MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases

Rubao Lee[1,2] Xiaoning Ding[2] Feng Chen[2] Qingda Lu[2]* Xiaodong Zhang[2]

[1] Institute of Computing Technology
Chinese Academy of Sciences
Beijing, 100080, China

[2]Dept. of Computer Science & Engineering
The Ohio State University
Columbus, OH 43210, USA

{liru, dingxn, fchen, luq, zhang}@cse.ohio-state.edu

ABSTRACT

In a typical commercial multi-core processor, the last level cache (LLC) is shared by two or more cores. Existing studies have shown that the shared LLC is beneficial to concurrent query processes with commonly shared data sets. However, the shared LLC can also be a performance bottleneck to concurrent queries, each of which has private data structures, such as a hash table for the widely used hash join operator, causing serious cache conflicts. We show that cache conflicts on multi-core processors can significantly degrade overall database performance. In this paper, we propose a hybrid system method called *MCC-DB* for accelerating executions of warehouse-style queries, which relies on the DBMS knowledge of data access patterns to minimize LLC conflicts in multi-core systems through an enhanced OS facility of cache partitioning. MCC-DB consists of three components: (1) a *cache-aware query optimizer* carefully selects query plans in order to balance the numbers of cache-sensitive and cache-insensitive plans; (2) a *query execution scheduler* makes decisions to co-run queries with an objective of minimizing LLC conflicts; and (3) an *enhanced OS kernel facility* partitions the shared LLC according to each query's cache capacity need and locality strength. We have implemented MCC-DB by patching the three components in PostgreSQL and Linux kernel. Our intensive measurements on an Intel multi-core system with warehouse-style queries show that MCC-DB can reduce query execution times by up to 33%.

1. INTRODUCTION

Driven by the Moore's Law, computer architecture has entered a new era of multi-core structures [1]. Multi-core processors are being widely utilized by many applications including DBMSs, and are becoming standard computing platforms. On such a processor, there are two or more computing cores that share an on-die last level cache (LLC), such as the Intel Core i7 and the Sun UltraSPARC T2 processors. In the meanwhile, the low-price of DRAM and new 64-bit memory address space make it possible to equip computers with very large main memory. These hardware advancements bring new challenges to the design and implementation of DBMSs since the performance bottleneck has been shifted from the slow I/O access speed to high memory access latency [2] [3]. This performance bottleneck shifting in computer systems for data-intensive applications, such as a DBMS, makes the LLC a critical component to improve both memory and overall performance. In this paper, we investigate several important issues on effective utilization of LLCs in multi-core processors for warehouse-style queries that are represented by TPC-H[1] and the Star Schema Benchmark (SSB) [23].

1.1 Cache Contention and Its Challenges

The shared LLC in multi-core processors is a double-edged sword for DBMS transactions. A favorable consequence is described as follows. When concurrent query execution processes access common tuples and index data in a database, the LLC enables multiple cores for constructive data sharing to reduce unnecessary memory accesses. In the LLC, a query execution process can reuse the cache lines loaded by its co-runners [24]. However, an unfavorable consequence of LLC creates a barrier for DBMS transactions. For example, private data structures during a query execution process, such as a hash table for the hash join operator, cannot be shared by multiple processes. Thus, multiple data structures from different queries can easily cause cache conflicts in the shared LLC whose capacity is often only several Megabytes. Since these data structures are frequently accessed during query execution, they can suffer from a large volume of off-chip DRAM accesses with long latencies.

A major objective of our work is to make query execution processes that have different data access patterns and thus different locality strengths (which is defined, in general, as the reuse distance of a data object reflecting its access frequency) to adopt to multi-core architecture with a shared LLC. For example, a hash table during hash join execution has strong locality, where the data would be reused later. In contrast, the tuples during a sequential table scan have weak locality since no data would be reused. Due to the locality strength difference, the performance impact from allocated cache space varies across different query execution

*Now at Intel Corp

[1]http://www.tpc.org/tpch/.

VLDB '09, August 24-28, 2009, Lyon, France

processes. Therefore, the cache allocation policy for a query process must be dependent on its locality strength.

To meet our objective, we need to address several critical issues of understanding and identifying access patterns and their locality strengths in query operators, which can be supported by the DBMS domain knowledge. However, a technical challenge is to enable a DBMS at the application layer to effectively utilize the shared LLC on multi-core processors. This is because the cache is managed by the on-chip hardware controller that is several layers lower from application software. Processors normally use an approximation mechanism of Least-Recently Used (LRU) algorithm to make cache replacement decisions. However, a LRU-like algorithm is a demand-based scheme that cannot prevent the cache space from being polluted by weak-locality data, such as sequentially scanned tuples or one-time accessed data in databases [15][25]. Thus, under existing cache management, data sets with strong locality of concurrent queries can be replaced by highly demanded but weak-locality data sets. In other words, the underlying hardware cache management in multi-core processors is unable to identify weak-locality data accesses among concurrent query execution processes.

Although the hardware is not able to understand data access patterns during a query execution, a DBMS in the application software domain has a clear picture about the data access patterns for a given query execution plan [22]. The physical operators have fixed and predictable data access patterns, which have already been used in DBMS's special buffer cache management to address I/O issues [6][31]. However, the DBMS running at the user space cannot directly manage the hardware cache allocation. The critical issue is how to make effective cache allocations for different queries guided by data access patterns. In this paper, we propose a hybrid system framework called MCC-DB (Minimizing Cache Conflicts for DataBase), which relies on the DBMS knowledge of data access patterns to make multi-core-aware plan selection and query scheduling, and to minimize the cache conflicts in multi-core systems by an enhanced operating system facility of cache partitioning.

1.2 A Motivating Example

We illustrate the cache conflict problem by showing performance loss caused by conflict misses between co-running queries in a multi-core processor. We used two different queries (a hash join and an index join [2]) as workloads, and executed them independently or together in PostgreSQL on a dual-core CPU with a shared L2 cache. Figure 1 shows the L2 miss rates for the two queries and their combinations. When the hash join was running alone, its L2 miss rate was only 6.03%. But when it was co-running with another hash join instance, the miss rate increased to 21.3%; When its co-runner became index join, its miss rate further increased to 27.7%, and it suffered a 56% performance degradation in wall clock time. However, index join was not affected too much regardless of its co-runner. This experiment shows that a query with strong locality can easily be affected by another query with weak locality. Such cache conflicts for running database workloads have also been observed in the architecture community [19]. We summarize three performance issues based on this motivating example:

1. Different performance behaviors are observed by co-running different query types mainly due to different locality strengths. This suggests that the query optimizer should be shared LLC-aware when it makes selection from candidate query plans.

2. A query execution scheduler should make shared-LLC-aware decisions to co-run different types of queries on a multi-core processor. It should avoid co-running those queries that cause significant LLC conflicts.

3. To effectively execute multiple queries on a multi-core processor, partitioning the shared LLC for co-running processes according to their locality strengths is needed in order to best utilize the limited LLC space.

[2]In this paper, we use the term "index join" to represent "index-based nested loop join".

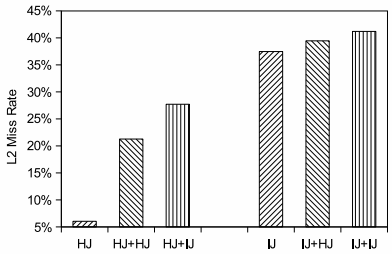


Figure 1: L2 miss rates when running a single or co-running various combinations of a hash join (HJ) and an index join (IJ).

We have been motivated by the three issues to design and implement a multi-core aware DBMS system (MCC-DB) in order to efficiently run a commonly used DBMS on multi-core processors with minimal modification to the existing DBMS implementation.

1.3 MCC-DB: A Framework with Collaborative Efforts from Both DBMS and OS

MCC-DB applies DBMS's knowledge of query execution patterns at the application level to guide the system and architecture to make actions by effectively utilizing the shared LLC at runtime. It utilizes an enhanced operating system capability to partition cache among concurrent processes.

Although an OS cannot directly make on-chip cache allocation, it can control how to allocate pages in the main memory through the virtual-physical address mapping. Since the LLC is physical address indexed, the OS can utilize the page coloring technique [30] to make cache partitioning among processes [18][28]. The foundation of MCC-DB is supported by both DBMS and OS and is outlined as follows.

1. **DBMS Control.** Since the DBMS is able to predict data access patterns during query execution, we rely on its knowledge to distinguish query locality strengths. Two key components are the query optimizer and the execution scheduler. The former selects query plans to generate a balanced number of strong-locality and weak-locality query plans, while the latter reduces potential LLC conflicts via co-running query plans with different locality strengths.

2. **OS Support.** Once concurrent query execution processes are scheduled to run on a multi-core processor, MCC-DB will use the enhanced OS mechanism to enforce necessary cache allocation to each process so that the utilization of the shared LLC is optimized.

1.4 Our Contributions

The contributions of our work are three-fold. First, we have identified the cache conflict problem of running a DBMS in multi-core processors. We have also shown that technical challenges to address this problem are beyond the ability scope of the DBMS itself. Second, we have made a strong case for a collaboration between the DBMS and the OS to achieve the goal of minimizing cache conflicts. We have designed and implemented MCC-DB that effectively breaks the performance bottleneck in the shared LLC. Finally, we have evaluated MCC-DB on a modified PostgreSQL and a modified Linux kernel, and have shown that MCC-DB can reduce query execution times by up to 33% for warehouse-style queries. To our best knowledge, MCC-DB is the first multi-core cache optimized DBMS system with a well documented design and performance evaluation. We believe that this hybrid system framework can be easily adopted to both commercial and open source databases in practice.

The rest part of this paper is organized as follows. Section 2 discusses the cache conflict problem. Section 3 introduces our MCC-DB framework. Section 4 presents how to determine query locality. Section 5 describes MCC-DB without OS support, while section 6 describes MCC-DB with cache partitioning support of the OS. Performance evaluation is in section 7. Section 8 presents related work. We conclude this paper in the last section.

2. CACHE CONFLICTS ON MULTI-CORES

Increasing the number of processing cores can improve the inter-query parallelism for DBMS transactions. However, the limited cache space would be shared by more concurrent query executions, which can lead to unnecessary cache conflicts and cause undesired performance degradations. In essence, the cache conflict occurs due to three reasons.

1. Different query executions can have very different data locality strengths, which determine how much a query can benefit from the allocated cache space.

2. The simple LRU-based cache replacement policy used in LLC does not consider how a query can really benefit from the cache but only considers how to satisfy a query's cache capacity demand.

3. A query execution process has its private data structures that need to be frequently accessed. However, such data structures can be replaced by one-time accessed data structures (weak locality), or by other similar data structures due to limited cache capacity.

In order to well understand the problem, we first show the diverse locality strengths of DBMS queries, then we discuss the drawback of LRU-based cache replacement in the LLC.

2.1 Diverse Cache Localities of Warehouse-style Database Queries

In this section, we use TPC-H queries (1GB data set) as examples of warehouse-style queries to demonstrate the existence of different locality strengths across various query executions. Our experimental system is a DELL PowerEdge 1900 server, which has two Intel Core2Quad Xeon X5355 2.66GHz CPUs, 16GB FB-DIMM memory, and five 146GB 15,000 RPM SCSI disks. Each Xeon processor has four cores, and every two cores share a 4MB L2 cache (the LLC). We use RedHat Enterprise Linux Server 5 with the Linux kernel 2.6.20 and EXT3 file system. The DBMS used in our experiments is the PostgreSQL 8.3.0.

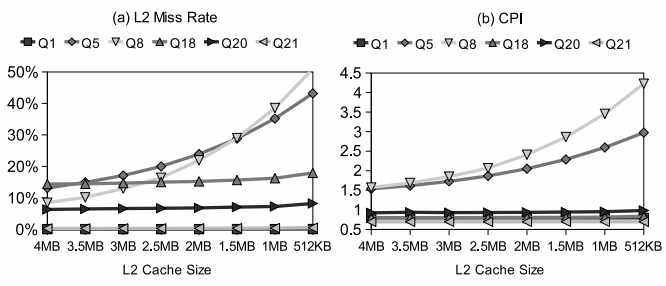


Figure 2: The performance of TPC-H queries when shrinking the L2 cache size.

In order to examine how the cache size affects query execution performance, we use MCC-DB's cache partitioning mechanism (more details in Section 6) to alter the available L2 cache space allocation for each query execution and examine the changes of its performance correspondingly. In our experiments, the allocated L2 cache size is varied from 4MB to 512KB in the descending order. We measured the performance by two metrics, the *L2 cache miss rate* and the *Cycles Per Instruction (CPI)*, [3] as shown in Figure 2. The figure does not show the queries with too short execution times and Query 9, which has a CPI of 9.66 to 11.83 and a L2 miss rate of 38.8% to 49.3%.

As shown in Figures 2 (a) and (b), we can find that there is a strong correlation between the CPI (execution time) of a query execution and the corresponding L2 cache miss rate. This indicates that the L2 cache plays a key role in determining the query execution performance. We can also see that different query executions show diverse behaviors when we change the available cache size. We can generally classify the queries into two groups:

(1) **Cache-sensitive queries** (Q5, Q8, and Q9) – their execution times (CPI) are significantly affected by the size of the allocated L2 cache space. The three queries are all dominated by multi-way hash joins.

(2) **Cache-insensitive queries** (Q1, Q18, Q20, and Q21) – their execution times do not change when we reduce the cache space. Among them, Q1 is dominated by a sequential table scan, Q18 is dominated by hash joins, and Q20 and Q21 are dominated by nested sub-query executions.

In essence, cache sensitivity of a query is determined by its locality strength. Depending on the data access patterns of operators for evaluating these queries, the queries have the following three types of locality strengths:

(1) **Strong locality** – a query has a frequently-reused data structure whose size is very small compared to the cache size. Common query types are hash aggregation on a sequential table scan (e.g. Q1) and hash join with small hash tables (e.g. Q18). A strong-locality query is cache insensitive as long as the cache space allocated to it can hold its frequently-reused data structure. It has the least performance impact on its co-runners, but it can be affected by the co-runners.

(2) **Moderate locality** – a query has a frequently-reused

[3]We use the perfmon tool to examine hardware counters (available at: http://perfmon2.sourceforge.net/).

data structure whose size is yet comparable to the cache size. Thus, a small change in cache size would make a perceptible variation in cache misses. This category is mainly hash join with moderate hash tables (e.g. Q5, Q8, and Q9). A moderate-locality query is cache sensitive and very vulnerable. It can be easily affected by its co-runners, but it has only a moderate performance impact on its co-runners.

(3) **Weak locality** – a query is characterized as follows: (1) data accesses during its execution are not reused frequently (e.g., index join or index-scan-based sub-query execution), or (2) despite the existence of a frequently-reused data structure, its size is much larger than cache size (e.g. hash join with large hash tables). A weak-locality query is cache insensitive. It has the most strong performance impact on its co-runners, but it is nearly never affected by the co-runners.

2.2 Limitations of the LRU-based Cache Replacement in the LLC

Despite the existence of diverse query locality strengths, the LLC does not make locality-driven cache allocation, due to the limitations of the LRU-based cache management.

In a typical commercial CPU, the LLC is a set-associative and physical address indexed cache. It contains multiple sets, and there are multiple ways in each set. For a data object corresponding to a cache line, its physical address in the memory statically determines which set the object will be loaded into. The way replacement algorithm in the set determines which way in the set the object will stay at.

The cache management is done by a LRU-like algorithm, which only uses the recency information of blocks when selecting a victim to be replaced. In particular, it always selects the *least recently used* data for eviction. Though simple, it cannot prevent the cache space from being *polluted* by the highly-demanded blocks with one-time access patterns. As shown in Figure 1, a large number of rarely reused index-scan data can easily grab cache space from its co-runner. In the OS buffer cache or the DBMS buffer pool, more complex replacement algorithms, such as Clock-Pro [14] (in Linux and NetBSD), LIRS [15] (in MySQL) or 2Q [16] (in PostgreSQL), can be used to extend the LRU with the reuse distance information to address the weak locality issue. However, such sophisticated algorithms are too complex to be used in hardware to manage the LLC on multi-core processors. Thus, the current LRU-based replacement in LLC is incapable of addressing the cache conflict problem.

3. MCC-DB: AN OVERALL FRAMEWORK FOR MINIMIZING CACHE CONFLICTS

On multi-core platforms, database queries with different locality strengths contend for shared cache spaces. Conventional DBMSs are designed and optimized for single core platforms, and thus cannot efficiently handle concurrent queries to minimize cache conflicts on multi-core platforms. In this section, we first analyze this inability of conventional DBMSs, and then present MCC-DB's overall structure, focusing on its enhancement and extensions over conventional DBMSs.

3.1 Core Components in MCC-DB

To understand the inability of conventional DBMSs in reducing cache conflicts, we summarize the fundamental differences of query execution between single-core and multi-core platforms, and analyze the new requirements of query processing on multi-core platforms. First, on single-core platforms, queries are processed in a batch mode. Thus, a query plan is made based on performance optimization of itself subject to the available computing resource to the query. A query is scheduled to execute in a standard database environment based on its priority by context switching, and the cache space is switched among different queries one at a time. In contrast, multiple queries are processed concurrently on multi-core platforms. Query plan optimization is no longer based on just a single task but attempts to achieve high performance collaboratively with other queries. An example to demonstrate such a case is presented in paper [26] that examines whether a query plan can benefit from the data loaded in the buffer pool by previous queries. Second, after query optimization, queries will be co-scheduled and co-run on multi-core platforms, and grouping them differently would have very different performance implications. Finally, the LLC is shared by multiple cores, inevitably causing cache conflicts among multiple queries. Such conflicts may significantly degrade system performance if the LLC is shared and competed for in an unmanaged manner.

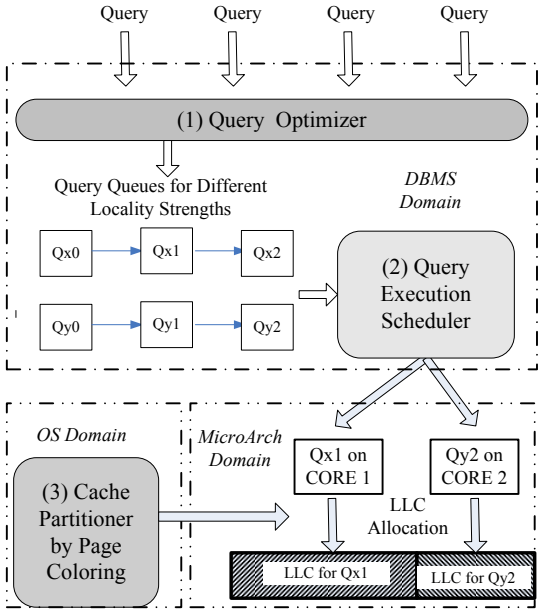


Figure 3: Enhancement and extensions in the MCC-DB framework over conventional DBMSs.

To respond to the challenges and new requirements of query processing on multi-core platforms, our MCC-DB framework enhances the existing query optimizer in DBMSs, and extends conventional DBMS architectures by introducing two new components: (1) a query execution scheduler for co-scheduling queries and (2) a cache partitioner using joint efforts from both DBMS and OS. Figure 3 shows an overview of enhancement and extensions in MCC-DB, each of which is responsible for reducing cache conflicts at a stage during the query processing lifetime.

(1) **Query Optimizer** determines which plan is to be used to execute a query. Our enhancement works when there are multiple candidate plans that are estimated to have similar execution costs. In such a case, the query optimizer selects a candidate plan that can balance cache utilization with other queries when they are co-running on multi-cores.

(2) **Query Execution Scheduler** is responsible for co-

scheduling a group of queries. In this paper, we assume a static workload with multiple queries to be executed. The scheduler selects queries with minimum cache conflict when they co-run, focusing on reducing query execution times. Currently, the scheduler does not consider query waiting times. However, it is not difficult to extend the scheduler to take waiting times into account to prevent starvation. This extension is beyond the scope of this paper, and we will consider it in our future work.

(3) **Cache Partitioner**, which is different from the above two components both working in the DBMS domain, requires support from the OS domain. It controls cache sharing by partitioning and allocating shared LLC spaces according to the cache demand of each co-running query.

3.2 Technical Issues and Challenges

In order to well utilize the powerful resources of multi-core processors with the MCC-DB framework, we must address the following technical issues and challenges.

First, to understand the indication of different queries and query plans on cache contention, we need to characterize and identify the locality strength of each query or query plan. Locality strength is important information for the query optimizer to select a query plan, for the execution scheduler to group queries, and for the cache partitioner to determine partition sizes. Query locality detection can be conducted at runtime by OS or hardware [9, 25, 28]. However, it could incur high overhead and inaccuracy. In MCC-DB, we leverage the DBMS's knowledge on query execution patterns to quantify locality strengths and to control query execution. We address this challenge in Section 4.

Second, each of the three components has its own key technique issues and challenges. For example, how does the query optimizer make trade-offs between reducing execution costs and reducing cache conflicts? How can the scheduler decide which queries can be co-scheduled with minimum performance impact from cache conflicts? How can the cache partitioner make cache allocation decisions for co-running queries according to their locality strengths?

Finally, the three components are not stand-alone. Collaborative actions must be made effectively among these components in three layers of the underlying computing environment: DBMS, OS, and multi-core architecture. Our research shows that plan selection policies (in the query optimizer) and scheduling policies (in the execution scheduler) can be greatly affected by cache allocation policies (via the cache partitioner). In Section 5, we describe the query optimizer and the scheduler without consideration of cache partitioning. This represents the best effort made only by the DBMS itself. Then, in Section 6, we present the policies in all the three components that work closely to minimize cache conflicts.

4. QUERY LOCALITY ESTIMATION

In this section, we discuss how query locality is estimated in the query optimizer.

4.1 Execution Patterns of Star Schema Queries

We target star schema based queries as represented by SSB, which is considered to be more representative than TPC-H in simulating real warehouse workloads [29]. The query structures of star schema queries are similar, i.e., aggregations on equal joins among the fact table and one or more dimension tables. Their execution plans may have two parts: (1) the join part that is a join tree including join operators and leaf-level scan nodes and (2) the aggregation and sorting part over the joined tuples. The execution patterns of such plans are as follows:

(1) The join part dominates the query execution. For example, when executing SSB queries with GROUP BY (Q2.1 to Q4.3) in PostgreSQL over a 8GB SSB database, 87% to 97% of the execution times are spent on executing multi-way joins. Thus, join operators and leaf-level scan nodes are the major contributors to the query locality strength.

(2) The aggregation and sorting part, if it exists, usually has a very small working set due to the small cardinality of the aggregation result. For example, with the hash aggregation operator, only about 100KB hash tables are built in all SSB queries with GROUP BY for a 8GB database. Thus, the aggregation and sorting part has only very limited impact on cache contention and on overall performance.

Based on the patterns, our locality estimation method does not consider the limited impact of aggregation and sorting, but only considers the locality strength of the join tree that is determined by two factors: (1) the locality strength of a single join operator (discussed in subsection 4.2), and (2) the shape of the tree that determines how multiple operators are combined (discussed in subsection 4.3).

4.2 Operator Locality Estimation

We select two operators, hash join and index join, for our study. There are two reasons. (1) They are the most important and representative join implementations in major DBMSs. Sort-merge join is not selected because star-schema-based queries cannot make cases where sorting-based algorithms outperform hash-based ones [10]. By examining query plans via an explicit *explain* command, we confirmed that PostgreSQL never selects sort-merge join when it processes any SSB queries. We also found that PostgreSQL never selects the sorting-based GroupAggregate operator. (2) Their data access patterns, with different data sizes, can represent all the three types of locality strengths in Section 2.

We analyze data access patterns of the two operators as follows. For hash join, most of its execution time is spent on probing the hash table by each tuple in the probing relation. The probing tuples have one-time access patterns. However, the hash table, which includes the hash bucket array and corresponding temporal tuples [4], is frequently reused. Thus we expect that the locality strength of hash join is closely related with its hash table size. For index join, each outer relation tuple starts an index lookup on the inner relation. Outer relation tuples have one-time access patterns. Although the locality strength of index join can also be affected by the size of index pages and tuples of the inner relation accessed during index lookups, index join has weak locality most probably in practice. The most possible case that the query optimizer selects index join instead of hash join is that the number of tuples in the outer relation is very small so that unnecessary accesses to the tuples in the inner relation can be avoided. Considering a typical join on a dimension table and the fact table, due to the huge size of the fact table and the primary key-foreign key relationship between the two tables, consecutive index lookups on the fact table have difficulty in finding data reuse opportunities in the LLC with the capacity of several megabytes.

[4]In PostgreSQL, a hash node for hash join holds a private copy of each tuple in its child node.

4.2.1 Experiments

In order to understand locality strengths and related cache conflicts of hash join and index join, we use SSB-based synthetic queries to characterize them. These queries involve only a 2-way join and have no GROUP BY. They have the following forms:

select sum(LO_REVENUE) from PART, LINEORDER
where P_PARTKEY = LO_PARTKEY
and ((P_CATEGORY = ?) or (P_CATEGORY = ?) or ...)

We select these queries because a very common pattern we can see in SSB queries is a join between the PART table and the LINEORDER table and then a sum function on the LO_REVENUE column. The selection condition on the PART table is the logical disjunction of multiple expressions on the P_CATEGORY column. Each expression is a comparison between the column and a constant value, for example $P\_CATEGORY =' MGFR\#11'$. By statistics, this column has 25 unique values, and each value is corresponding to a similar number of tuples. Therefore, each expression has an approximate selectivity of 4%. According to the number of expressions, we name these queries as PLQ_1, ..., and PLQ_{25}.

When using hash join to execute each query, the hash table is built on the tuples of the PART table. In this experiment, we use a 2GB and a 4GB SSB data set. For the scale of 2GB, the hash table size for a single expression is about 392KB. By increasing the number of expressions in the disjunction clause from 1 to 25, the hash table size can increase from 392KB to 9.28MB. For the scale of 4GB, the size can further increase to 18.6MB. When using index join to execute each query, the PART table is the outer relation that drives index scans on the LINEORDER table. In this experiment, we examine two index scan methods: the traditional B$^+$-tree index scan and the bitmap index scan.

We examine three combinations for the queries: (1) co-running two hash joins (hash/hash), (2) co-running two index joins (index/index), and (3) co-running a hash join and an index join (hash/index). For hash/hash and index/index, we run two instances of the same query. For hash/index, we first select query PLQ_{25}, which has the longest execution times for both hash join and index join, as a common co-runner. Then for each target query under examination, we run it together with query PLQ_{25}. In this way, we can ensure that the target query would not finish earlier than query PLQ_{25}, and the target query is constantly under the pressure of query PLQ_{25} during the execution.

We use the execution time of running a target query alone as the baseline case. Then we run two queries using the aforesaid three combinations to measure the performance degradations, relative to the baseline cases. Figure 4 shows the results (we report representative queries considering the graph size). In this figure, the X-axis values are the hash table sizes of hash joins for the queries, in the ascending order. For brevity, we merge experimental results for two data-set configurations in the same figure. In particular, the queries with hash tables no larger than 8.9MB are from the 2GB data-set configuration, and the rest queries are from the 4GB data-set configuration. The Y-axis is the performance degradation relative to the baseline cases. We made observations as follows. (1) An index join, using index scan or bitmap index scan, only has small and stable performance degradations, no matter whether it co-runs with a hash join or an index join. (2) An index join can affect its hash join co-runner with a hash table smaller than 12.3MB more significantly than a hash join. (3) When the hash table size is no larger than 1.1MB, two hash joins have slight interference with each other. (4) When the hash table size is between 1.1MB and 12.3MB, the performance degradations of hash joins caused by a co-running hash join are high (>10%). Even higher performance degradations (over 50%) can be found when the co-runner is an index join. (5) When the hash table sizes are larger than 12.3MB, the performance degradations of hash joins are similar to that of index joins.

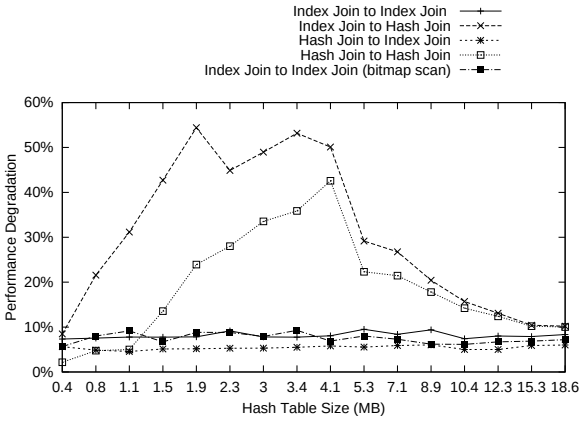


Figure 4: Performance degradations when co-running hash join and index join.

4.2.2 Identifying Operator Locality Strengths

Our experiments provide us with a basis to distinguish locality strengths of the two operators. First, according to our analysis, index joins have weak localities. Our results confirm the observations in paper [33] which shows that index joins with B$^+$-trees or even cache-conscious CSB$^+$-trees [27] suffer from significant cache thrashing and miss penalty. Second, the locality strengths of hash joins are dependent on their hash table sizes (S) and cache sizes (C). Motivated by the test results, we adopt the following rules to quantitatively identify the locality strength of a hash join, and classify hash joins into three categories:

1. If $S < \frac{C}{3}$ (1.33MB), the hash join has strong locality.
2. If $\frac{C}{3} \leqslant S < 3C$, the hash join has moderate locality.
3. If $S \geqslant 3C$ (12MB), the hash join has weak locality.

Although intuitively two co-running hash joins both with a hash table smaller than $\frac{C}{2}$ should not cause cache contention, our experiment shows that their performance degradations are more than 20%. This is because, in practice, other components in the database may consume a small amount of cache as well. Therefore, we add a small slack and use $\frac{C}{3}$ and $3C$ as boundaries to identify the locality strength of a hash join. Our experiments show that this setting performs pleasantly well in practice.

Table 1 summarizes performance degradations due to cache conflicts. There are mostly two kinds of cache conflict degrading performance: (1) *capacity contention*: two moderate-locality hash joins suffer cache conflict misses due to limited cache space. (2) *cache pollution*: an index join or a weak-locality hash join pollutes the LLC so that a strong-

Locality Strength	Strong	Moderate	Weak
Strong	low	moderate	high
Moderate	moderate	high	high
Weak	low	low	low

Table 1: Performance degradations of queries with locality strengths shown in rows caused by co-runners with locality strengths shown in columns.

or moderate-locality hash join suffers significant performance degradation. In general, capacity contention is unavoidable because it is caused by the limited cache space. However, cache pollution is eliminable via cache partitioning. More importantly, cache pollution is a much more serious problem than capacity contention due to faster cache line loading when accessing data with weak locality. Once cache pollution occurs, even hash join with strong locality can suffer high performance degradation (e.g. PLQ_2 with 0.8MB hash table). In contrast, the same hash join is free from capacity contention. This implies that, without cache partitioning to remove cache pollution, the DBMS can only attempt to use scheduling power to address the issue, and the effectiveness is very limited.

4.3 Locality Estimation of Join Trees

Having presented a quantitative method to identify the locality strength of hash join and index join at the operator level, we are now in a position to estimate the locality strength for a given multi-level join tree. According to the star schema, a base table join can only occur between a dimension table and the fact table. Therefore, for a multi-way join, a left-deep join tree is the most possible plan. In such a tree, the first join at the lowest level must be between a dimension table and the fact table, and its output tuples drive next joins with other dimension tables in a pipelined way. A key feature of a hash join node in the tree is that its hash table can only be built on a dimension table, but not on the sub-level join.

For a multi-level join tree, theoretically, each join at a level may have an impact on the overall locality strength of the join tree. However, since each join behaves as a filter to discard those tuples that cannot pass the join predict, the later joins would not be executed as frequently as the preceding joins during the pipelined join executions. Without knowing the actual data, the query optimizer finds it difficult to accurately estimate the execution frequency of each join. Therefore, we only consider the first two joins in the join tree. The accumulated hash table size of the two joins, if they exist, determines the locality strength of the whole join tree, according to the rules presented in section 4.2.2. If both the joins are index joins, the whole join tree is estimated to have weak locality.

In MCC-DB, we modified PostgreSQL's query optimizer to enable it to estimate the hash table size for a hash join. Because a hash table can only be built on a dimension table, the query optimizer can have an accurate estimation of the hash table size, which depends on the number of tuples for building the hash table and the widths of all projected columns. The number of tuples depends on the cardinality of the dimension table and the selectivity of the predict on the table. Because a dimension table is not updated frequently, the query optimizer can accurately estimate these values with statistics information.

5. MCC-DB WITHOUT PARTITIONING

In this section, we assume that there is no support from the OS domain for cache partitioning. Then we describe how the query optimizer makes plan selection (in subsection 5.1) and how the execution scheduler co-schedules queries (in subsection 5.2). As there is no OS support, this represents the best efforts made only by the DBMS itself. This is for comparisons with MCC-DB with cache partitioner (in Section 6). With the comparison, we want to show that a solution within only the DBMS domain is not enough and that OS support for cache partitioning is critical to minimizing cache conflicts. This is also supported by our experiments.

5.1 Plan Selection Policy in Optimizer

A conventional query optimizer estimates execution costs of candidate plans using a cost model and selects the optimal one with the smallest cost. However, this principle does not fit a multi-core environment because cache conflict is not considered. An optimal plan may suffer high cache conflict during execution so that it may be worse than a sub-optimal candidate that does not raise high cache conflict. Therefore, a multi-core-aware query optimizer must make the trade-off between reducing execution cost and reducing cache conflict. We extend the query optimizer from two aspects as follows.

First, if possible, it generates multiple plans with similar execution costs by selecting multiple physical join trees for a given logical join tree. Since we only consider the first two levels of a join tree to determine its locality strength, the query optimizer tries to generate all possible sub-trees for the first two levels, instead of only selecting the one that has the smallest cost. These sub-trees may have different physical operators for each join node. If a sub-tree is allowed to participate the final plan generation, it must satisfy the following condition. Its cost must be no higher than the smallest cost of these sub-trees by a threshold value. We set it as 30%, considering that a weak-locality plan can slow down a strong-locality plan by up to about 30%, as shown in Figure 4. With this method, multiple candidates (with different join trees) can be generated.

Second, the query optimizer selects the best one from these candidates with a selection policy on the basis of two metrics: plan locality strength and execution cost. The query optimizer estimates locality strengths of these candidates using the method in Section 4. Our plan selection policy works cooperatively with the scheduling policy. (1) The query optimizer gives the highest priority to strong-locality plans, and selects the one with the smallest execution cost, considering that two such plans have low performance degradations if they co-run. (2) If there is no strong-locality plan, the query optimizer selects the one with the smallest cost from weak-locality plans, considering that such a plan will not affect or be affected by another weak-locality plan if they co-run. (3) If there is no weak-locality plans, the query optimizer selects the moderate-locality plan with the smallest execution cost.

5.2 SLS: a Co-scheduling Policy in Scheduler

Motivated by the performance results by co-running queries with different locality strengths shown in Section 4, we propose a co-scheduling policy, called *SLS (Same Locality Strength)*, based on the following observation. A weak-locality query can affect a query with strong/moderate locality more severely (due to cache pollution) than a query with strong/moderate locality does (due to capacity contention). With SLS, the

DBMS maintains two plan queues. One is for strong/moderate-locality plans and the other is for weak-locality plans. SLS selects plans from the same queue to co-run, instead of plans from different queues. Specifically for the first queue for strong/moderate-locality plans, SLS always co-schedules two plans with the smallest hash table sizes.

We use an example to demonstrate SLS's effectiveness. Four instances of PLQ_3 are to be executed. Two of them are hash joins with strong localities, and the other two are index joins with weak localities. Each hash join instance uses a hash table of 1.13MB. In the worst case, each hash join instance co-runs with an index join instance. As we have shown in Figure 4, the execution times of hash join instances can be increased by over 30%, compared to the cases in which they are not co-scheduled with other queries. However, by considering query locality strengths, SLS will co-schedule two hash join instances, and co-schedule two index join instances. Comparing with the cases in which they are not co-scheduled with other queries, the performance of hash join instances is degraded by only 5%, and the performance of index join instances is degraded by only 6%.

The example shows that SLS accelerates query executions by avoiding performance degradations caused by cache pollution. However, co-scheduled queries can still suffer performance degradation caused by capacity contention, especially when moderate locality queries are executed. To minimize performance degradation caused by cache conflicts, cache partitioning is required, as introduced in the next section.

6. MCC-DB WITH CACHE PARTITIONER

The cache partitioner is the component to reduce cache conflicts in the query execution phase. Through cache partitioning, we can control the allocation of the shared cache space among queries, and eliminate cache pollution caused by weak-locality queries. In this section, we will first introduce the cache partitioning mechanism, and then present how to re-design the plan selection and the scheduling policy.

6.1 Cache Partitioning Mechanism

Because hardware-based cache partitioning is not available in any commercial processor, we provide a software mechanism that essentially emulates page-level cache partitioning based on a well accepted OS technique, *page coloring* [30].

The basic idea is outlined as follows. A physical memory address contains several common bits between the cache index and the physical page number. These bits are referred to as *page color*. The last level cache is divided into multiple non-overlapping regions, each of which corresponds to a page color. The memory pages of the same color are mapped to the same cache region. Thus, by assigning different page colors to memory objects and/or threads, we can partition the cache space among them. For example, we can assign a subset of colors to an object by placing them in the memory pages of the selected colors, the available cache space that can be used by the object would be limited in the corresponding cache regions. In our platform, the four-core Intel Xeon processor has two 4MB, 16-way set associative L2 caches, each of which is shared by two cores. The memory page size is 4KB, so we have at most 64 colors ($\frac{cache\_size}{page\_size \times cache\_associativity}$).

We implemented the cache partitioning mechanism in the Linux kernel 2.6.20. In our implementation, we only use five least significant color bits in a physical address. Therefore, we have 32 different colors and each color corresponds to a chunk of 128KB L2 cache space. A page color table is maintained to guide the virtual-physical page mapping for threads sharing the same virtual memory space. Each entry in the table specifies a set of colors that the virtual page can be mapped to. Each thread has a pointer in its task structure pointing to the page color table. We also modified the buddy system in the memory management module of the Linux kernel, which is in charge of mapping virtual pages to physical pages, so that the physical pages can be allocated as specified in the page color table. A set of new system calls are added to allow updating the page color table at the user level. Applications, such as a DBMS, can use the system calls to enforce cache partition decisions for its global and heap objects.

6.2 The Interface between DBMS and OS

Essentially, the OS only provides a facility to support cache partitioning. The DBMS is responsible for deciding how to allocate cache space among queries. We design a special interface between the DBMS and the OS, which wraps the low-level system calls for updating the page color table. Each process is associated with a color specification file, which specifies how many colors a global or a heap object is allocated (which corresponds to the amount of cache space). Non-specified objects in all processes are assigned with a default number of page colors. A query execution process is allowed to re-write its color specification file at any time, but it must invoke the system call to notify the OS to update its page color table and re-partition the cache.

The default colors allocated to each query are the colors reserved for the shared buffer pool. That means, any data object in each process is mapped to this reserved cache space that is shared by all processes, unless it is assigned with extra page colors. Each process can manage color allocation for its private data structures (e.g. hash tables). For a query plan with weak locality, its process is not allowed to use more colors than the default ones for the buffer pool. For a query plan with strong/moderate locality, its hash tables (allocated in a heap object) can use all 32 colors. After the query optimizer estimates the locality strength of a query plan, it will invoke the system call to update the page color table if the plan has strong/moderate locality.

The buffer pool does not need too much cache space because data accesses to it (including sequential scans and index scans) have weak locality from the viewpoint of the LLC. The minimum number of colors allocated to it is to satisfy its physical memory need. In our experiment, we use 1600MB buffer pool, therefore we allocate four colors for it (corresponding to 2GB memory space). Our experiments show that this setting performs well in practice.

6.3 Query Optimization and Scheduling with Support of Cache Partitioning

With strong support from the OS kernel, the plan selection policy and the co-scheduling policy in the DBMS need to be re-designed to exploit the optimization opportunities brought by cache partitioning. We first present the co-scheduling policy and then the plan selection policy, because the latter is based on the requirements of the former.

Without cache partitioning, the scheduler can only isolate strong/moderate-locality plans and weak-locality plans, so that weak-locality plans would not interfere with plans with stronger locality. Unfortunately, performance degradation may still happen due to capacity contention among co-

running moderate-locality plans. Cache partitioning provides new scheduling opportunities. It makes locality-driven cache allocation possible and eliminates cache pollution by isolating cache space used by plans with different localities. With cache partitioning, a weak-locality plan would not affect a strong/moderate-locality plan even when they are co-running (we will show evidence in the next section). Meanwhile, the weak-locality plan can still retain its performance, since it is insensitive to cache size.

Here we propose a new scheduling policy, called *Mixed Locality Strength* (MLS), to leverage the caching partitioning mechanism. MLS must work together with the cache partitioner. Similar to SLS, the DBMS still maintains two plan queues, one for strong/moderate-locality plans, and the other for weak-locality plans. MLS always co-schedule a pair of plans from different queues. During execution, cache partitioning is used to protect the executions of strong/moderate-locality plans. If one of the two queues is empty, MLS does the same co-scheduling as SLS – two plans from the same queue are co-scheduled.

To make MLS work well, we propose a new plan selection policy. As introduced in subsection 5.1, the query optimizer still generates multiple candidate plans (their cost differences are under 30%). The main idea of the new policy is to let the query optimizer generate the same number of strong/moderate-locality plans and weak-locality plans as possible as it can, so that MLS can assign each strong/moderate-locality plan a weak-locality co-runner. If there are more weak-locality plans than strong/moderate-locality plans, the query optimizer selects the plan with the smallest cost from those strong/moderate-locality candidates, and vice versa. If there is no required candidate, the query optimizer selects the one with the smallest cost, as a conventional query optimizer does. In addition, if the two queues are balanced, the optimizer gives its priority to strong-locality candidates.

Finally, we need to point out that solely applying cache partitioning is sub-optimal, and must work together with MLS. If the execution scheduler incorrectly co-schedules two plans both with moderate locality, then cache partitioning cannot reduce performance degradation caused by capacity conflict. Similarly, if two plans both with weak locality are co-scheduled, cache partitioning can hardly help either, since such plans are not sensitive to cache size. Only when the execution scheduler uses MLS to co-schedule plans with different locality strengths, cache partitioning can improve performance by eliminating cache pollution.

7. PERFORMANCE EVALUATION

In this section, we first evaluate how cache partitioning can reduce cache conflicts. Then, we compare the performance of the scheduling policies without considering and with considering cache partitioning. After that, we evaluate the effectiveness of the query optimizer. Finally, we evaluate our locality estimation method using standard SSB queries.

7.1 Evaluation of Cache Partitioning

We examine how cache partitioning can improve execution performance of a moderate-locality query by reducing cache pollution caused by weak-locality queries. We use the queries with a 2GB SSB database introduced in subsection 4.2. The moderate-locality query is a hash join for PLQ_5 with a 1.86MB hash table, and we denote it as ML_{HJ} (hash join with moderate locality). We select two different types of weak-locality queries. One is a hash join for PLQ_{25} with a 18.6MB hash table (the PART table is from a 4GB database), and we denote it as WL_{HJ} (hash join with weak locality). The other is an index join for query PLQ_{25}, and we denote it as WL_{IJ} (index join with weak locality). We tune the table sizes so that ML_{HJ} and WL_{HJ} have similar execution times when they are running alone, and ML_{HJ} and WL_{IJ} have similar execution times when they are running together. Table 2 lists execution times and L2 miss rates of the three queries when they are running alone. It clearly shows that ML_{HJ} has a much lower miss rate than WL_{HJ} and WL_{IJ}.

	ML_{HJ}	WL_{HJ}	WL_{IJ}
Execution Time	42.19s	42.44s	62.38s
L2 Miss Rate	6.03%	34.82%	37.47%

Table 2: Query execution times and L2 miss rates

We conduct two tests: co-running ML_{HJ} and WL_{HJ}, and co-running ML_{HJ} and WL_{IJ}. In each test, we examine query execution times and L2 miss rates of two co-runners when shrinking L2 cache space allocated to the weak-locality queries [5], from 4MB (32 colors) to 512KB (4 colors). Figures 5 (a) - (d) show the curves for the two metrics in the two tests, respectively. We have the following three major observations. First, without cache partitioning (i.e. the weak-locality co-runner uses 32 colors), ML_{HJ} suffers significant performance degradation. In the first test, ML_{HJ}'s L2 miss rate is increased to 19%, and its execution time is increased to 54s. In the second test, ML_{HJ}'s performance is degraded by a larger degree, because WL_{IJ} has a longer execution time than WL_{HJ}. Its L2 cache miss rate and execution time are increased to 28% and 66s, respectively. Second, performance degradation of ML_{HJ}, reflected by both execution times and L2 miss rates, is reduced if cache partitioning is used. For example, when WL_{IJ} uses only 4 colors, the execution time of ML_{HJ} is reduced by 33%, compared with the case without cache partitioning. Third, despite the difference in query types, both WL_{HJ} and WL_{IJ} are insensitive to the cache space size allocated to them due to their weak locality nature. They can nearly retain their performance, in terms of both execution time and L2 miss rate, when the cache space allocated to them is reduced.

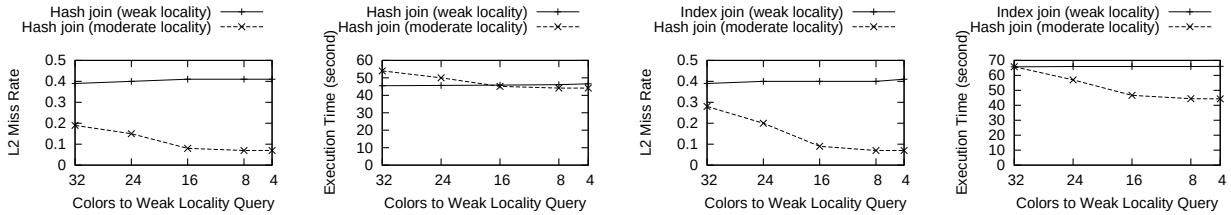
We can draw two conclusions from the experiment. (1) A weak-locality query can cause significant cache pollution so that a moderate-locality query suffers significant performance degradation. (2) Via eliminating cache pollution, cache partitioning can protect the performance of the moderate-locality query, while the performance of the weak-locality query is hardly sacrificed.

7.2 Evaluation of Scheduling Policies

We evaluate the effectiveness of the execution scheduler and compare two scheduling policies: SLS and MLS. Recall that, SLS co-schedules queries with similar locality strengths without considering cache partitioning, while MLS co-schedules queries with different types of locality strengths, assisted by cache partitioning.

We use four queries (PLQ_2, PLQ_6, PLQ_{11}, and PLQ_{24}) with a 2GB SSB database. For each query, we execute four instances (two hash join plans and two index join plans). The hash table sizes of the four queries are 0.78MB, 2.26MB,

[5]We are actually shrinking the cache space for the buffer pool. But the hash table in ML_{HJ} can use all colors.



(a) Test 1: Miss Rates (b) Test 1: Times (c) Test 2: Miss Rates (d) Test 2: Times

Figure 5: L2 cache miss rates and execution times of co-running queries with different locality strengths when using cache partitioning to reduce the colors to the queries with weak locality.

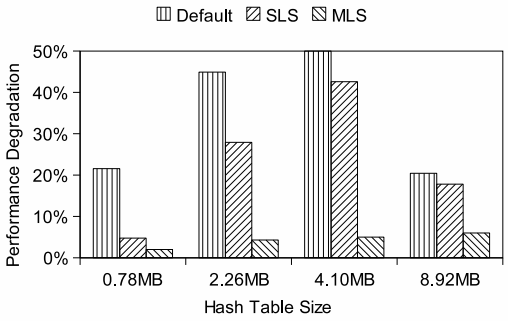


Figure 6: Performance degradations of four hash joins by different scheduling policies.

4.10MB, and 8.92MB, respectively. Based on their locality strengths, SLS co-schedules two hash join plans and then two index join plans. However, MLS co-schedules a hash join plan with an index join plan, while cache partitioning is working cooperatively to limit the cache space allocated for the index join plan. For the comparison against a conventional DBMS system, which does not consider query execution scheduling, we also co-run a hash join plan and an index join plan without any support from cache partitioning. We focus on analyzing the performance degradations of hash join plans caused by cache contention, and their execution times when running alone are the baselines. We did not report the test results for index join plans since they are insensitive to cache conflicts.

Figure 6 shows the performance degradation of the hash join plan for each query in the above three cases (the first one is the default case without any scheduling, the second one is with SLS, and the last one is with MLS). For each query, it shows much better performance under SLS or MLS than it does without any scheduling policy. Moreover, the experiment shows that MLS with consideration of cache partitioning can improve query performance at a larger degree than SLS does, especially when the queries have large hash tables. For example, when hash table size is 8.92MB, the performance of SLS is only slightly better than the case without any scheduling. However, MLS can still significantly outperform the case without any scheduling.

The performance difference between SLS and MLS can be explained as follows. SLS improves query performance because index join plans have no chance to hurt hash join plans. It works when capacity conflicts between small hash tables do not exist, or are less significant than cache pollution caused by index join. However, MLS with cache partitioning can almost eliminate cache pollution so that the performance of hash join plans can be guaranteed. The experiment results show that the DBMS itself can only reduce cache conflicts in a limited scope. Minimizing cache conflicts must rely on the cache partitioning support from the OS.

7.3 Evaluation of Query Optimization

We evaluate the effectiveness of the query optimizer. We execute four instances of query PLQ_{13}. We select this query because its hash join plan (hash table size is 4.9MB) and index join plan have similar execution costs estimated by the query optimizer (the cost of index join is higher than that of hash join by 17.2%). Since all the instances are from the same query, a traditional query optimizer only selects the same plan for them (e.g. hash join plan in PostgreSQL).

When running alone, the execution times of the hash join plan and the index join plan are 10.27s and 11.39s, respectively. However, due to cache conflicts, the average time of two concurrent hash join plans is 13.61s, and the average time of two index join plans is 12.64s. In contrast, the query optimizer in MCC-DB generates two index join plans and two hash join plans. Then, MLS co-schedules a hash join plan with an index join plan, under the cooperation of cache partitioning to protect the hash join execution. With the combined efforts, the average execution time of the four instances is decreased to 11.36s (reduced by 16.5%), compared with the default case of using four hash join plans.

This experiment shows that when there are similar execution costs between candidate plans, the query optimizer plays a key role of determining whether the execution scheduler and the cache partitioner can find desirable query combinations.

7.4 Evaluation of Locality Estimation

We have used standard SSB queries with an 8GB database to evaluate MCC-DB for two goals: (1) to examine the accuracy of our locality estimation method for complex queries with multi-way joins and (2) to demonstrate the effectiveness of cache partitioning for these queries. Among all the 13 SSB queries, we studied 7 long-running queries (whose execution times are all more than 19s), but did not use remaining queries (whose execution times are all less than 7s). Except Q1.1, all the 7 queries contain 4-way or 5-way joins, and they can be classified into two groups. The query optimizer prefers index join plans for Q1.1, Q2.1, Q3.2, and Q4.3 in the first group, while it prefers hash join plans for Q3.1, Q4.1 and Q4.2 in the second group. In the first group, the four queries with index join plans are estimated to have weak locality. In the second group, determined by the accumulated size of their

first two hash tables in their join trees, Q3.1 and Q4.1 are estimated to have moderate locality (the accumulated sizes are 2.5MB and 2.02MB, respectively), while Q4.2 is estimated to have strong locality (the accumulated size is 1MB).

To evaluate the accuracy of locality estimation, we co-ran each index join query from the first group with each hash join query from the second group, and measured performance degradation compared with them running alone. For each pair of queries, we examine the effectiveness of cache partitioning to protect the hash join execution. We summarize 4 major observations from the test results. (1) The four index join queries have low performance degradation (from 7% to 12%) when each of them is co-running with any of hash join queries. (2) Both Q3.1 and Q4.1 suffer significant performance degradation (from 29% to 35%) when each of them is co-running with any of index join queries. (3) Compared with Q3.1 and Q4.1, Q4.2 has lower performance degradation (from 16% to 22%). (4) Compared with the cases without cache partitioning, MCC-DB accelerates the three hash join queries by 11% (co-running Q4.2 and Q2.1) to 21.5% (co-running Q3.1 and Q1.1), while the four index join queries are only slowed down by up to 4% (Q3.2). In addition, in order to examine the locality difference of the three hash join queries we co-ran the same two instances of each of them, and found that (1) Q3.1 and Q4.1 have 14% - 15% performance degradations and (2) Q4.2 has only a 5% performance degradation. These results, measured by both the performance degradation and the effectiveness of cache partitioning, show that the estimated locality strengths of these queries are consistent with their runtime behaviors.

By examining executions of the three hash joins, we confirmed that estimated hash table sizes by the query optimizer are very close to their actual sizes. The maximal difference of hash table size estimations is only 11.8% (Q4.1 and Q4.2), which does not affect the determination of query locality strength. The difference is mainly due to the selectivity estimation error of a selection condition on the CUSTOMER table. We should also note that, despite the strong locality nature of Q4.2, the accumulated size of all hash tables in its join tree is up to 11.6MB, which is very close to the value for a weak-locality query. This reflects that our locality estimation method is reasonably accurate by only considering two level joins. Actually, for all the three queries, only 0.96% - 3.8% of tuples from the fact table can reach the third level of join, and thus the impact of hash tables after the second-level join can be negligible. The experiment results show the effectiveness of the locality estimation in MCC-DB.

8. RELATED WORK

Our work is related to two research areas: (1) database research on optimizing DBMSs for memory hierarchy and (2) system research on improving the shared cache utilization on multi-core processors. The two areas are traditionally distinct, but the gap on common system concerns between them has been narrowed recently.

8.1 Optimizing DBMS for Memory Hierarchy

Database researchers have proposed many solutions to improve database performance with new hardware trends. In the single-core era, there is substantial work on analyzing and solving the problem of the long memory access latency that is a key performance bottleneck for a DBMS [2][3]. Cache-optimized index structures were studied in [27][11][33]. Cache-aware partitioning [3] and prefetching [5] techniques were proposed to improve hash join implementations. Zhou and Ross studied how to improve instruction cache utilization [34]. Manegold, et al. [22] proposed generic models to estimate database operation costs according to the memory hierarchy. He and Luo studied cache-oblivious query processing techniques [13]. These research projects aim to make a single query benefit from the cache, which is orthogonal to our work. We focus on improving the shared cache utilization for concurrent queries that have different locality strengths.

New research problems emerge when running a DBMS on the multi-threading and multi-core systems. For simultaneous multi-threading processors, Lo, et al. [19] analyzed the impact of inter-thread cache interference on database performance. A work-ahead set strategy was proposed to overlap computation in the main thread and long memory access latency in a 'helper' thread [32]. For multi-core processors, Hardavellas, et al. [12] found that the L2 hit latency becomes the new performance bottleneck. Cieslewicz and Ross [7] proposed how to optimize the aggregation operation on multi-cores. Qiao, et al. [24] introduced a scheduling technique to cooperate multiple memory scans to reduce pressure on memory bandwidth. None of these research projects focus on the LLC conflict problem on multicore processors.

8.2 OS Scheduling and Cache Partitioning

In the system research area, various solutions were proposed to improve the shared cache utilization on multi-core processors. Two topics are related to this work: thread scheduling and cache partitioning.

Fedorova, et al. [8] extended the OS scheduler to co-schedule a group of queries with the minimal cache misses. They further proposed to adjust threads' time slices according to their cache usage [9]. Cache partitioning for applications with different locality strengths can be implemented in hardware or in OS. A hardware scheme was proposed to use special circuits to monitor cache accesses and to enforce way-based partitioning [25]. OS-based cache partitioning with page coloring were studied in [28][18]. We have recently made efforts to bridge program analysis and cache partitioning in the OS to improve LLC performance for scientific computing applications [20].

9. DISCUSSIONS AND CONCLUSION

MCC-DB presented in this paper makes an effective collaborative effort between the DBMS and the OS to minimize the shared cache conflicts in multi-core processors.

We have demonstrated that there are two types of LLC conflicts degrading query execution performance in multi-core processors. First, cache pollution conflicts are mainly caused by a flooding of weak-locality data accesses. Our study shows that cache pollution is the most serious performance concern in LLC. By experiments, we have also shown that MCC-DB can effectively minimize cache pollution conflicts, which are the most harmful. Second, under a cache-pollution-free condition, another type of LLC conflict is caused by limited cache capacity as queries with strong or moderate locality compete for the shared cache space. Capacity conflict is normally less harmful than cache pollution. Under the MCC-DB framework, the query scheduler can be naturally extended to dynamically assign queries by a cache capacity-aware policy to minimize cache conflicts for many cores.

Our locality estimation method has been tested on star schema based warehouse queries. We do not consider bushy

join trees or plans with multiple segments separated by blocking operators [21][4]. Such plans could have different locality strengths at different segments. MCC-DB can be extended with the granularity of plan segment by adopting dynamic re-optimization techniques during query executions [17].

We plan to enhance the scheduling capability by first identifying multiple queries with shared data sets and individual queries with private data sets, and then making efforts to to protect the shared data sets for their queries to further minimize cache misses.

10. ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation under grants CCF-0620152, CCF-072380, and CNS-0834393. We thank the anonymous referees for their comments. We also thank our colleague Bill Bynum for reading this paper and his comments.

11. REFERENCES

[1] S. Adee. the data: 37 years of moore's law. *Spectrum, IEEE*, 45(5):56–56, May 2008.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[3] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[4] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating progress of long running SQL queries. In *SIGMOD*, 2004.

[5] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17, 2007.

[6] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB*, pages 127–141, 1985.

[7] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[8] A. Fedorova, M. I. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX*, 2005.

[9] A. Fedorova, M. I. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*, 2007.

[10] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *ICDE*, pages 406–417, 1994.

[11] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B^+-trees. In *SIGMETRICS*, pages 283–294, 2003.

[12] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, pages 79–87, 2007.

[13] B. He and Q. Luo. Cache-oblivious databases: Limitations and opportunities. *ACM Trans. Database Syst.*, 33(2), 2008.

[14] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the clock replacement. In *USENIX*, pages 323–336, 2005.

[15] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, pages 31–42, 2002.

[16] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[17] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[18] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, pages 367–378, 2008.

[19] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *ISCA*, pages 39–50, 1998.

[20] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-OLP: Improving last level cache performance through software-controlled object-level partitioning. In *PACT*, 2009.

[21] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. Toward a progress indicator for database queries. In *SIGMOD*, 2004.

[22] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, pages 191–202, 2002.

[23] P. O'Neil, B. O'Neil, and X. Chen. The star schema benchmark (SSB). 2007. http://www.cs.umb.edu/ poneil/StarSchemaB.PDF.

[24] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core CPUs. *PVLDB*, 1(1):610–621, 2008.

[25] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, pages 423–432, 2006.

[26] R. Ramamurthy and D. J. DeWitt. Buffer-pool aware query optimization. In *CIDR*, pages 250–261, 2005.

[27] J. Rao and K. A. Ross. Making B^+-trees cache conscious in main memory. In *SIGMOD*, 2000.

[28] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *MICRO*, 2008.

[29] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik. One size fits all? part 2: Benchmarking studies. In *CIDR*, pages 173–184, 2007.

[30] G. Taylor, P. Davies, and M. Farmwald. The TLB slice - a low-cost high-speed address translation mechanism. In *ISCA*, pages 355–363, 1990.

[31] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-all replacement for a multilevel cache. In *FAST*, 2007.

[32] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *VLDB*, 2005.

[33] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, 2003.

[34] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD Conference*, pages 191–202, 2004.