Modeling Multithreaded Query Execution on Chip Multiprocessors

Konstantinos Krikellas [‡]
Greenplum Inc.
kkrik@greenplum.com

Stratis D. Viglas
School of Informatics
University of Edinburgh
sviglas@inf.ed.ac.uk

Marcelo Cintra
School of Informatics
University of Edinburgh
mc@inf.ed.ac.uk

ABSTRACT

Modern CPUs follow multicore designs with multiple threads running in parallel. The dataflow of query processing algorithms needs to be adapted to exploit such designs. We identify memory accesses and thread synchronization as the main bottlenecks in a multicore execution environment. We present a uniform framework to mitigate the impact of these bottlenecks in multithreaded versions of the most frequently used query processing algorithms, namely sorting, partitioning, join evaluation, and aggregation. We analytically model the expected performance and scalability of the proposed algorithms. We conduct an extensive experimental analysis of both the analytical model and the algorithms. Our results show that: (*a*) the analytical model adequately captures the performance of the algorithms, and (*b*) the algorithms themselves achieve considerable speedups compared to their single-threaded counterparts.

1. INTRODUCTION

This paper presents a detailed analysis of multithreaded query execution on multicore processors. Extending the elementary query evaluation operators for multithreaded processing is far from straightforward. Multithreading introduces resource contention that penalizes scalability; cores share resources both at the hardware (caches and physical memory) and at the software (lock-based synchronization) levels, thereby restricting the degree of parallelism. To counter that we posit that multiple threads should independently process cache-resident data to the highest possible extent, thereby minimizing contention and enhancing parallelism. To that end we: (*a*) propose a uniform framework to generalize the most frequently used query processing algorithms for multithreaded execution, and (*b*) present an analytical model to estimate the multithreaded performance of the proposed algorithms. The model statically estimates the speedup of multithreaded execution. To the best of our knowledge, this is the first paper that provides a uniform framework for and an analytical performance model of multithreaded query execution on chip multiprocessors.

Multicore means shared memory. Modern CPUs integrate multiple cores and provide hardware support for parallel processing. Their architecture resembles shared-memory systems: the cores share main memory and, possibly, the lowest level of the cache hierarchy. Query evaluation on this type of parallel systems has been tackled before (*e.g.*, [9]); previous work, however, has not taken into consideration the cache hierarchy and its impact on multithreaded execution. As shown in [1, 14, 16], database workloads suffer from excessive stalls due to the high latency of memory operations. This is aggravated in multicore processors as the memory subsystem serves requests from multiple cores [18, 23].

Busier is faster. Multicore processors have more "raw" processing power, which is not harvested when executing data-intensive workloads. To alleviate this, we propose to exploit cache locality by maximizing the amount of processing whenever a data block is in the CPU caches. For example, "pushing" more query-relevant processing into partitioning an input may result in an extra per-thread processing cost of 13%; however, the cores are now busier processing instead of waiting for memory operations. The busier a core is with processing cache-resident data, the less it contends with the other cores for accessing the memory. The extra per-thread cost in the previous example results in an almost three-fold improvement in the Cycles Per Instruction (CPI) ratio when the technique is applied to a quad-core Intel Xeon E5420 CPU. In turn, this speeds up the execution of the entire query.

We apply this approach to the prominent query evaluation algorithms and provide a uniform framework for multithreaded processing. Our goals are to: (*a*) minimize data transfers from main memory, and (*b*) evenly distribute both work and data across multiple threads. To minimize synchronization overhead we assign different input and output streams to each thread; locking (if any) is performed on a coarse granularity, thus aiding parallel execution.

Modeling scalability. To assess multithreaded execution, we analytically model the effect of input cardinality, tuple size, selectivity, and projectivity to performance, according to the characteristics of the host hardware. We introduce the *multithreaded utility ratio*: the ratio of the time spent for fetching each input unit to its total processing time. High values of the ratio denote fetch-dominated operations; in this case memory accesses incur an inflated *effective* cost, thus restricting scalability. Conversely, low values of the utility ratio show that there is sufficient computational load to overlap with data fetching, so thread contention for memory accesses is limited and scalability is enhanced. Using this ratio, we analytically estimate the query processing cost and the expected speedup of multithreaded execution.

Contributions. The main contributions of this work can be outlined as follows:

- We give a uniform framework to extend existing query processing algorithms for multithreaded execution on multicore CPUs.
- We present partitioning and buffering techniques that

[‡] Work done while author was at the University of Edinburgh.

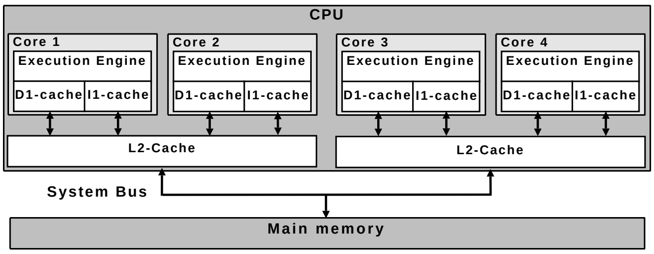


Figure 1: The architecture of the Intel Xeon E5420

determine which part of the input each thread processes and where in the memory hierarchy it is buffered.

- We introduce an analytical model to accurately estimate the speedup of multithreaded query execution.

The rest of this paper is organized as follows: in Section 2 we present the main characteristics of multicore CPUs. In Section 3 we give a general framework for multithreaded execution and algorithms for the main query processing operations. We analytically model the proposed algorithms in Section 4, while in Section 5 we conduct an experimental study of our proposals. We present related work in Section 6 and conclude in Section 7.

2. CHIP MULTIPROCESSORS

During the past decade, the dominant trend in processor design is the integration of multiple processing cores on the same die. Termed *chip multiprocessors* (CMPs), multicore chips natively support parallel execution, while combining scalability with energy efficiency [11]. Multicore chips have been implemented in various ways. The main difference is the type of parallelism supported by each core. Some processor designs, *e.g.*, the Intel Quad Core and the AMD Phenom, support out-of-order execution and Instruction-Level Parallelism (ILP); alternatively, the pipelines of the Sun UltraSPARC T2 and the IBM Power 6 support only in-order execution but use Thread-Level Parallelism (TLP). There are also hybrid designs, *e.g.*, the Intel Core i7 CPU, which combine out-of-order execution with hardware supported multithreading, similar to Simultaneous Multithreading. A detailed analysis of design trends in processor architecture and their effect on the execution of OLTP and DSS workloads, can be found in [10].

Designs also differ in terms of the memory hierarchy, specifically whether on-chip caches are shared between all or some of the cores. In Figure 1 we sketch the Intel Xeon E5420 quad-core processor: each pair of cores shares a common L2-cache and cores from different pairs communicate through the memory bus. In other designs, *e.g.*, the AMD Phenom and the Intel Core i7, each core has its own L1- and L2-caches, while all cores share a common on-chip L3-cache. The salient challenge in multicore CPUs is to keep all cores processing data at rates close to their clock. To do so, manufacturers improve memory throughput by integrating memory controllers inside the chip and using multiple memory banks. Still, if the caches and the memory are concurrently accessed by all cores, contention for their utilization may increase the latency of memory operations and degrade performance.

As multiple cores share main memory but not necessarily individual caches, it is common practice to replicate data inside the caches of different cores to enhance parallelism. Cache coherency involves the propagation of data writes from one core to the others. Caches are organized in small blocks termed *cache lines*. When one cache line is shared between cores and is updated by one of them, the other cores invalidate their cached copy and refetch the cache line on the next access. Invalidation takes place on true sharing, *i.e.*, cores access the same data of the cache line, or on false sharing, *i.e.*, when one core updates a part of the cache line that no other core accesses. Coherency protocols "snoop" updates to all cores or use directories to maintain data sharing information [11].

Concurrent execution at the hardware level (*i.e.*, processing independently scheduled threads) does not imply synchronization at the software level. The latter is achieved by providing hardware support for atomic operations through mutexes and *spin locking*. Each mutex is a memory word set to 0 when free and 1 when locked; to operate on the mutex, a core must have it in its D1-cache. To acquire a lock, a core continuously probes the mutex (*i.e.*, the core "spins") using the *compare-and-swap* instruction. Once the lock is acquired the core executes the synchronized code and resets the lock. Each core spins on a locally cached copy of the mutex without affecting other cores. Whenever the mutex is released, cache coherency requires that the cache line containing it be invalidated and refetched. The first core to refetch the cache line will acquire the lock; other cores waiting on the lock will continue to spin.

3. MULTITHREADED PROCESSING

We now provide a framework for parallelizing the most frequently used query processing algorithms [8]: sorting, partitioning, join evaluation, and aggregation. To that end:

- We use the N-ary Storage Model (NSM) with tuples stored consecutively within pages of 4kB. Each table resides in its own file on disk; a storage manager is responsible for caching file pages in the buffer pool. We do not use vertical partitioning as we want to keep the same baseline with most commercial and research database systems. We also want to explicitly account for the interaction between the query engine and the storage manager in our analysis.
- Our techniques only depend on the number of threads that can be efficiently supported by hardware. Naturally, the techniques need to be "fitted" to a specific CPU but the approach is uniform and remains largely the same across CPUs. For instance, the Intel Xeon 5400 series of quad-core processors of Figure 1 (the one also used in our experiments) has per-core pipelines supporting out-of-order execution. However, there is no in-core support for TLP so only four concurrent threads are supported by hardware. We will be pointing out any such subtleties that require fitting the data flow to each CPU.

Our approach stems from the observation that CMPs are in essence shared-memory systems. Parallel query evaluation has been tackled before [7, 9]; the rule of thumb is to split the input in disjoint partitions and then process them in parallel. However, the naïve extension of this technique for multicores would not take advantage of the cache hierarchy's buffering effect. For example, synchronizing accesses to a shared hash table would severely penalize performance in case the table does not fit inside caches [3]. Thus,

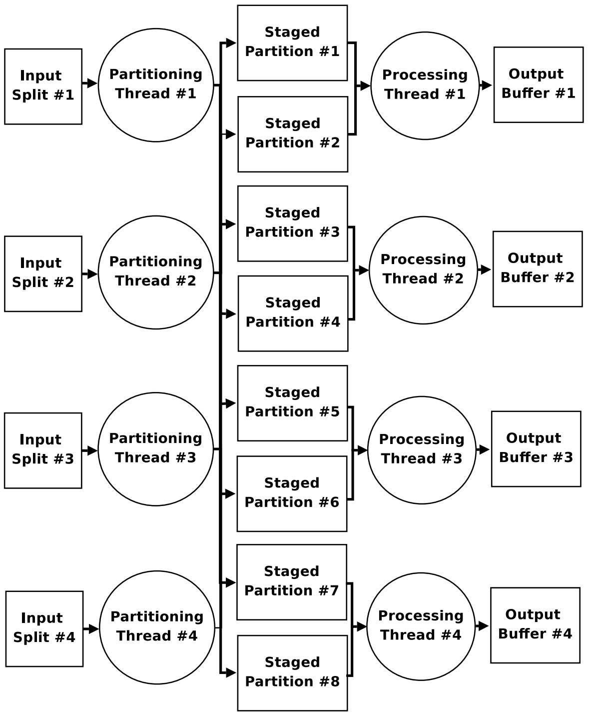


Figure 2: Multithreaded operator implementation

we fine-tune the implementation of partitioning and parallel processing to the characteristics of multicore processors. We focus on reducing concurrent memory requests by interleaving memory accesses and cached data processing to the highest possible extent. This technique keeps the cores busy and reduces memory stalls. We also avoid using fine-grained thread synchronization. Threads are initialized once for each operation and use *restricted affinity* (*i.e.*, they are assigned to a specific core); that way they can run with the minimum synchronization overhead. Finally, we pay special attention to avoid false sharing: we align shared data (such as mutexes) with the size of the cache line and replicate writeable variables and buffers for each thread.

An example of the uniform framework for the implementation of each operator is shown in Figure 2. The input is first split in as many "splits" as there are threads of execution that can be efficiently supported by hardware (*e.g.*, four splits for the Intel Xeon 5400, eight splits for the Intel Core i7). For each primary table we divide the total page count by the number of threads; each split is assigned to one thread. Next, we partition the input in disjoint partitions using the specified number of threads. Each thread scans its split and writes tuples to appropriate output partitions. We do not use tuple references, but copy to the partitions the fields required for further processing. That way we increase cache locality and avoid uncontrollable and costly random access patterns outside the cache hierarchy. After partitioning all inputs, we invoke a new team of threads to process the partitions. A set of disjoint partitions is assigned to each thread and processed with no synchronization overhead. Threads store output tuples to individually assigned output buffers. The set of all output buffers is the final operator output that will either be used by subsequent operators, or be forwarded to the client as a final result.

3.1 Data staging

During data staging selections and projections are applied and the input is appropriately "formatted". For example, for merge join, inputs are sorted, while for hash join the input is hash-partitioned. Our measurements have shown that data staging can take up to 90% of the total execution time of an operator. It is therefore important to adapt all common staging algorithms for multithreaded execution.

Our algorithms use partitioning for multithreaded processing with minimal overhead. The main algorithms are: (a) range partitioning, (b) hash partitioning, and (c) value mapping. Range partitioning generates partitions containing tuples within a specific range of values of the partitioning attribute. Value distribution statistics, *e.g.*, histograms, can be used to extract the bounds of each partition to balance the distribution of tuples to partitions. Hash partitioning uses hash and modulo computations to map tuples to partitions with no assumption on value distributions. This leads to similarly sized partitions. Finally, the values of the partitioning attribute can be directly mapped to partitions, a technique applicable if the partitioning attribute has only a few distinct values. We elaborate on each staging algorithm.

Sorting. We build on the AlphaSort algorithm [19], where input partitions fitting the cache hierarchy are sorted with *quicksort* and then merged through multi-way merging. We use N hardware-supported threads to sort partitions and assign $\left(\frac{1}{N}\right)^{\text{th}}$ of the total number of input pages to each thread. Each thread applies quicksort to partitions that fit inside its share of the lowest cache level. For example, in the Intel Xeon processor of Figure 1 the partition size is less than half the size of the L2-cache; for the AMD Phenom quad-core processor, where each core has its own L2-cache and shares the on-chip L3-cache, the partition size should be less than a quarter of the capacity of the L3-cache.

After sorting each partition we invoke N new threads to merge the partitions. We use range partitioning to separate work. We assign a specific range of values to each thread, as shown in Figure 3 (value ranges are individually shaded). Each thread processes only the part of each partition that contains values in its assigned range. The sorting threads specify the tuple range for each merging thread in each partition during the previous step. Through value statistics, it is possible to assign ranges to threads so that each thread will output approximately the same number of tuples. That way all threads will have comparable processing rates. Each merging thread maintains a heap of the currently examined tuples from each partition to identify the tuple with the minimum value. Note that no synchronization is needed during sorting since threads process disjoint datasets.

We tackle data skew using static and dynamic techniques. To assign value ranges to threads, the system exploits histograms and cardinality statistics to compute ranges that are estimated to create partitions of similar size. We further adopt a dynamic approach similar to the one presented in [13]. Threads are initially assigned a specific value range, assuming that each thread will approximately process $\frac{|K|}{N}$ tuples, where $|K|$ is the input cardinality. When a thread has processed $\left(\frac{|K|}{N} + \mathsf{thres}\right)$ tuples, where threshold thres is the expected overflow factor, the input is skewed, so other threads have already processed the tuples within their assigned value range. At that point, all threads join and the remaining input is redistributed to them. This process is

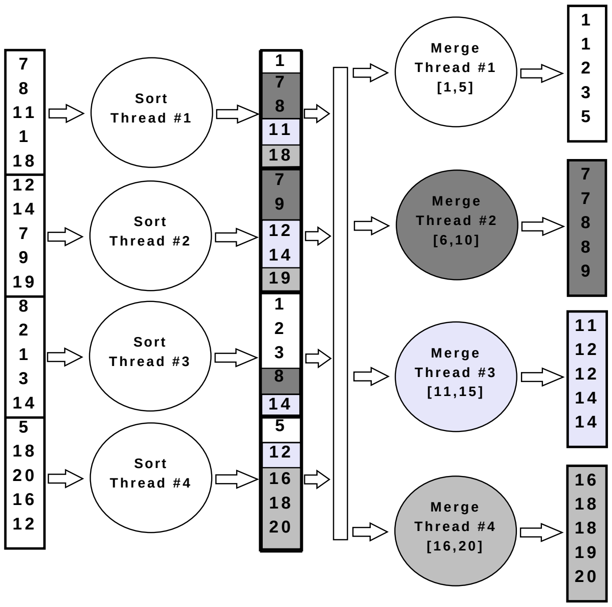


Figure 3: Multithreaded sorting

repeated until the input is entirely sorted.

Partitioning. Hash and range partitioning use the same multithreaded process, the difference being the function used to forward tuples to partitions. As shown in Figure 2, each thread scans its split of input pages and forwards tuples to partitions by applying a partitioning function. We use buffering on a page granularity, as each thread uses one page from each partition to store tuples. When a page fills up, the thread replaces it with a new one through a call to the storage manager.

This simple approach has two drawbacks. Firstly, storage manager interaction needs to be an atomic operation; thus, requests to the storage manager need to be serialized. Secondly, and more importantly, the only per-tuple processing is the evaluation of the partitioning function. This requires at most a few tens of CPU cycles, while fetching data from main memory costs an order of magnitude more. Since memory is a shared resource across all cores, if multiple cores issue memory requests concurrently, memory operations will be queued [18] and their *effective* latency will increase; this restricts the scalability of multithreaded partitioning. We have verified this hypothesis for the Intel Xeon 5400 processor, which uses a single memory bus, but it is likely to hold for processors with multiple embedded memory controllers.

The solution we propose is to maximize reuse by processing the input to a greater extent once it is cache-resident. One way of doing so is sorting each full partition page before replacing it with a new page. That way, the partition page is prepared to be further processed at a negligible cost. If the number of partitions is moderate we can expect the page to be inside the L2-cache (or even the L1-cache) before being sorted, thus sorting is performed efficiently. Since the partitions end up containing sorted pages, one merging phase per partition is needed to sort it. This step can be integrated with query evaluation, as we shall see in Sections 3.2 and 3.3. This technique resembles the *MapReduce* algorithm [6]; we combine partitioning with page sorting to better adapt execution to the characteristics of CMPs.

To quantify the difference between partitioning alone, as used in previous work on parallel DBMSs, and the proposed integration of page sorting to partitioning, we compare the results of hardware profiling for these two techniques on the CPU of Figure 1 in Table 1.[1] The input table has 1M tuples of 72 bytes each. The overhead of partitioning the input while sorting each partition page in single-threaded execution is 74% over partitioning the input alone, but is reduced to 13% when four threads are used. Furthermore, though in both cases the L2-cache misses increase (due to the interaction with the storage manager and thread synchronization), simple multithreaded partitioning increases the CPI ratio by a factor of 2.3 and the number of pending memory requests by a factor of 2.4; combined partitioning and sorting results in a slight increase of a factor of 1.2 for the CPI ratio and the pending requests. The above show that, though the same dataset is accessed in both cases, the cores need to wait longer for memory operations in hash partitioning alone because they all attempt to access main memory at the same time. When combining partitioning with sorting, while one core is busy sorting a page, the remaining cores face less contention for memory operations. Synchronization overhead is also reduced as the time to obtain a reference to a new page from the storage manager is only a small portion of the time to fetch a page and sort it.

Listing 1: Accessing the mapping directory

```
int offset = lookup(directory, value);
if (offset < 0) {
  lock(directory.lock);
  offset = lookup(directory, value);
  if (offset < 0) offset = insert(directory, value);
  unlock(directory.lock);
}
```

Algorithm	Threads	Time	CPI	L2-cache misses	Pending requests
Partition	1	0.085s	1.68	335	4672
	4	0.072s	3.86	699	11086
Partition and Sort	1	0.148s	1.21	342	7556
	4	0.083s	1.41	661	9008

Table 1: Profiling results for partitioning

Value mapping. If the partitioning attribute has a small number of distinct values, one can map each value to a specific partition, using a directory to maintain this mapping. We use a sorted array of attribute values and perform binary search for lookups. Hash-based solutions are also possible; we preferred binary search to avoid the effect of data skew in a data structure that is heavily used. Note that there is a limit beyond which this approach becomes inefficient: if the partitioning attribute has a high distinct cardinality the mapping directory will span outside the L1-cache and accesses will trigger cache misses.

Each thread scans its assigned input split and copies its tuples to the corresponding partitions. Since tuple processing requires a directory lookup (and may trigger an insertion), there is sufficient computational load to overlap with memory operations, resulting in considerable speedups. The more entries the directory has, the closer to linear the speedup will be: the time spent on lookups dominates the cost of fetching data. Note that since the number of distinct values is small, all cores share the same directory. In Listing 1 we show the code to synchronize directory insertions and lookups. The synchronization penalty is paid until the directory contains all entries. From then on threads replicate the directory inside each core's L1-cache and perform lookups without locking it.

---

[1] We show sample counts for L2-cache misses and pending memory requests extracted with the OProfile tool [20].

3.2 Join evaluation

Merge join. The input tables are staged by sorting them on the join attributes. After sorting the input tables, we initialize a new set of threads to evaluate the join predicate. Each thread processes a specific value range of the join attribute and evaluates the join for corresponding partitions; there is also a separate output buffer per thread. Data skew is treated using the techniques for merging sorted partitions of Section 3.1. As partitions are disjoint there is no synchronization overhead. The only performance restriction is the ability of the memory subsystem to provide the cores with data in the rates the threads consume them.

Hash join. Recall that during hash partitioning each page of each partition is also sorted. Thus, there is no need to build per-partition hash tables during the join phase. Each input is partitioned using a fanout wide enough for the largest corresponding partitions of each table to fit in the lowest cache level. For example, if we join table A of size 100MB with table B of size 250MB using four threads on a quad-core processor with a shared 8MB L2-cache (and no L3-cache), the partitions of both tables should be smaller than 1MB: during the join phase the threads sharing the L2-cache will be joining two partitions each. Thus, we use a fanout of at least 250 for both tables (*i.e.*, the size of the largest table over the target size of each partition). In practice, it is better to use higher fanouts (even double). Doing so will amortize the variance in partition sizes, and procure for space to hold instructions and data belonging to the operating system and the storage manager, as well as the merging buffers that will be shortly introduced.

After partitioning the inputs and individually sorting the partition pages, we start new threads to join the corresponding partitions. Each thread processes a disjoint set of partitions, so all threads work independently. To address data skew, partitions are allocated to threads so their combined size is approximately equal for each thread. The first step is to merge the pages of each partition and generate a fully sorted partition. As this is repeated for all partitions, we dedicate a single output buffer per thread and we (re)use it to store the tuples of each partition in sorted order. Since the partition size is small, one can expect the merging buffers for all threads to remain inside the lower cache level during the join process, thus avoiding accesses to main memory. After merging, we join corresponding partitions just as in merge join. Note that the partitions have already been brought in the lowest cache level so this step is efficient. Our hybrid join technique interleaves computation with memory operations and efficiently exploits the cache hierarchy; at the same time it incurs negligible synchronization overhead.

Map join. If the join attributes have a small number of distinct values we stage the inputs using value mapping. We then join the partitions for the same attribute value with nested loops join. Map join is applicable only if both inputs have a small distinct value cardinality. Its performance degrades as more entries appear in the mapping directory: as the directory grows it will not fit in the L1-cache, so lookups trigger cache misses.

3.3 Aggregation algorithms

Sort aggregation. We first sort blocks of the input on the grouping attributes. In line with performing as much computation as we can during data staging, we modify the merging phase of Section 3.1 to incorporate the on-the-fly evaluation of the aggregate functions. That way, we avoid flushing the sorted output to memory and refetching it to the caches to compute the aggregate values of each group. Doing so reduces main memory accesses and enhances parallelism.

$R.a$

value	*id*
x	0
y	1
z	2

$R.b$

value	*id*
A	0
B	1
C	2

$R.c$

value	*id*
10	0
20	1
30	2
40	3

(a) Multiple mapping directories

$$\begin{aligned}&\text{Offset}(R.a = y, R.b = \text{C}, R.c = 20)\\ &= \; R.a[y] \cdot |R.b| \cdot |R.c| + R.b[\text{C}] \cdot |R.c| + R.c[20]\\ &= \; 1 \cdot 3 \cdot 4 + 2 \cdot 4 + 1 = 21\end{aligned}$$

(b) Offset of aggregate value

Figure 4: Mapping directories for aggregation

Partition-based aggregation. We first hash- or range-partition the input and individually sort the pages of each partition (see also Section 3.1). The partitioning fanout can be smaller than the one used in join evaluation, as there is only one input. Next, we invoke new threads, each processing disjoint sets of partitions. For each partition, the thread merges the sorted pages; instead of saving the output to a merge buffer (as with join evaluation) it directly evaluates the aggregate values per group and outputs them, which significantly reduces the number of memory operations.

Map aggregation. If all grouping attributes have small distinct value cardinalities, we can aggregate in a single pass. The input is first split to the number of threads used. We keep a mapping directory for each grouping attribute, with directories shared across threads. We generate an array of aggregate values, one per aggregate function per thread. A thread looks up each tuple in each directory and finds the row to update in its private array of aggregate values. For example, consider grouping a table R on fields a, b and c. The mapping directories are shown in Figure 4, where we also show how we can compute the offset of the row to update in the aggregation arrays. Since the distinct value cardinality for the grouping attributes is small, the mapping directories quickly fill up and hold all input values; thus, aggregation bears minimal synchronization overhead. After processing all tuples, the individual aggregate value arrays are "merged" depending on the aggregate function (*e.g.*, for `sum`, corresponding group values are added).

The scalability of multithreaded aggregation grows with the size of the mapping directories, as lookups become more expensive and overlap to a greater extent with input tuple fetching. Directories, however, should not grow too large: as the directories and aggregation arrays grow (the size of each aggregation array being the product of distinct values of each grouping attribute), they start "spilling" outside the L1-cache, or even the L2-cache, so lookups and aggregate value updates are likely to trigger cache misses. This is aggravated by multiple threads sharing the lowest cache level, so the cache capacity available per thread is reduced.

4. PERFORMANCE MODELING

In CMPs, multiple threads can work independently provided there is no synchronization overhead and their datasets are cache-resident; this would provide linear speedups. This is not always feasible, though, as threads will contend to access memory-resident data. Consider N threads processing a single relation: they will have to share the physical

P	page size (bytes)
CL	cache line size (bytes)
K	input tuple cardinality
K'	staged tuple cardinality, $0 \leq K' \leq K$
D	distinct value cardinality
T	input tuple size (bytes)
T'	staged tuple size (bytes), $1 \leq T' \leq T$
$L1$	cost for L1-cache access (CPU cycles)
$L2$	cost for L2-cache access (CPU cycles)
M	cost for main memory access (CPU cycles)
OUT	cost for building an output tuple (CPU cycles)
N	number of threads
LK	cost per locking operation (CPU cycles), 0 for $N = 1$
TO	overhead per thread (scheduling, joining *etc*)

Table 2: Model parameters

memory. If all need to fetch data at the same time, requests will be serialized in the memory system [18], diminishing the performance gains of multithreaded execution.

Consider a memory block (*e.g.*, a hash partition). Each thread's operation on it can be divided in three stages: (a) the fetching stage, where the block is requested from main memory, (b) the processing stage, and (c) the locking stage, where the thread interacts with the storage manager to request a new block. Ideally, with N threads, one thread will be fetching and $N-1$ threads will be processing cache-resident blocks. We define the *multithreaded utility ratio* R of Equation 1 as the time gained by overlapping operations through having multiple threads operate on different parts of the input. The numerator, C_f, is the cost of fetching a block; the denominator is the sum of the costs of fetching, processing (C_p), and locking (C_l).

$$R = \frac{C_f}{C_f + C_p + C_l} \tag{1}$$

Let M be the cost of a memory access. In single-threaded execution main memory is accessed by one thread. For N threads the memory bus is shared; in the worst case an equivalent $\left(\frac{1}{N}\right)^{\text{th}}$ of the maximum memory throughput is available to each core and, hence, the cost of a memory access reaches MN. Through overlapping operations, captured by the utility ratio R, the effective memory throughput will be greater. We define M', the *effective memory access cost*, as shown in Equation 2. If R is less than $\frac{1}{N}$, block operations will overlap so threads will face negligible contention for accessing memory. Else, the cost will increase depending on the multithreaded utility ratio and will approach MN as $R \to 1$, *i.e.*, when there is no processing overlap among threads.

$$M' = \begin{cases} M & R \leq \frac{1}{N} \\ MNR & R > \frac{1}{N} \end{cases} \tag{2}$$

We use this framework to estimate the speedup of multithreaded execution and give formulas for the cost of each algorithm based on a per-memory-access model. We then extract memory utility ratios for each algorithm of Section 3 and "plug in" these ratios to the cost formulas. Our goal is not to have an accurate description of execution on a per CPU-cycle granularity (which is most likely impossible due to the complexity of modern hardware), but a coarse characterization of the differences between single- and multithreaded execution. We therefore track the accesses of each algorithm to each level of the memory hierarchy. We do not account for calculations running over registers, as their execution costs are negligible compared to memory operations. We also omit the impact of hardware prefetchers, cache associativity, and non-blocking caches: their impact depends on the design of each CPU and the runtime environment. The parameters of our model are shown in Table 2; we assume a two-level deep cache hierarchy.

4.1 Sorting

The first step of sorting is to split the input into partitions of B bytes each and sort them using quicksort; the partitions are merged to produce the final sorted output. To generate a single partition to be sorted, the core needs to fetch both the input data and the partition's cache lines. For primary tables we have to account for projections and for filtering the input on (any) selection predicates, as explained in Section 3. The size of the input that is used to fill one partition is estimated to $\frac{KT}{K'T'}B$. For each partition, $\left(1 + \frac{KT}{K'T'}\right)B$ bytes will be fetched from main memory, costing M cycles for each cache line of CL bytes. The cost of fetching a single input partition is given by Equation 3. A generated partition of $\frac{B}{T'}$ tuples is (at least) L2-cache-resident. To apply quicksort, tuples need to be L1-cache-resident. Each tuple needs to be fetched twice from the L2-cache, for reading and writing it. In our implementation, each tuple examination and exchange required roughly four L1-cache accesses, for a total of $\frac{B}{T'}\log\left(\frac{B}{T'}\right)$ operations. The total cost of sorting a partition is shown in Equation 4.

$$C_f^{\text{sort}}(B) = \left(1 + \frac{KT}{K'T'}\right)\frac{B}{CL}M \tag{3}$$

$$C_p^{\text{sort}}(B) = 2\frac{B}{CL}L2 + 4\frac{B}{T'}\log\left(\frac{B}{T'}\right)L1 \tag{4}$$

The utility ratio of the sorting step, $R^{\text{sort}}(B)$, is given by Equation 5. We use that to derive the cost of multithreaded execution. The entire relation will produce $\frac{K'T'}{B}$ partitions, so fetching the input and the partitions requires $\frac{KT+K'T'}{CL}$ memory accesses. This will be divided across N execution threads, with each thread having an effective memory access cost equal to M', as defined by Equation 2 when R is substituted for $R^{\text{sort}}(B)$. Since sorting runs inside the cache hierarchy (mainly in the L1-cache), the use of N threads will most likely result in a linear speedup, so the cost for sorting the input is reduced by a factor of N. Given all these observations, the cost of the sorting step is given by Equation 6.

$$R^{\text{sort}}(B) = \frac{C_f^{\text{sort}}(B)}{C_f^{\text{sort}}(B) + C_p^{\text{sort}}(B)} \tag{5}$$

$$C^{\text{sort}}(B) = \left(KT + K'T'\right)\frac{M'}{N \cdot CL} + \frac{C_p^{\text{sort}}(B)}{N} \tag{6}$$

The second step in sorting a relation is to merge the individually sorted partitions. We maintain a heap of processed tuples across merged partitions, as explained in Section 3. The input contains $\frac{K'T'}{B}$ partitions of $\frac{B}{CL}$ cache lines each, so the cost of fetching the sorted partitions during the merging phase is given by Equation 7. Each tuple will be fetched twice, since we need to insert its value in the heap, and then output it to the appropriate position in the merged output. However, some algorithms (*e.g.*, merge aggregation) do not require materializing the sorted output, so we include a factor S, set to 2 if we materialize the output, or 1 otherwise. The processing cost is given by Equation 8, stemming from heap processing: for each output tuple, the input tuple with the smallest value is retrieved and the heap is re-organized.

$$C_f^{\text{merge}}(B,S) = S\frac{K'T'}{B}M\frac{B}{CL} = SK'T'\frac{M}{CL} \tag{7}$$

$$C_p^{\text{merge}}(B) = 2K'\log\left(\frac{K'T'}{B}\right)L1 \tag{8}$$

As with partition sorting, the utility ratio of the merging step $R^{\text{merge}}(B,S)$ is given by Equation 9. For the total cost of the merging step we generalize the last two equations for N threads, as shown in Equation 10. We cater for multiple threads by substituting $R^{\text{merge}}(B,S)$ in Equation 2 and dividing Equation 7 by the number of threads N; we do the same for the heap processing cost of a partition. The cost of the entire algorithm is the sum of Equations 6 and 10.

$$R^{\text{merge}}(B,S) = \frac{C_f^{\text{merge}}(B,S)}{C_f^{\text{merge}}(B,S)+C_p^{\text{merge}}(B)} \tag{9}$$

$$C^{\text{merge}}(B,S) = SK'T'\frac{M'}{N\cdot CL}+\frac{C_p^{\text{merge}}(B)}{N} \tag{10}$$

4.2 Partitioning

Recall from Section 3.1 that the general partitioning algorithm is similar to sorting, with two differences: (a) quicksort is applied on a per-page granularity, and (b) there is a locking overhead when directing tuples to partitions, as multiple threads will be adding pages to them. The cost $C_f^{\text{part}}(P)$ of fetching a page for partitioning is given by Equation 11, *i.e.*, similar to Equation 3 with B substituted for P, as each partition page is individually sorted. Pages are most likely buffered in the L2-cache, so they need to be fetched to the L1-cache before being sorted, and written back to the L1-cache. The cost of processing a partition page is given by Equation 12, *i.e.*, similar to Equation 4, but assuming that the page is L1-cache-resident on its second access.

$$C_f^{\text{part}}(P) = \left(1+\frac{KT}{K'T'}\right)\frac{P}{CL}M \tag{11}$$

$$C_p^{\text{part}}(P) = \frac{P}{CL}(L2+L1)+4\frac{P}{T'}\log\left(\frac{P}{T'}\right)L1 \tag{12}$$

The utility ratio of partitioning, $R^{\text{part}}(P)$, is defined in Equation 13 where the denominator includes the locking overhead (since the new page needs to be added to the partition). The total cost of multithreaded partitioning using N threads is given by Equation 14, where we use the effective memory access cost (obtained by Equation 2 with $R = R^{\text{part}}(P)$). The formula is similar to Equation 6 with the only difference being the addition of the cost for locking each page of each partition (a total of $\frac{K'T'}{P}$ pages).

$$R^{\text{part}}(P) = \frac{C_f^{\text{part}}(P)}{C_f^{\text{part}}(P)+C_p^{\text{part}}(P)+LK} \tag{13}$$

$$\begin{aligned} C^{\text{part}}(P) &= \left(KT+K'T'\right)\frac{M'}{N\cdot CL} \\ &+ \frac{C_p^{\text{part}}(P)}{N}+\frac{K'T'}{P}LK \end{aligned} \tag{14}$$

Locking is used to synchronize the interaction with the storage manager. Assuming the partitioning fanout is F, each thread will contend with the remaining $N-1$ threads; the probability of any thread requesting access to a partition is $\frac{1}{F}$. The probability of contention then depends on the factor $\frac{N!}{F^N}$ (*i.e.*, all permutations of threads into the probability of all threads accessing the same partition); that is very small. It also depends on the ratio of the duration of the lock to the duration of page processing, which also includes data fetching and sorting ($\frac{C_l}{C_f+C_p+C_l}$). We therefore expect that threads rarely need to wait for a lock to be released.

The partition pages are individually sorted, so we need to merge them in a separate step, similarly to general sorting. The difference lies in the use of the merge buffer that replaces memory accesses with accesses to the L2-cache. The fetching and processing costs are therefore modified as shown in Equations 15 and 16. Recall that if the size of the L2-cache is $|L2|$, the partition size will be roughly $\frac{|L2|}{2N}$.

$$C_f^{\text{merge}}(P,S,M) = K'T'\frac{M}{CL}+SK'T'\frac{L2}{CL} \tag{15}$$

$$C_p^{\text{merge}}(P) = 2K'\log\left(\frac{|L2|}{2NP}\right)L1 \tag{16}$$

In Equation 15, S is 0 when the output is processed on-the-fly (*e.g.*, in aggregation), or 2 when the output is saved to the merge buffer. The modified utility ratio and the merge cost are shown in Equations 17 and 18. The total cost for partitioning is the sum of Equations 14 and 18; M' is given by Equation 2 after setting $R = R^{\text{merge}}(P,S)$.

$$R^{\text{merge}}(P,S) = \frac{C_f^{\text{merge}}(P,S,M)}{C_f^{\text{merge}}(P,S,M)+C_p^{\text{merge}}(P)} \tag{17}$$

$$C^{\text{merge}}(P,S) = \frac{C_f^{\text{merge}}(P,S,M')+C_p^{\text{merge}}(P)}{N} \tag{18}$$

4.3 Join evaluation

All join algorithms run exclusively inside the L1-cache and build on the staging primitives. When joining there is no need to synchronize threads, as they operate over disjoint inputs (see also Section 3.2). The difference between the algorithms lies in where they "read" their data from. For sort-merge join each partition is read from main memory, while for hash join the input is buffered in the L2-cache. Thus, we only need to assess the cost of fetching the input and generating the output. Assuming two inputs A and B, and N threads, the cost of processing the entire input will be given by Equation 19, where $\sigma_{\bowtie}$ is the selectivity factor of the join predicate. For sort-merge join the input tables are fetched from main memory, so the cost will be given by Equation 20. For hash join, the equivalent cost of fetching from the L2-cache is given by Equation 21. To those costs we need to add the thread scheduling overhead, equal to $N\cdot TO$ in all cases.

$$C_p^{\text{join}} = \frac{K'_A K'_B \sigma_{\bowtie}}{N}OUT \tag{19}$$

$$C_f^{\text{merge-join}} = \left(K'_A T'_A + K'_B T'_B\right)\frac{M}{N\cdot CL} \tag{20}$$

$$C_f^{\text{partition-join}} = \left(K'_A T'_A + K'_B T'_B\right)\frac{L2}{N\cdot CL} \tag{21}$$

System	Dell Precision T5400
Processor	Intel Xeon E5420
Number of cores	4
Frequency	2.5GHz
Cache line size	64B
I1-cache	32KB ×4
D1-cache	32KB ×4
L2-cache	6MB ×2
L1-cache access latency	3 cycles
L1-cache miss latency (sequential)	9 cycles
L1-cache miss latency (random)	14 cycles
L2-cache miss latency (sequential)	48 cycles
L2-cache miss latency (random)	85-250 cycles
RAM type	4x1GB Fully Buffered DIMM DDR2 667MHz

Table 3: Testbed specifications

The total cost of sort-merge join will be equal to the cost of sorting both inputs (Equations 6 and 10 with S set to 2), plus fetching the blocks of both inputs from main memory (Equation 20), plus the cost of generating the output (Equation 19), plus the cost of thread scheduling ($N \cdot TO$). Similarly, one can extract the cost of hash join evaluation: it is equal to the cost of partitioning the input (Equation 14 and Equation 18 with S set to 3 to include each input's contribution to Equation 21 as well), plus the output generation cost (Equation 19), plus the thread scheduling overhead.

4.4 Aggregation

Recall from Section 3.3 that aggregation is evaluated on-the-fly, without restructuring the input. For merge and hash aggregation this means that we do not materialize the output of the merging phase; rather, we use it directly to update the aggregate values. The aggregation cost is given by the data staging cost equations: we set S to 1 for merge aggregation and to 0 for hash aggregation. We also include the scheduling cost $N \cdot TO$ for multithreaded execution.

Map aggregation makes a single pass over the input with no intermediate staging. Memory accesses overlap with lookups on the mapping directories, as the latter are cache-resident. Assuming G grouping attributes, A aggregation functions, and binary search for mapping directory lookups, input fetching and processing are given by Equations 22 and 23 respectively; D_i is the distinct value cardinality of group i.

$$C_f^{\text{map}} = \frac{KT}{CL} M \tag{22}$$

$$C_p^{\text{map}} = \left(\sum_{i=0}^{G} \left(\log\left(D_i\right) L1 \right) + A \cdot L2 \right) K' \tag{23}$$

The first term in Equation 23 is the cost of binary search in each directory; the second term is the cost of updating the aggregation arrays. The assumption is that the mapping directories fit in the L1-cache, while the (possibly) larger aggregation arrays are evicted to the L2-cache. We can estimate the map aggregation cost as shown in Equation 25, where M' is given by using the utility ratio of Equation 24.

$$R^{\text{map}} = \frac{C_f^{\text{map}}}{C_f^{\text{map}} + C_p^{\text{map}}} \tag{24}$$

$$C^{\text{map}} = \frac{KT}{N \cdot CL} M' + \frac{C_p^{\text{map}}}{N} \tag{25}$$

5. EXPERIMENTAL STUDY

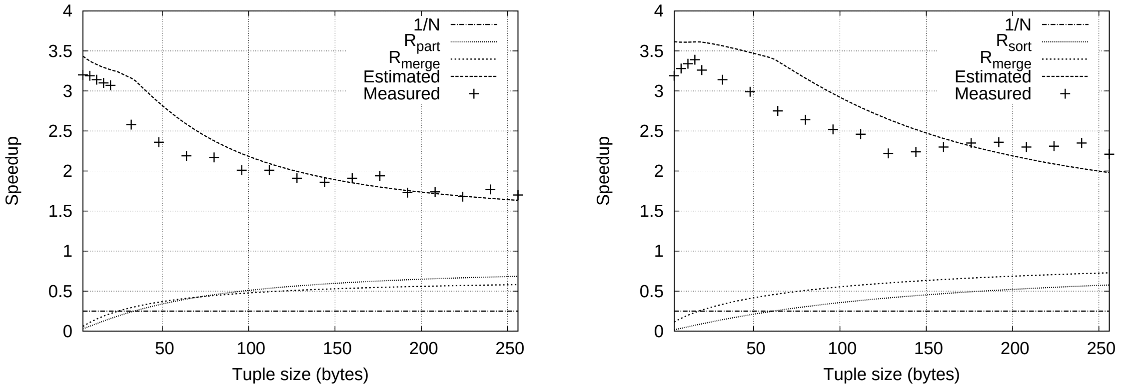
To verify the efficiency of our proposals and the correctness of the analytical model, we implemented our algorithms in C and conducted an extensive experimental study. The hardware platform we used was a Dell Precision T5400 workstation, with an Intel Xeon E5420 quad-core processor, clocked at 2.5GHz with 4GB of physical memory running GNU/Linux (64-bit version, kernel 2.6.26). The C code was compiled with the GNU `gcc` compiler (version 4.3.2) using the `-O2` compilation flag. We used the `pthread` thread library. Details about the testbed are shown in Table 3. The cache latencies were measured with the RightMark Memory Analyser [22].

We used tables of various schemata and cardinalities and stored them using NSM. Primary tables were cached in the buffer pool of a typical storage manager. All intermediate results (*e.g.*, partitions) were saved as temporary tables, also controlled by the storage manager.We hard-coded all benchmark queries to reduce instruction-level overhead using the holistic query evaluation model [14]. This was beneficial to single-threaded performance, as multithreading can exploit the instruction caching and issuing mechanisms of multiple cores. We expect iterator-based implementations of our algorithms (*e.g.*, based on the *exchange* operator of [9]) to result in higher speedups but slower response times. We ran each query ten times in isolation and report the average response times; the deviation was less than 3% in all cases. We also report the speedup when moving from single-threaded to multithreaded execution.

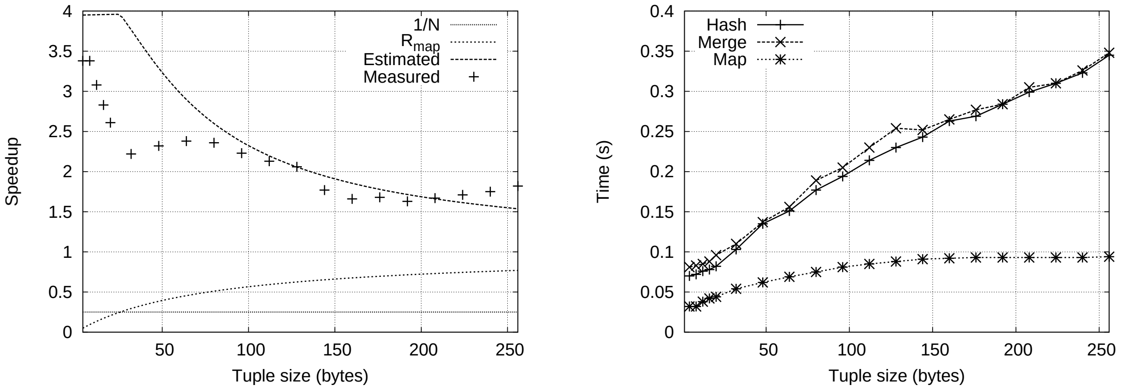
Measured speedups were compared with the ones estimated by the analytical model. To apply the model, we set N to 4, as our reference CPU supports one thread per core, $L1$ to 3, $L2$ to 14 and M to 100, as accesses are both sequential and random. We calibrated the locking cost LK to $5M$ and TO to 2.5% of total execution time. We set OUT to zero and did not generate results (unless explicitly stated), to isolate the multithreaded performance of the algorithms; result generation runs inside the L1-cache for each thread and thus inflates scalability.

5.1 Aggregation

We measured the impact of input tuple size by using a table of 1M tuples ($K = K'$) and varying the tuple size between 4 and 256 bytes ($T = T' \in [4, 256]$), using one grouping attribute with 1,000 distinct values (D). The estimated and measured costs for merge, hash, and map aggregation, as well as their comparative performance when using four threads, are shown in Figure 5. When R becomes greater than $\frac{1}{N}$ we expect the effective memory access cost M' to start increasing. This is verified experimentally, as the slope significantly grows when R exceeds this threshold. The estimate for hash aggregation is more accurate than that for merge aggregation. The fluctuation in the latter is due to cache line alignment effects, which are not included in our model. In terms of algorithm performance, the measured speedup is over 3 for small tuple sizes. It degrades for wider tuples, as the cores will spend more time fetching data from memory. This is more intensive in hash than merge aggregation, as the computational load for sorting and merging larger blocks keeps the cores busy to a higher extent. For map aggregation, the mapping directory has enough entries to make the lookup cost comparable to the cost of fetching small tuples. As the tuple size grows the fetching cost scales and dominates, resulting in poorer performance. The



(a) Hash aggregation (b) Merge aggregation



(c) Map aggregation (d) Multithreaded performance

Figure 5: Impact of tuple size on aggregation

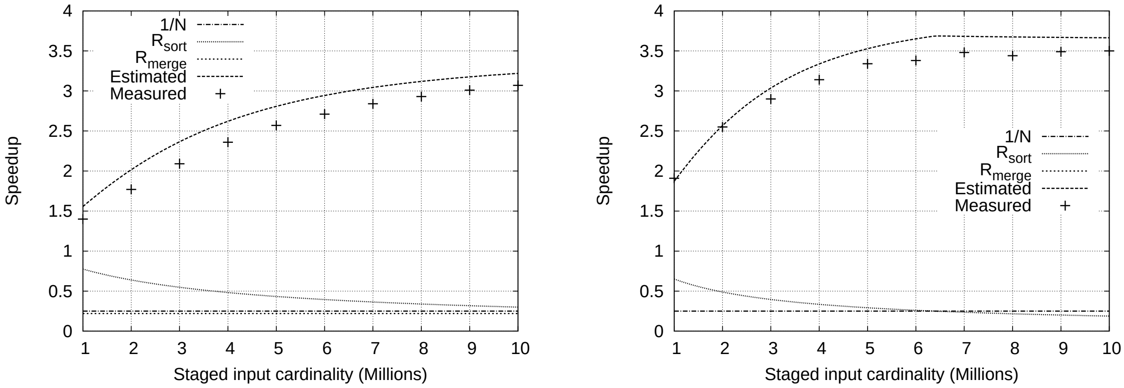
deviation in Figure 5(c) for small tuple sizes is due to overestimating the cost of updating the aggregation arrays: it varied between $L1$ and $L2$, but is set to $L2$ in Equation 25. As shown in Figure 5(d), merge and hash aggregation have comparable performance, as they incur a similar number of accesses to main memory. Map aggregation needs no input staging and is thus faster and less sensitive to changes in tuple size, for the given (small) number of values of the grouping attribute.

We then measured the impact of input cardinality after applying selections and projections. We used a table of K=10M tuples of T=72 bytes each and varied the selectivity between 0.1 and 1; each tuple after staging was 20 bytes (T'); D was set to 1,000 again. The results are shown in Figure 6. The performance is accurately modeled, with estimated and measured curves for all aggregation algorithms being close and following the same trends. For a small selectivity, the cost of fetching the primary table is higher than sorting the filtered data. As selectivity grows the speedup increases and converges to a maximum value, reached when R is less than $\frac{1}{N}$. Note that the merge-based implementation gives higher speedups, as it better exploits the computational power of multiple cores. As for comparative multithreaded performance (Figure 6(d)), hash aggregation outperforms merge aggregation by a factor proportional to selectivity. Map aggregation widely outperforms the other algorithms and is less sensitive to selectivity as it does not build intermediate partitions.

The number of distinct values of the grouping attribute(s) has a detrimental effect on the performance of map aggregation, as it affects the size of the directories and the aggregation arrays. As the grouping cardinality increases, the auxiliary data structures are evicted to lower cache levels. This penalizes performance, as there is a significant increase in cache misses and scalability, as all threads compete for accessing memory to a greater extent. This is shown in Figure 7 for an aggregation query on 10M tuples of 72 bytes each, using one grouping attribute of varying cardinality D and four `sum` functions. In the first two cases there is no result generation; in the third case we show the impact of result generation on scalability. Merge and hash aggregation are moderately affected by the cardinality of the grouping attribute, their difference being the number of iterations during quicksort runs. Map aggregation is 2.5 times faster for small cardinalities but its performance degrades fast, indicating the inflated cost for accesses to the L2-cache and the main memory. In terms of scalability (Figure 7(b)), hash and memory aggregation exhibit high speedups, growing with cardinality. Map aggregation has a low speedup for small cardinalities, as the directory lookup cost is too small to hide memory latencies. Then, speedups grow with cardinality and start dropping again, as the auxiliary data structures are evicted to the L2-cache or outside it. Output generation provides sufficient computational load to mask memory accesses (Figure 7(c)), with all algorithms exhibiting speedups over 3 for considerable result sizes.

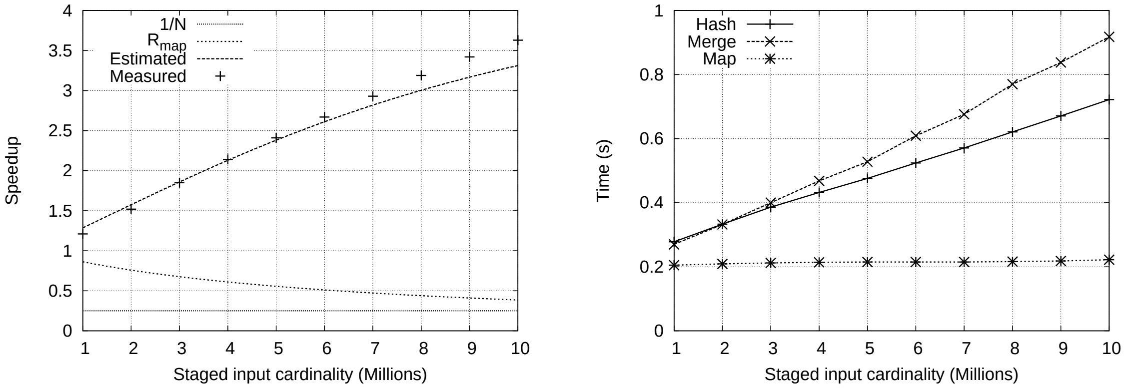
5.2 Join evaluation

We next studied multithreaded join evaluation for varying input tuple size, input cardinality, and join selectivity. We joined two tables of 1M tuples each. The outer table's tuples



(a) Hash aggregation

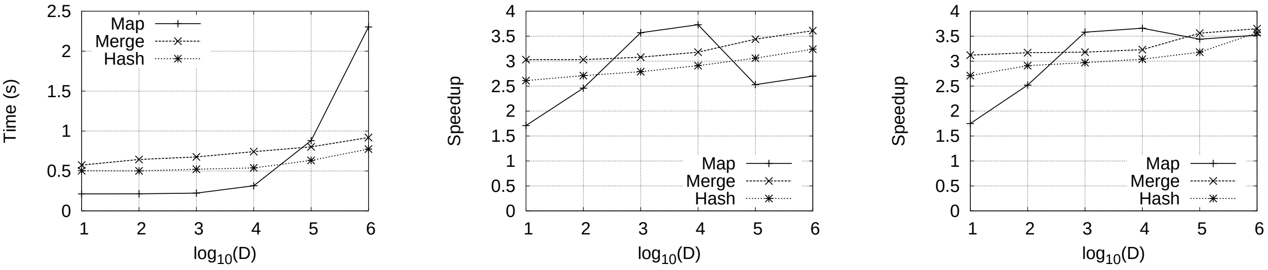
(b) Merge aggregation



(c) Map aggregation

(d) Multithreaded performance

Figure 6: Impact of selectivity on aggregation



(a) Multithreaded performance

(b) Measured speedup

(c) Speedup with output generation

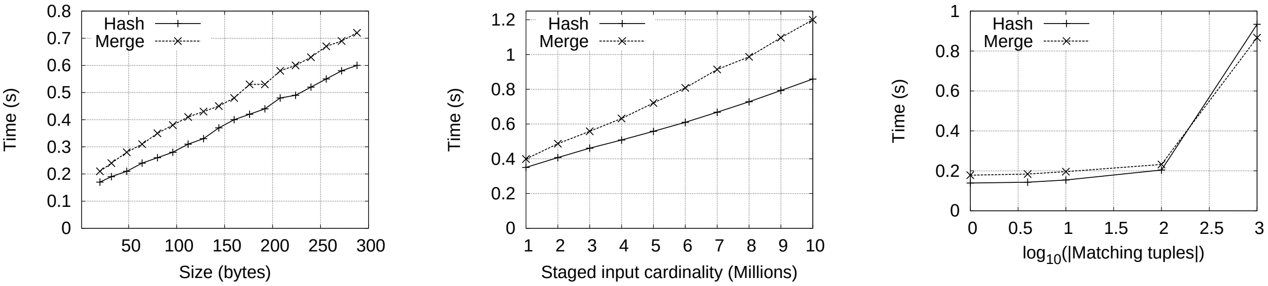
Figure 7: Impact of group cardinality

were 72 bytes long; the tuple size after staging was 20 bytes. The inner table's tuple size varied between 20 and 300 bytes. Each outer tuple matched with 10 inner tuples. The results shown in Figures 8(a) and 9(a) exhibit trends similar to the ones of aggregation (Figure 5): input staging accounted for 90% of execution time (omitting result generation) and is the same process for both aggregation and join evaluation. Hash join performs better; the use of merge buffers increases cache locality and reduces the cost of memory operations. Still, merge join results in higher speedups due to the higher computational cost of sorting larger blocks.

For cardinality experiments we used two tables with tuple sizes of 72 bytes, reduced to 20 bytes after staging; each outer tuple matched with 10 inner ones. The outer table's cardinality was 1M and the inner's was 10M, but we filtered the inner table with a predicate of selectivity ranging between 0.1 and 1. The results of Figures 8(b) and 9(b) are similar to those of Figure 6, with speedups increasing and converging to a maximum value. In terms of join predicate selectivity, we joined two tables of 1M tuples, 72 bytes each, but staged to 20 bytes. We varied the number of matching inner tuples per outer tuple to 1, 4, 10, 100, and 1,000. As join selectivity grows, the speedup is close to linear for both algorithms, as shown in Figures 8(c) and 9(c). This is due to join predicate evaluation effectively "backtracking" between multiple matches. Processing runs inside the L1-cache, reducing the frequency of memory accesses and resulting in high speedups.

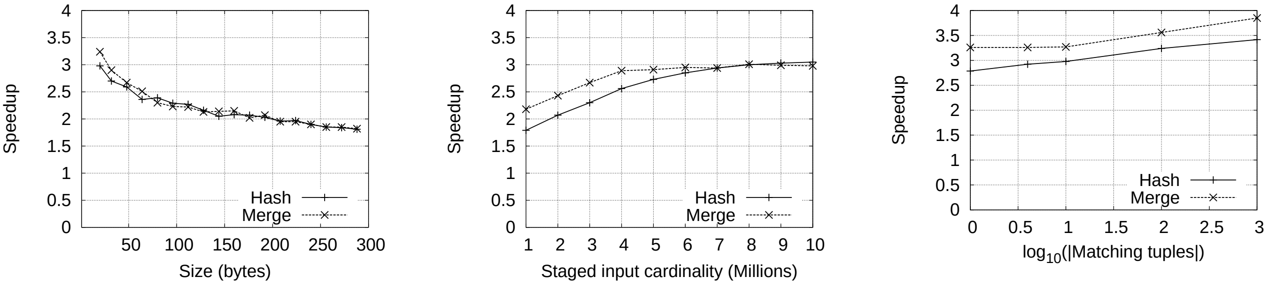
5.3 Pipelined operators

We now move on to a query combining two joins and an aggregation. We used three tables with 1M tuples of 72 bytes each. In the first join, each outer tuple matched with 4 inner ones; in the second join the number of matching



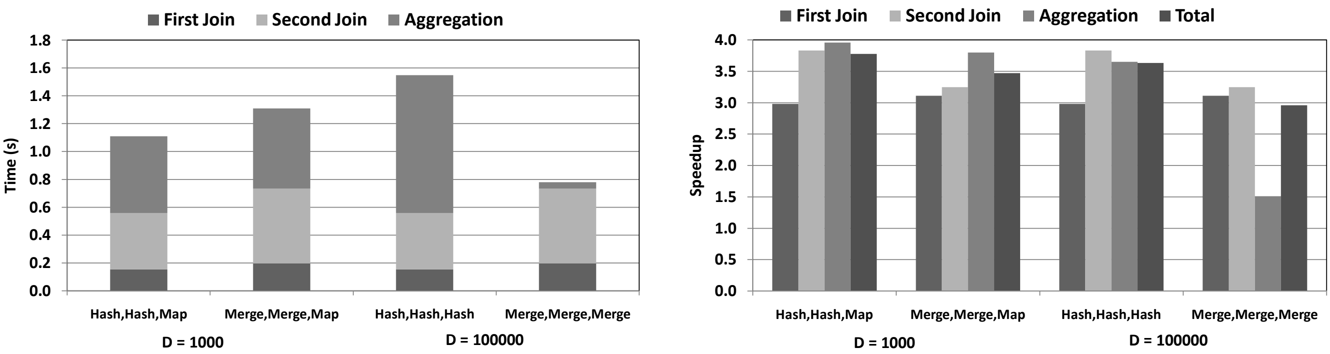
(a) Input tuple size (b) Input selectivity (c) Join predicate selectivity

Figure 8: Multithreaded performance of join evaluation



(a) Input tuple size (b) Input selectivity (c) Join predicate selectivity

Figure 9: Measured speedup for join evaluation



(a) Multithreaded performance (b) Measured speedup

Figure 10: Multiple operators

inner tuples was 10. The two joins produce 4M and 40M tuples respectively. We used both merge and hash join. The result was `sum`-aggregated over one grouping attribute with either 1,000 or 100,000 distinct values. In the first case we used map aggregation. In the second case, the grouping attribute was the same as the join attribute of the second join, to measure the impact of sorted runs. The results are shown in Figure 10; the labels indicate the algorithms used for each operator.

Hash join is faster than merge join, verifying once again that the use of an L2-cache buffer for merging pays back. For aggregation, when the number of values for the grouping attribute is 1,000, the use of map aggregation is very efficient: it needs 0.55s for 40M tuples, resulting in a throughput of 72.6M tuples/s. In terms of scalability, the reduction in tuple size allows all operators apart from the first to work on small tuples and, hence, they do not fetch data not needed for processing. The observed speedups are over 3 and, for hash join and map aggregation, close to linear.

When the number of groups increases to 100,000, hash and merge aggregation become more efficient as map aggregation exhibits excessive cache misses. We use either all-hash or all-merge algorithms. The cost of hash aggregation is twice that of map aggregation in the previous case (*i.e.*, when $D = 1{,}000$). However, since the output of the second join is already sorted on the grouping attribute, merge aggregation does not need intermediate partitions, but is evaluated in a single pass of the join result. A direct comparison of map and merge aggregation shows that the latter needs only a small portion of the time needed by the former, as there are no directory lookups and updates of aggregate arrays. However, the speedup of merge aggregation is limited as there is no computational load to effectively mask the cost of memory accesses.

6. RELATED WORK

Simultaneous multithreading (SMT), a form of TLP, was explored in [24]: a helper thread was used to aggressively prefetch data to be used by the main thread. This technique is not applicable in multicores with no in-core support for TLP, as the helper thread will fetch data to a different L1-cache than the one used by the main thread. The authors of [5] examined inter-operator communication and proposed using chunks of the output as buffers for each thread. We use a separate output buffer per thread to avoid synchronization and a similar approach for partitioning (see also [4]), since each thread has exclusive access to one partition page. As we sort pages during partitioning the processing time per page increases, and thread contention for locking is minimized.

In [3], the authors tested and modeled the use of private and/or shared hash tables for aggregation on CMPs. Their approach is tailored to processors supporting multiple (four for the employed CPU) threads inside each core; it is not clear how it can efficiently be ported to architectures with no in-core support for TLP. The combination of SIMD instructions with multithreading on multicores was studied in the context of mergesort [2] and join evaluation [13]. This approach proves highly efficient when processing vertically partitioned data, but it cannot be directly applied to query engines processing NSM-based pages. Our framework is independent of the storage layout.

In [17], the authors gave an analytical model for single-threaded main-memory query execution. The model captured the cost of stalls, *e.g.*, cache and TLB misses, according to the access pattern. Our model does not distinguish between sequential and random access patterns but accounts for accesses to the L1-cache, as CPUs do not have enough memory ports to serve successive read and write operations. Finally, [12] tackled work sharing in CMPs and modeled the performance of concurrently processed, staged queries; [21] investigated scheduling of multiple queries for scan sharing; and [15] suggested the use of page coloring to prevent cache thrashing when concurrently executing multiple queries. These are complementary to our work: we focus on intra-operator parallelism and model the contention for shared hardware resources.

7. CONCLUSIONS AND FUTURE WORK

We studied multithreaded query processing on chip multiprocessors. By identifying main memory accesses and thread synchronization as the performance bottlenecks, we provided a uniform framework for implementing query processing algorithms that: (*a*) reduces contention for hardware resources, and (*b*) bears minimal synchronization overhead. We analytically modeled the performance and scalability of each algorithm to statically estimate the benefit of multithreaded execution. We implemented and experimentally validated our proposals. The results verify the correctness of our model and the efficiency of our algorithms, which, in some cases, achieve almost linear speedups.

8. REFERENCES

[1] A. Ailamaki *et al.* DBMSs on a Modern Processor: Where Does Time Go? In *The VLDB Journal*, 1999.

[2] J. Chhugani *et al.* Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2):1313–1324, 2008.

[3] J. Cieslewicz and K. A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *VLDB*, 2007.

[4] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, pages 25–34, 2008.

[5] J. Cieslewicz, K. A. Ross, and I. Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, 2007.

[6] J. Dean and S. Ghemawat. MapReduce: Simpliied Data Processing on Large Clusters. In *OSDI*, 2004.

[7] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[8] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2), 1993.

[9] G. Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, 1994.

[10] N. Hardavellas *et al.* Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, pages 79–87, 2007.

[11] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaumann Publishers Inc., 4 edition, 2006.

[12] R. Johnson *et al.* To Share or Not To Share? In *VLDB*, 2007.

[13] C. Kim *et al.* Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[14] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[15] R. Lee *et al.* MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *PVLDB*, 2(1):373–384, 2009.

[16] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a Join? - Dissecting CPU and Memory Optimization Effects. In *VLDB*, 2000.

[17] S. Manegold *et al.* Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB*, 2002.

[18] T. Moscibroda and O. Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *SS*, 2007.

[19] C. Nyberg *et al.* AlphaSort: A Cache-Sensitive Parallel External Sort. *VLDB J.*, 4(4), 1995.

[20] OProfile. A System Profiler for Linux, 2008. `http://oprofile.sourceforge.net/`.

[21] L. Qiao *et al.* Main-memory scan sharing for multi-core CPUs. *Proc. VLDB Endow.*, 2008.

[22] RightMark. RightMark Memory Analyser, 2008. `http://cpu.rightmark.org/products/rmma.shtml`.

[23] Sandia National Laboratories. More chip cores can mean slower supercomputing, Sandia simulation shows, 2009. `http://www.sandia.gov/news/`.

[24] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving Database Performance on Simultaneous Multithreading Processors. In *VLDB*, 2005.