Data Processing on Modern Hardware

Tutorial 5

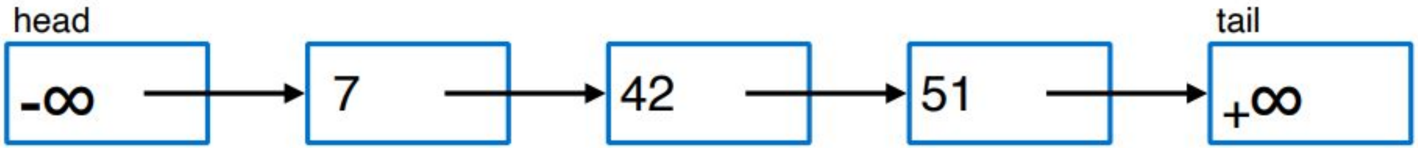
Ferdinand Gruber

Michalis Georgoulakis

Assignment 5 - Synchronization

Assignment 5 - Synchronization

- Databases are often faced with highly concurrent workloads
- Hardware offers us parallelization opportunity in multiple cores
- Synchronization :'(

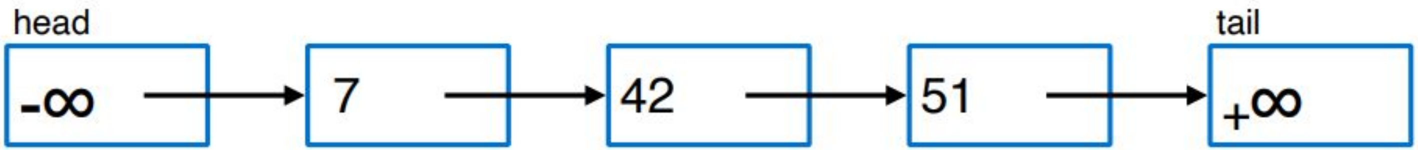


Assignment 5 - Synchronization

- **Goal**: synchronize access on a list-based set
  - sorted
  - no duplicates
- Supported Operations

```
bool contains(T k) { return false; }
void insert(T k) { }
void remove(T k) { }
```

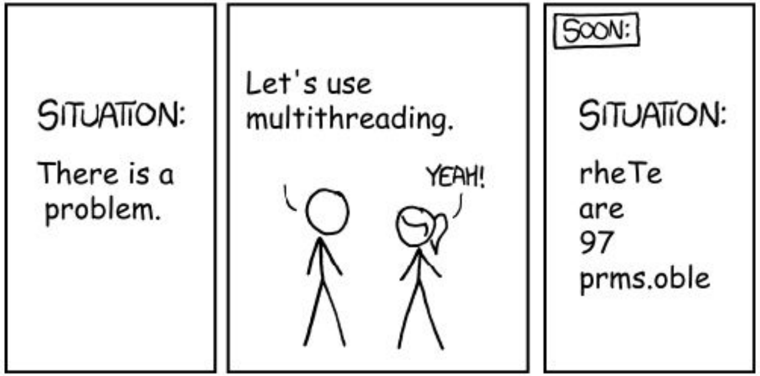
- We give you a baseline implementation without synchronization



Assignment 5 - Synchronization

You give us implementations of the three operations with the following approaches

- Coarse-Grained Locking
- Coarse-Grained Locking with Read/Write Locks
- Lock Coupling
- Lock Coupling with Read/Write Locks
- Optimistic Locking
- **Bonus**: Optimistic Lock Coupling

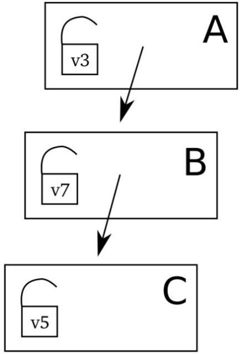


Assignment 5 - Synchronization

Optimistic Lock Coupling

traditional

1. lock node A
2. access node A
3. lock node B
4. unlock node A
5. access node B
6. lock node C
7. unlock node B
8. access node C
9. unlock node C



optimistic

1. read version v3
2. access node A
3. read version v7
4. validate version v3
5. access node B
6. read version v5
7. validate version v7
8. access node C
9. validate version v5

Figure 1: Comparison of a lookup operation in a 3-level tree using traditional lock coupling (left-hand side) vs. optimistic lock coupling (right-hand side).

```
struct Entry {
  T key;
  std::atomic<Entry *> next;
  std::atomic<uint64_t> version = 0; // Version counter for the lock coupling
  M mutex;                           // Mutex for each element in the list
};
```

Assignment 5 - Synchronization

Thread Management tools

C++

- **TBB Library =>** provides abstractions, easier to use (`tbb::spin_mutex`, `tbb::spin_rw_mutex`)
  - Intel library with high-level abstractions for parallelism. It abstracts the complexity of thread management and allows developers to focus on parallel algorithms.
  - **TBB** creates a pool of system threads to execute tasks concurrently (`tbb::task_arena`).

C

- **Threads (POSIX)** => more control, but also more manual management and synchronization (`pthread_create` / `pthread_join`)

Assignment 5 - Synchronization

Mutexes

- Synchronization primitives to protect shared data structures

Lock (exclusive): prevents other threads from acquiring the same mutex until it is released.

Shared Lock: provides shared access to the resource while preventing exclusive write access by other threads

Scoped Lock: allows for automatic locking and unlocking of multiple mutexes in a scoped manner.

Mutex Flavors
`tbb::spin_mutex` does not scale well, but can be very fast in lightly contended situations

Assignment 5 - Synchronization

Analysis

- How expensive is locking for the different approaches?
- How do the approaches perform in regard to #clock-cycles, #instructions and IPC?
- Which approach provides the best performance?

Help: “bench” function in **main.cpp**

Workload Types

- Read-only workload (contains)
- Mixed workload (insert, update, and contains)

Vary workload type, # of threads, domain name.

Assignment 5 - Synchronization

Deadline: **11/06**

Submission Instructions

- First fork the assignment repository.
- Then, in your forked repository, add:
  - Code that implements the assignment
  - A 1-page report answering the assignment questions (**report.pdf**)

Before you get started, you can have a look at:

- TBB Parallel Processing Example
- Optimistic Lock Coupling Paper
- https://databasearchitects.blogspot.com/2020/10/c-concurrency-model-on-x86-for-dummies.html

Assignment 3 - Hardware Optimized Hash Joins

Sample Answers

Assignment 3

1. Hash Join Baseline

```
// Assume relation r is smaller, return number of matched tuples
uint64_t hash_join(relation &r, relation &s) {
  uint64_t matches = 0;
  std::unordered_set<keyType> hashTable;
  // Step 1: build phases
  for (auto &t : r) {
    hashTable.insert(t.key);
  }

  // Step 2: probe phase
  for (auto &t : s) {
    matches += hashTable.count(t.key);
  }
  return matches;
}
```

Build Phase: Easily parallelizable

Probe Phase: Requires no synchronization

Data Structures:
We use unordered_set, unordered_map to group unique elements.

2.1 Naive Partitioning

Assignment 3

```
partition partition_naive(relation &r, size_t start, size_t end, uint8_t bits, uint8_t shift) {
  SplitHelper split(bits);
  // TODO implement the naive partitioning here

  // Step 1 build histograms -> Prefix sum
  // create a histogram with #entries = #partitions = fanOut
  std::vector<uint64_t> histogram(split.fanOut, 0);
  for (size_t i = start; i < end; i++) {
    auto bucket = (r[i].key >> shift) & split.mask;
    histogram[bucket]++;
  }

  // Step 2 use prefix sum to partition data
  std::vector<uint64_t> startPositions(split.fanOut, 0);
  uint64_t prefixSum = 0;
  for (size_t i = 0; i < split.fanOut; i++) {
    startPositions[i] = prefixSum;
    prefixSum += histogram[i];
  }

  // Step 3 partition
  relation partitionedRelation(prefixSum);
  std::vector<uint64_t> offset(split.fanOut, 0);
  for (size_t i = start; i < end; i++) {
    auto bucket = (r[i].key >> shift) & split.mask;
    auto position = startPositions[bucket] + offset[bucket];
    partitionedRelation[position] = r[i];
    offset[bucket]++;
  }

  return {partitionedRelation, startPositions};
```

1. Create histogram equal to the number of partitions.

2. Iterate over histogram to calculate starting position of each partition in the partitioned relation.

3. Bucket number + offset determine the position where each relation should be placed.

Assignment 3

2.2 Multi-pass Partitioning

```
partition partition_multiPass(relation &r, uint8_t bits1, uint8_t bits2) {
  // Partition 1. stage
  partition p1 = partition_naive(r, bits1);

  // Partition 2. stage
  relation result;
  std::vector<uint64_t> startPositions;
  uint64_t offset = 0;

  for (uint64_t i = 0, limit = p1.s.size(); i < limit; i++) {
    auto start = p1.s[i];
    auto end = (i == limit - 1) ? p1.r.size() : p1.s[i + 1];

    auto p2 = partition_naive(p1.r, start, end, bits2, bits1);

    for (auto &x : p2.r) {
      result.push_back(x);
    }
    for (auto &x : p2.s) {
      startPositions.push_back(offset + x);
    }
    offset += end - start;
  }

  return {result, startPositions};
}
```

Idea: Creating too many partitions can easily thrash the TLB cache.

- Each partition requires its own TLB entries.

Motivation: Splitting partitioning into two phases can reduce the fan-out of each stage.

Assignment 3

2.3 Software-managed Buffers & Non-Temporal Writes

```
std::pair<partition, std::vector<uint64_t>> partition_softwareManaged(relation &r, uint8_t bits) {
  SplitHelper split(bits);

  // Step 1 build histograms -> Prefix sum
  std::vector<uint64_t> histogram(split.fanOut, 0);
  for (auto &t : r) {
    auto bucket = t.key & split.mask;
    histogram[bucket]++;
  }

  // Step 2 use prefix sum to partition data
  std::vector<uint64_t> startPositions(split.fanOut, 0);
  uint64_t prefixSum = 0;
  for (size_t i = 0; i < split.fanOut; i++) {
    startPositions[i] = prefixSum;
    auto tmp = histogram[i];
    if (tmp % tuplesPerCL != 0) {
      // align the offsets for non-temporal writes
      tmp += tuplesPerCL - (tmp % tuplesPerCL);
    }
    prefixSum += tmp; // update the offset of the next bucket
  }
```

Ensure each partition size is a **multiple of cache line size.**

Assignment 3

2.3 Software-managed Buffers & Non-Temporal Writes

```
// Step 3 partition
relation partitionedRelation(prefixSum);
std::vector<uint64_t> offset(split.fanOut, 0);
// initialize a software managed buffer per partition and its current offsets
std::vector<SoftwareManagedBuffer> softwareManagedBuffers(split.fanOut);
std::vector<uint64_t> bufferOffsets(split.fanOut, 0);

for (size_t i = 0, limit = r.size(); i < limit; i++) {
  auto bucket = r[i].key & split.mask;

  auto &buffer = softwareManagedBuffers[bucket];
  auto &position = bufferOffsets[bucket];

  buffer.tuples[position] = r[i];
  position++;

  // perform non temporal write when the buffer is full
  if (position == tuplesPerCL) {
    auto outPosition = &partitionedRelation[startPositions[bucket] + offset[bucket]];
    auto writePtr = reinterpret_cast<uint8_t *>(outPosition);
    storeNontemp(writePtr, &buffer);
    offset[bucket] += tuplesPerCL;
    position = 0;
  }
}
```

Create and populate software-managed buffers for each partition.

When buffer is full, non-temporal write to memory.

```cpp
  // handle non-empty buffers
  for (uint64_t i = 0, limit = split.fanOut; i < limit; i++) {
    auto &buffer = softwareManagedBuffers[i];
    auto &position = bufferOffsets[i];

    auto outPosition = startPositions[i] + offset[i];
    for (size_t j = 0; j < position; j++) {
      partitionedRelation[outPosition + j] = buffer.tuples[j];
    }
    offset[i] += position;
  }

  partition result = {partitionedRelation, startPositions};
  return std::make_pair(result, offset);
}
```

Handle non-empty buffers.

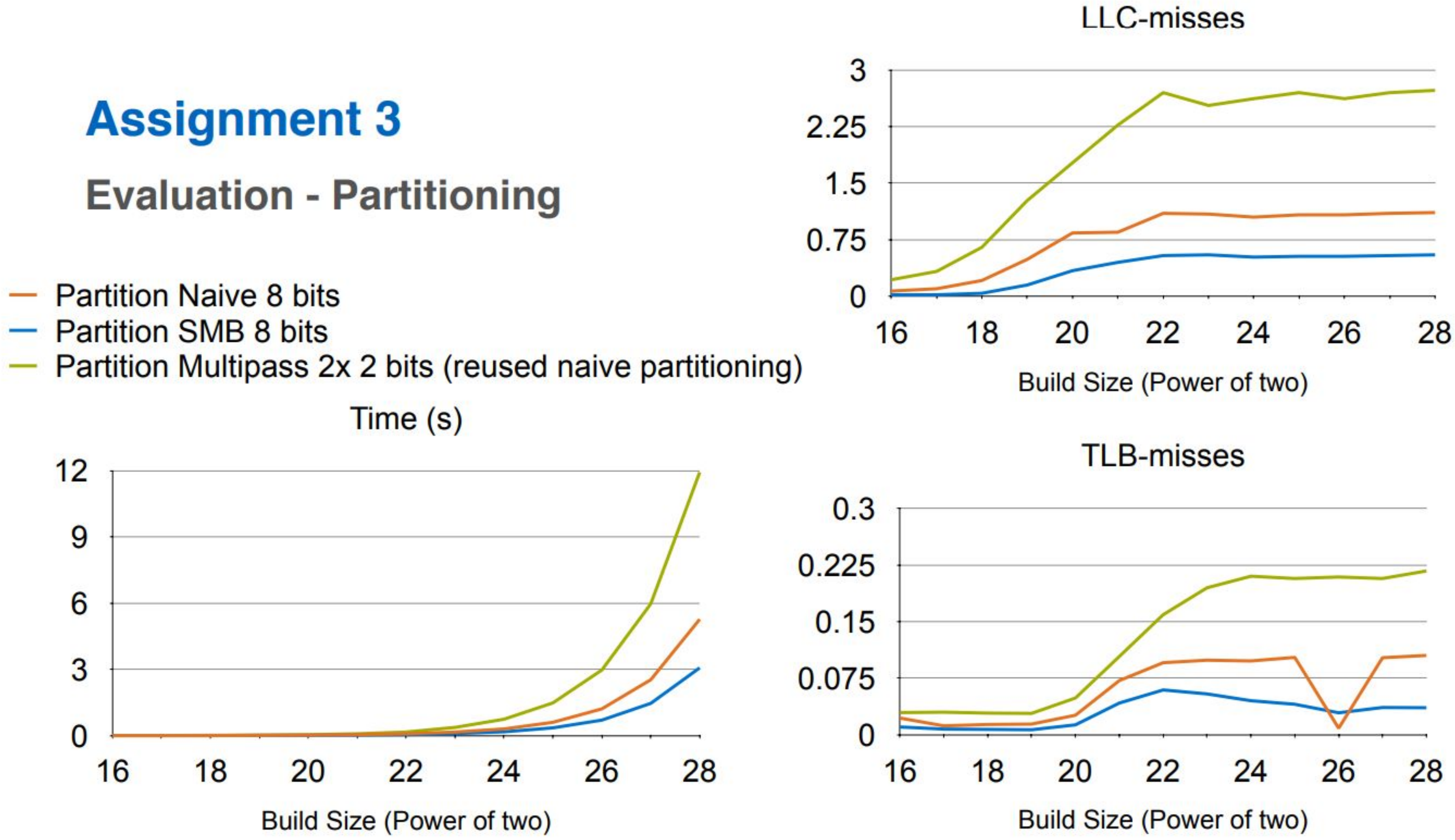
Assignment 3

3 Radix Join

```cpp
// Assume relation r is smaller, return number of matched tuples
uint64_t radix_join(relation &r, relation &s, Partitioning p) {

  uint64_t matches = 0;

  // Step 1: partitioning phase
  partition pR{};
  partition pS{};

  std::vector<uint64_t> pROffsets;
  std::vector<uint64_t> pSOffsets;

  switch (p) {
  case Partitioning::naive:
    pR = partition_naive(r, 8);
    pS = partition_naive(s, 8);
    break;
  case Partitioning::multiPass:
    pR = partition_multiPass(r, 2, 2);
    pS = partition_multiPass(s, 2, 2);
    break;
  case Partitioning::softwareManaged:
    std::tie(pR, pROffsets) = partition_softwareManaged(r, 8);
    std::tie(pS, pSOffsets) = partition_softwareManaged(s, 8);
    break;
  }

  // Step 2: partition-wise build & probe phase
  for (uint64_t i = 0, limit = pR.s.size(); i < limit; i++) {
    std::unordered_set<keyType> hashTable;
    // Step 1: build phases
    auto start_r = pR.s[i];
    auto end_r = i == limit - 1 ? pR.r.size() : pR.s[i + 1];
    auto start_s = pS.s[i];
    auto end_s = i == limit - 1 ? pS.r.size() : pS.s[i + 1];

    if (p == Partitioning::softwareManaged) {
      end_r = std::min(end_r, start_r + pROffsets[i]);
      end_s = std::min(end_s, start_s + pSOffsets[i]);
    }
    for (uint64_t j = start_r; j < end_r; j++) {
      hashTable.insert(pR.r[j].key);
    }

    // Step 2: probe phase
    for (uint64_t j = start_s; j < end_s; j++) {
      matches += hashTable.count(pS.r[j].key);
    }
  }
  return matches;
}
```

Assignment 3

Evaluation - Partitioning



Assignment 3 - Partitioning Evaluation

Multi-Pass Partitioning (Green Line)

- Performs two full rounds of naive partitioning (coarse then fine), so it touches the entire dataset twice.
- As a result, it incurs double the histogram/build overhead and double the scatter traffic, causing the **longest overall runtime**.

Software-Managed Buffers (SMB, Blue Line): Buffers tuples in small, cache-line-sized chunks before writing them out. Keeps each 64-byte buffer hot in L1 while it fills. Uses non-temporal (streaming) stores to write full cache lines straight to memory, bypassing L1/L2.

- **Benefits**:
  - Far fewer cache-line evictions and write-allocate misses
  - Lower LLC and TLB miss rates (especially once data approaches LLC size)

- **Trade-Off**:
  - Slight branch-misprediction overhead for “is buffer full?” checks
  - More bookkeeping for per-partition counters and buffer flushes

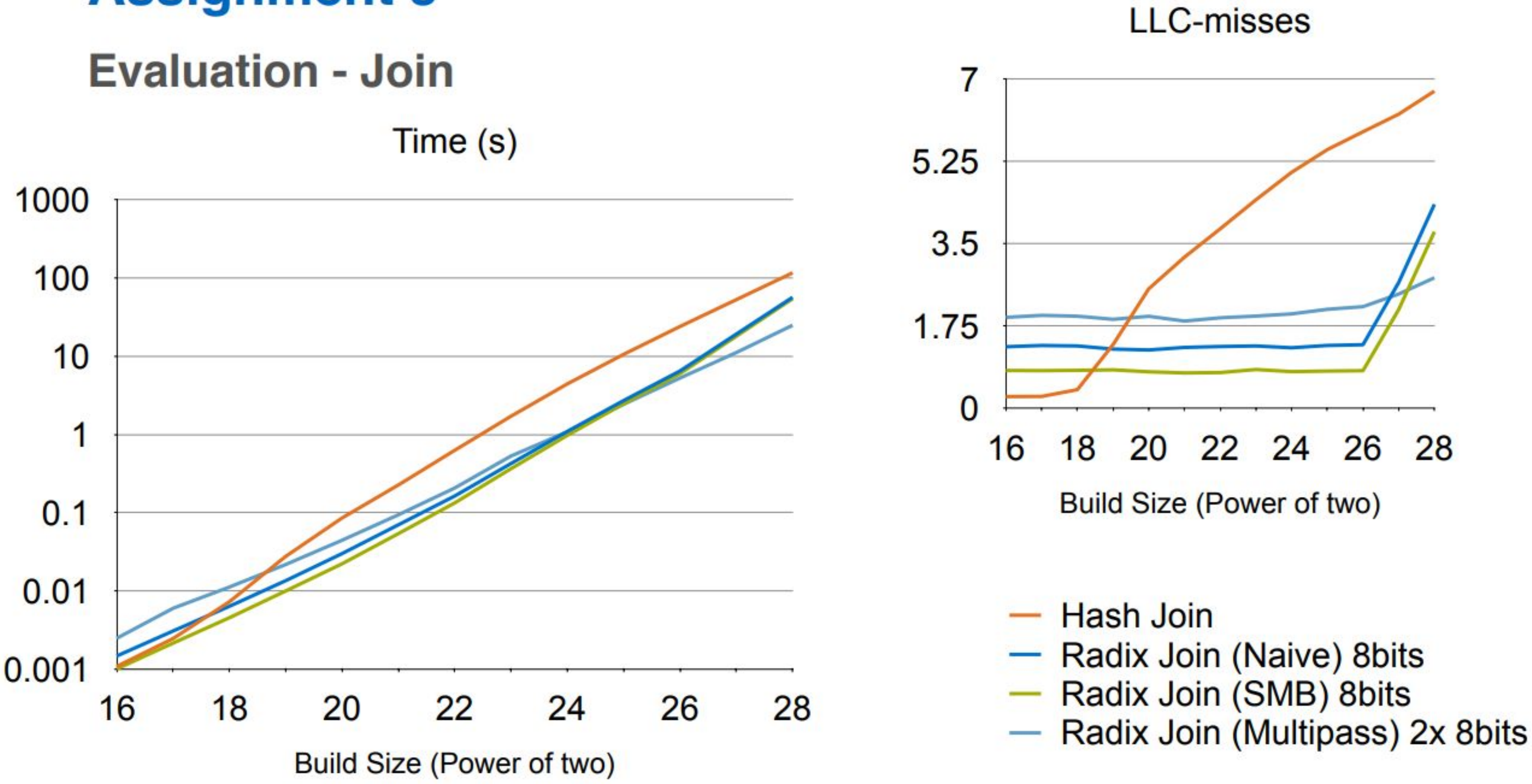
Why SMB Beats Naïve Once Data Exceeds Cache

- Around 2^{20} – 2^{21} (~1 – 2 million tuples ⇒ ~8 MB), working set nears our LLC (14 MB).
- **Naïve** scatter writes then forces repeated write-allocate and eviction of cached data, spiking LLC and TLB misses.
- **SMB** keeps write traffic in full 64-byte chunks, cutting LLC misses → smoother, lower-latency writes.
- Consequently, SMB’s overall time curve remains below naïve once you exceed L2/L3 boundaries.

Assignment 3

LLC-size: 14 MiB

Evaluation - Join



Assignment 3 - Radix Join Evaluation

Hash Join: Fastest when the entire build side fits in cache (up to ~2^{18} tuples ≈ 256 KB)

- No partitioning overhead—random hashtable lookups stay in L3.
- Once |R| > 2^{18}, LLC misses spike sharply as the hashtable no longer fits on-chip.

Radix Join (Partition-then-Join): Partitions keep each bucket cache-resident until |R| ≈ 2^{26} (≈ 64 MB)

- The “knee” in LLC misses appears around 2^{18} + #bits (i.e. 2^{26} for 8-bit partitions).
- Below that, each bucket fits in L3/L2, so build+probe per bucket generates very few LLC misses.

Partitioning Strategies

1. **Naïve (One-Pass):** Good once |R| > 2^{18}; but as soon as partitions themselves exceed L3 (≈ 2^{26}), LLC misses climb again.
2. **SMB (Software-Managed Buffers): Best in the mid-range (2^{18} … 2^{26})**
   - Small, cache-line-sized write buffers reduce write-allocate traffic and keep LLC/TLB misses lowest.
   - After |R| ≈ 2^{26}, partitions overflow L3 and LLC misses grow, but still below naïve one-pass.
3. **Multi-Pass (Two-Stage)**
   - **Only advantageous at the very largest sizes (≥ 2^{27})**, because re-splitting partitions delays bucket overflow.
   - Before that point, the extra scan/overhead outweighs the benefit—LLC misses stay low but total time is higher than SMB.

Questions?