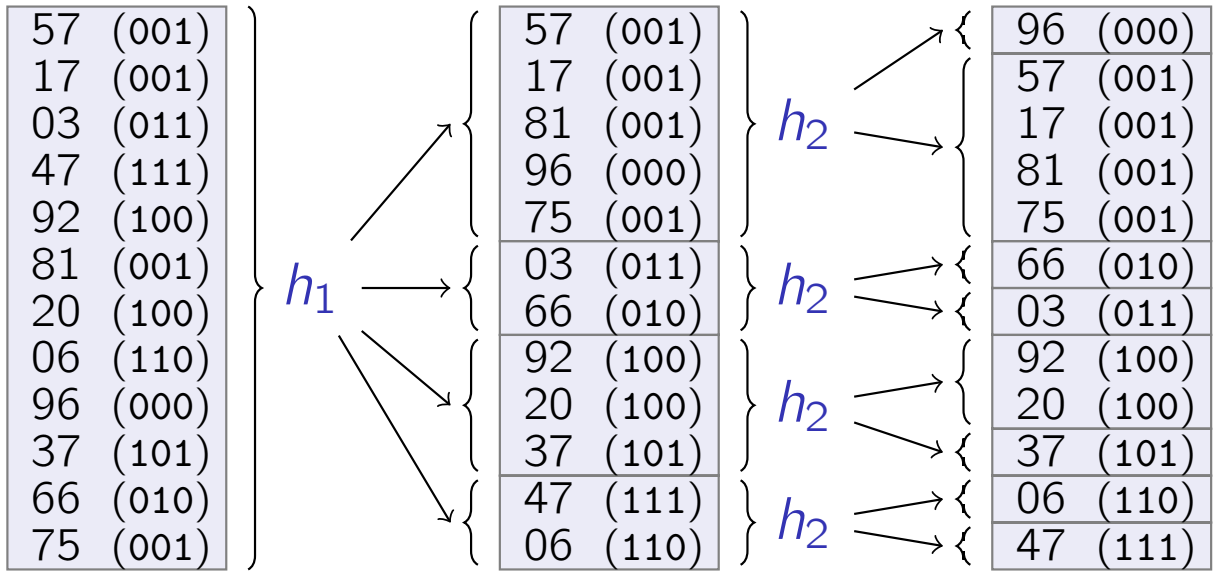
OProfile Script with Stream EDitor (sed)

```
# sampled events
#EVENTS="L1D_PEND_MISS:10000 LLC_MISSES:10000"
EVENTS="PAGE_WALKS:10000:1 DTLB_MISSES:10000"

...

EXECUTABLE="./PartitionedHashJoin"
OUTFILE="oprofile.txt"

for numbucketslog2 in ‘seq 13 23‘
do
  numbuckets=$((1<<$numbucketslog2))
  make clean
  make PartitionedHashJoin NUM_BUCKETS=$numbuckets RADIXBITS=$numbucketslog2

  sudo opcontrol --init
  sudo opcontrol --reset
  sudo opcontrol --setup --no-vmlinux $EVENT_LIST --image=PartitionedHashJoin

  sudo opcontrol --start-daemon

  sudo opcontrol --start

  $EXECUTABLE

  sudo opcontrol --shutdown

  echo -n "create_hashtable " >> $OUTFILE
  opreport -l | sed -n 's/create_hashtable//p' | sed 's/[0-9]*\.[0-9]*//g'>>$OUTFILE

done
```

Improving the Partition Phase

Hardware determines the maximum number of partitions we can create **at the same time**.

Idea: Partition in **multiple passes**.



Radix-Cluster Algorithm:

- $h_1, \ldots, h_P$ use same hash function but look at different bits.

Extend Partitioned Hash Join (→ Two-Passes)

- Add *shift* argument to `create_hashtable` so that we can look at different bits in every pass

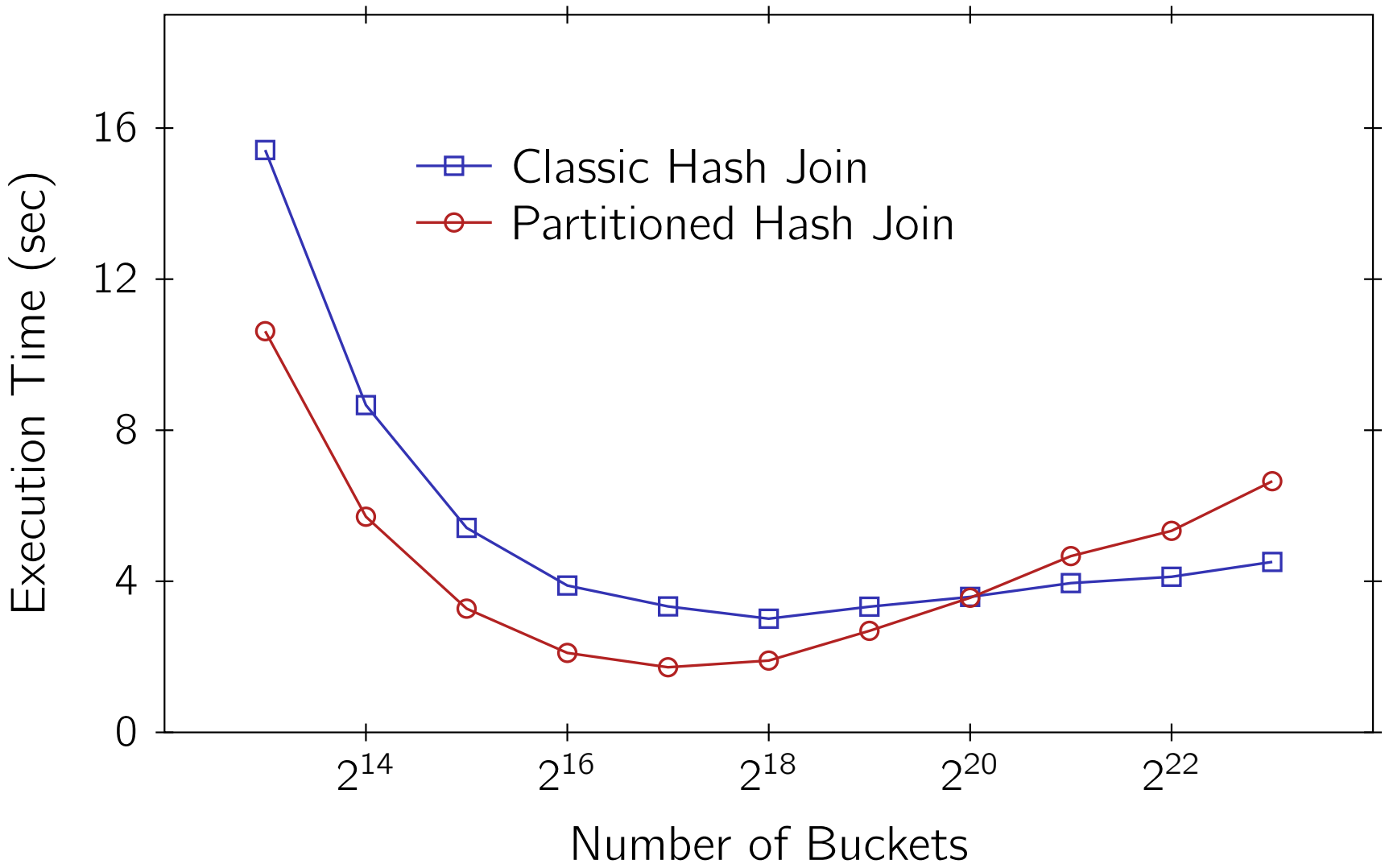
```
void create_hashtable(relation_t *rel, hashtable_t *hashtable, uint32_t num_buckets, uint32_t shift) {
  ...

  // count elements in each partition
  for (i = 0; i < rel->num_tuples; i++) {
    key = HASH(((rel->tuples[i].id)>>shift), num_buckets);
    cnt_elements_per_bucket[key]++;
  }

  ...

  // store relation in hash table
  for (i = 0; i < rel->num_tuples; i++) {
    key = HASH(((rel->tuples[i].id)>>shift), num_buckets);
    memcpy(&(hashtable->tuples[key][bucket_offset[key]]), &(rel->tuples[i]), sizeof(tuple_t));
    bucket_offset[key]++;
  }

}
```

Extend Partitioned Hash Join (→ Two-Passes)

```
long long unsigned multipass_partitioned_hash_join(relation_t *R, relation_t *S) {
  ...

  // first pass
  create_hashtable(R, &hashtable_R, num_buckets, 10);
  create_hashtable(S, &hashtable_S, num_buckets, 10);

  // second pass & join
  for(i=0; i<num_buckets; i++) {
    if((hashtable_R.num_tuples[i] > 0) && (hashtable_S.num_tuples[i] > 0)) {

      partial_R.num_tuples = hashtable_R.num_tuples[i];
      partial_S.num_tuples = hashtable_S.num_tuples[i];

      partial_num_buckets = 1024;

      create_hashtable(&partial_R, &partial_hashtable_R, partial_num_buckets, 0);
      create_hashtable(&partial_S, &partial_hashtable_S, partial_num_buckets, 0);

      // join current partition
      for(j=0; j<partial_num_buckets; j++) {
        if((partial_hashtable_R.num_tuples[j] > 0) && (partial_hashtable_S.num_tuples[j] > 0)) {
          ppartial_R.tuples = partial_hashtable_R.tuples[j];
          ppartial_S.tuples = partial_hashtable_S.tuples[j];

          result += nested_loops_join(&ppartial_R, &ppartial_S);
        }
      }
    }
  }
}
```

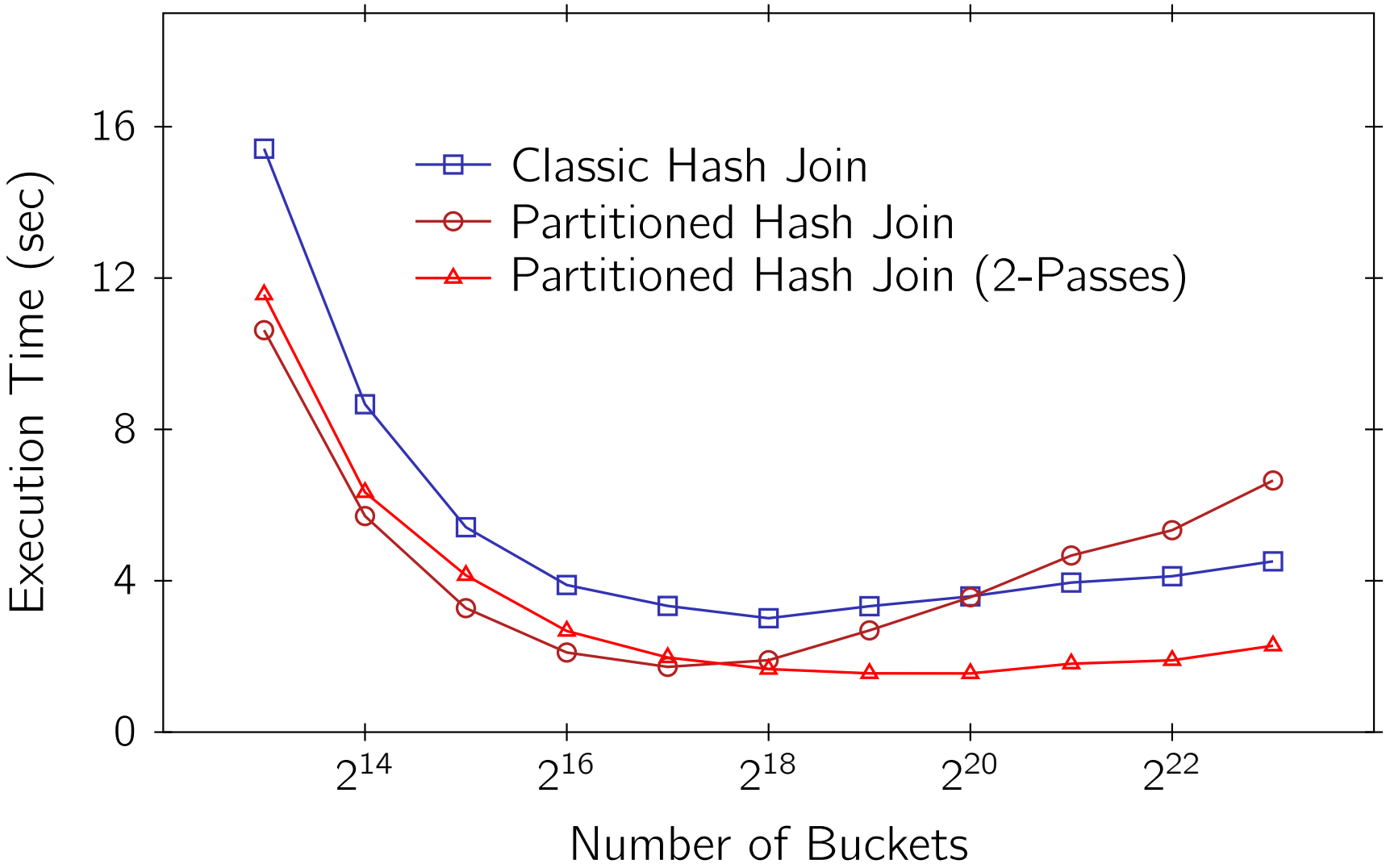
Partitioned Hash Join—Two Passes

Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



Partitioned Hash Join—Two Passes

Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



Radix Cluster Algorithm: Partition

```
void radix_cluster(relation_t *relin, relation_t *relout, uint32_t D, uint32_t R) {
  ...

  // mask for radix bits
  M = ((1 << D) - 1) << R;

  // number of buckets
  num_buckets = (1 << (D+R));

  // allocate memory to count elements per bucket, init
  bucket_count = calloc(num_buckets, sizeof(*bucket_count));
  bucket_offset = calloc(num_buckets, sizeof(*bucket_offset));

  // count elements in each partition
  for (i = 0; i < relin->num_tuples; i++) {
   key = HASHFUNC(relin->tuples[i].id) & M;
   bucket_count[key]++;
  }

  // compute start index of every bucket
  for(i = 1; i < num_buckets; i++) {
    bucket_offset[i] = bucket_offset[i-1]+bucket_count[i-1];
  }

  // copy tuples
  for (i = 0; i < relin->num_tuples; i++) {
    key = HASHFUNC(relin->tuples[i].id) & M;
    memcpy(&(relout->tuples[bucket_offset[key]++]), &(relin->tuples[i]), sizeof(tuple_t));
  }

  ...
}
```

Radix Cluster Algorithm: Join

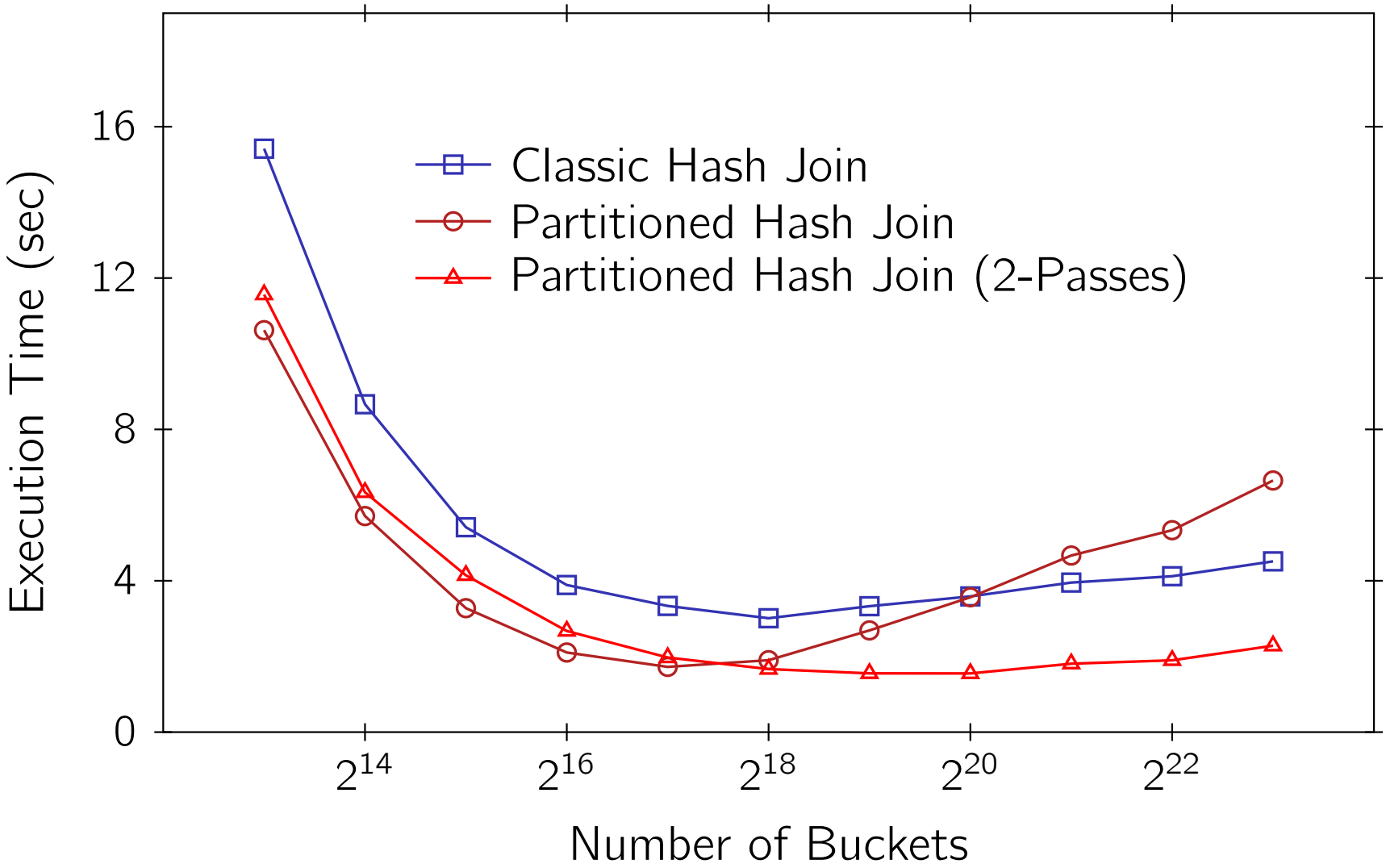
```
int radix_join(relation_t *R, relation_t *S, uint32_t D) {
  ...

  // mask for radix bits
  M = ((1 << D) - 1);

  // number of buckets
  num_buckets = (1 << D);

  // count elements in each partition
  for (i = 0; i < R->num_tuples; i++) {
    key = HASHFUNC(R->tuples[i].id) & M;
    cnt_elements_per_bucket[key]++;
  }

  // compute start index of every bucket
  for(i = 1; i < num_buckets; i++) {
    bucket_offset[i] = bucket_offset[i-1]+cnt_elements_per_bucket[i-1];
  }

  // copy tuples
  result = 0;
  for (i = 0; i < S->num_tuples; i++) {
    key = HASHFUNC(S->tuples[i].id) & M;
    for(j=bucket_offset[key];j<(bucket_offset[key]+cnt_elements_per_bucket[key]);j++) {
      if(R->tuples[j].id == S->tuples[i].id) {
        result++;
      }
    }
  }
  return result;
}
```

Radix Cluster Algorithm: Putting it all Together

```
if(RADIXBITS < 7) {
  ...
} else if(RADIXBITS < 13) {
  ...
} else if(RADIXBITS < 19) {

  // partition R in three passes
  radix_cluster(R, &reloutR,  6,              0);
  radix_cluster(&reloutR, R,  6,              6);
  radix_cluster(R, &reloutR,  (RADIXBITS-12), 12);

  // partition S in three passes
  radix_cluster(S, &reloutS,  6,              0);
  radix_cluster(&reloutS, S,  6,              6);
  radix_cluster(S, &reloutS,  (RADIXBITS-12), 12);

  // join
  result = radix_join(&reloutR, &reloutS, RADIXBITS);

} else if(RADIXBITS < 25) {
  ...
} else {
  ...
}
```

All Results

Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB



All Results

Relation $R = S = 8{,}000{,}000 \rightarrow 61.035$ MiB