Cache Hierarchy-Aware Query Mapping on Emerging Multicore Architectures

Ozcan Ozturk, Umut Orhan, Wei Ding, Praveen Yedlapalli, and Mahmut Taylan Kandemir, *Fellow, IEEE*

Abstract—One of the important characteristics of emerging multicores/manycores is the existence of "shared on-chip caches," through which different threads/processes can share data (help each other) or displace each other's data (hurt each other). Most of current commercial multicore systems on the market have on-chip cache hierarchies with multiple layers (typically, in the form of L1, L2 and L3, the last two being either fully or partially shared). In the context of database workloads, exploiting full potential of these caches can be critical. Motivated by this observation, our main contribution in this work is to present and experimentally evaluate a cache hierarchy-aware query mapping scheme targeting workloads that consist of batch queries to be executed on emerging multicores. Our proposed scheme distributes a given batch of queries across the cores of a target multicore architecture based on the affinity relations among the queries. The primary goal behind this scheme is to maximize the utilization of the underlying on-chip cache hierarchy while keeping the load nearly balanced across domain affinities. Each domain affinity in this context corresponds to a cache structure bounded by a particular level of the cache hierarchy. A graph partitioning-based method is employed to distribute queries across cores, and an integer linear programming (ILP) formulation is used to address locality and load balancing concerns. We evaluate our scheme using the TPC-H benchmarks on an Intel Xeon based multicore. Our solution achieves up to 25 percent improvement in individual query execution times and 15-19 percent improvement in throughput over the default Linux-based process scheduler.

Index Terms—Query, multicore, schedule, cache, architecture

1 Introduction

Growing performance gap between processors and main memory has made it worthwhile to consider off-chip data accesses in query processing [2], [3], [5], [15]. Especially in multi-query environments, exploiting data-sharing opportunities among concurrent queries can be critical for effective utilization of the underlying shared memory hierarchy. Given a set of queries, there may be a common retrieval operation for several cases to the same data. A query can benefit from the data previously loaded into the shared cache/memory space by another query. However, if these queries are scheduled independently, it is very likely that the same data is brought from off-chip memory to on-chip caches multiple times, thereby consuming off-chip bandwidth and slowing down overall execution. In addition, resource allocation and scheduling in multi-query environments are typically performed by the operating system (OS). For example, Linux task scheduler is oriented towards load balancing and can dynamically change the affinity of running processes (task migration) to utilize each core at its maximum. As it has no in-depth understanding of how database queries are processed individually, an OS scheduler may not exploit potential data sharing opportunities between two or more different queries in a shared cache. Even worse, treating database queries as ordinary processes and, consequently, scheduling them in a traditional manner may penalize concurrently-executing queries at runtime and may lead to degradation in overall system throughput. In shared-memory multicore architectures, on-chip cache performance is a major factor as far as workload performance is considered. In fact, application behavior can exhibit dramatic variations on different on-chip cache hierarchies depending on mapping and scheduling plans [20]. Moreover, cache contention due to hardware resource constraints has already been identified as a challenge that must be addressed in query processing context [23]. Therefore, running these servers on multicore architectures raises an important question from the *data-locality perspective*: how to schedule concurrently-running queries across available cores in order to better utilize the underlying shared cache hierarchy and improve the overall throughput of the system?

Our *goal* in this study is to make concurrent multi-query execution in conventional relational database systems effectively benefit from chip-level parallelism provided by emerging multicore architectures in a locality-aware fashion and, as a result, improve the overall throughput of the system. More specifically, we map queries to cores in such a way that cores can utilize the shared data kept in caches.

We address two main concerns in optimizing multi-query scheduling: *affinity* and *load balancing*.

In a relational database management system (DBMS), there are multiple ways of executing a given (SQL) query. When a

- *O. Ozturk is with Bilkent University, Bilkent, Ankara 06800, Turkey. E-mail: ozturk@cs.bilkent.edu.tr.*
- *U. Orhan is with Amazon Inc., Seattle, WA 98109-5210. E-mail: uorhan@gmail.com.*
- *W. Ding is with Qualcomm Innovation Center Inc., San Diego, CA 92121. E-mail: wding109@gmail.com.*
- *P. Yedlapalli is with VMware Inc., Palo Alto, CA 94304. E-mail: praveen.yadlapalli@gmail.com.*
- *M. T. Kandemir is with Pennsylvania State University, State College, PA 16801. E-mail: kandemir@cse.psu.edu.*

Manuscript received 14 July 2015; revised 30 Mar. 2016; accepted 7 Apr. 2016. Date of publication 1 Sept. 2016; date of current version 16 Feb. 2017. Recommended for acceptance by A. Gordon-Ross.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2016.2605682

query is submitted, query optimizer generates an ordered set of steps used to access data in an efficient manner. This set of operations is called a *query plan*, or an *execution plan*, or simply a *plan*. For example, a join operation among two tables followed by a selection is a possible plan for a specific query. Or alternatively, a selection in a table followed by a join with another table may give the same correct result. Each operator has different cost in terms of time it takes to complete the task. Therefore, multiple alternatives for the same query with widely varying performance may exist. In our approach, we use the estimated cost (execution time) for each operator and similarly, the total estimated cost (execution time) for a certain execution plan, to compare different alternatives.

If we know (i) the execution plan of each query, (ii) an estimated cost for each operator/plan, and (iii) the target multicore platform in advance, we can suggest compile-time assignments of queries to domain affinities. In this work, we try to solve query-to-core mapping problem on an underlying cache topology.

These assignments can improve data locality on shared caches over dynamic, OS based scheduling and lead to significantly less cache conflicts as well as reduced number of off-chip data accesses. On the other hand, a simple compile-time multi-query scheduling scheme that relies only on data sharing relations between queries tends to ignore dynamic modulations across workloads of different processors. At runtime, core utilizations can be reduced when a static scheduling scheme is employed and we may even end up with idle cores when queries have diverse execution times. Consequently, we also need to better utilize the available processing units through load balancing.

The techniques that we discuss in this paper identify common data retrieval operations in multi-query workloads and build *affinity relations* between queries that represent possible data sharing at runtime. Affinity relations are represented using an *undirected weighted graph*, where each node represents a query and an edge between two nodes indicates possible data sharing among the corresponding queries. Edge weights are calculated from the query plan estimations provided by the query optimizer. Using this graph, we then invoke a *hierarchical clustering algorithm* to generate *query-to-domain affinity mappings*. An *domain affinity* in this context refers to a particular cache structure bounded by a specific level of the cache hierarchy. It can be a private cache or a cache shared by multiple cores, and each *domain affinity level* covers all caches at that level, e.g., domain affinity level 2 includes all L2 caches. According to the generated mappings, each query is executed only on the cores that are connected to the corresponding domain affinity.

Our clustering strategy creates partitions starting from bottom level cache (close to main memory) until it has the same number of partitions as the number of target domain affinity levels. More specifically, this strategy tries to create the exact number of partitions as requested while maximizing the total edge weight within a partition (i.e., the amount of data sharing) and minimizing the total weight of cutting-edges. When moving to upper levels, the strategy takes the parent partition and divides it into the same number of available caches in the upper level. We further enhance this scheme by introducing vertex weights to model runtime working memory requirements of queries so that we can balance queries and reduce cache thrashing. Our proposed clustering strategy works as expected when the number of queries in the given workload is equal to or less than the available cores in the target architecture. In such a case, a particular core can be dedicated to a single query. However, when we increase the workload size, static domain affinity-query mappings can result in idle cores at runtime, especially when the queries in the workload have diverse execution times. A workload on a domain affinity may be finished before other domains, and consequently, the overall system utilization gets reduced compared to a dynamic OS-based scheduler since static mapping does not consider runtime reassignments. Motivated by this observation, we extend our clustering approach with an integer linear programming (ILP) based load balancing step where we try to balance the loads assigned to different domain affinities.

We implement our scheduling scheme as a middleware in PostgreSQL 8.4 [30], which takes a batch of queries to be executed in parallel and the cache topology information of the target multicore architecture as inputs. As a motivating point, this kind of batch scheduling schemes can be applied into real-world scenarios where several database users run a fixed set of queries for generating daily reports from a data warehouse. Hence, we evaluate our approach with workloads consisting of OLAP queries provided by the TPC-H benchmarks [37]. To summarize, we make two main contributions in this paper:

- We present a *cache topology aware* multi-query scheduling scheme for multicore architectures. This approach defines affinity relations between queries and assigns closely related queries into similar domain affinities in order to effectively utilize the on-chip cache hierarchy by exploiting data locality throughout the cache hierarchy.
- We explain how this scheduling strategy can be extended to reduce cache thrashing effects of concurrent queries sharing the same cache structures as well as to tolerate load balancing concerns brought by static domain affinity mappings.

Our experimental results on an Intel IvyBridge-EN multicore system indicate that the proposed scheduling algorithm achieves up to 25 percent improvement in query execution time and 15-19 percent improvement in overall system throughput. To the best of our knowledge, this is the first work that recognizes and take advantage of the disparities between different on-chip cache topologies for scheduling multiple queries in emerging multicore architectures.

In the next section, we give a detailed comparison of our approach with the prior related efforts. A brief background on multicore architectures, shared caches and data reuse can be found in Section 4. In Section 2, we motivate the cache topology aware multi-query scheduling problem in the context of emerging multicore platforms. Sections 5 and 6 present the details of our proposed multi-query scheduling scheme. In Section 7, we give an experimental evaluation of this scheme using commercial multicore machines. The paper is concluded in Section 8 with a summary of our major observations and possible future research directions.

2 Motivation

This section presents results motivating for a cache hierarchy-aware multi-query scheduler for multicores. For this

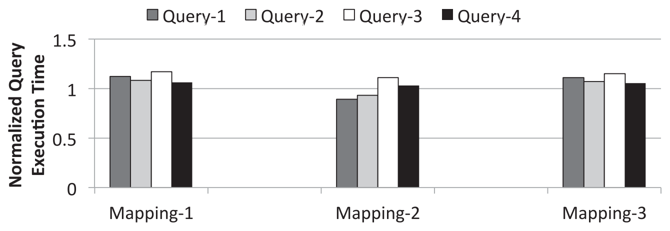


Fig. 1. Query execution times with different mappings.

experiment, we use four queries from TPC-H [37] and consider three different mappings to a dual-socket, Intel Ivy-Bridge-EN based architecture, where the cores in each socket have private L1 and L2 caches and they share the last-level (L3) cache, as shown in Fig. 2a. While there are different sharing characteristics among different queries, we observe some level of sharing in all cases tested. We give details of these tests with different queries in our experimental analysis section. The first mapping maps all queries (Queries 1 through 4) to one of the sockets; the second mapping maps Queries 1 and 2 to the first socket and Queries 3 and 4 to the second socket; and the third mapping maps Queries 1 and 3 to the first socket and Queries 2 and 4 to the second socket. In each experiment, each query is executed in a single core. The query execution times with each mapping are plotted in Fig. 1. Each bar is normalized with respect to the case where the corresponding query is executed in an isolated fashion in a core, without running any other query in any other core in the socket.

One can make several interesting observations from these plots. First, although each mapping uses the same number of cores (same parallelism), the execution time of a given query exhibits significant variances depending on the mapping used, indicating that cache performance plays a critical role. Second, when all queries are executed in the same socket, we see that the performance of each query suffers to varying degrees. This is expected and due to contention in the last level cache. However, when we move to the second mapping, we see that the performances of Query 1 and Query 2 improve over isolated executions. This is because of the data (table) sharing between these two queries. In fact, what happens is that the data brought to the last-level cache by one query are reused by the other, which means the former one practically fulfills a prefetching functionality for the other. As a result, both the queries benefit from colocation. On the other hand, Queries 3 and 4 still perform worse compared to their respective isolated executions, as they do not share much data (their results are slightly better than the first mapping because the contention coming from Querys 1 and 2 are eliminated. When we look at the results with the third mapping, we see that they are very similar to those of the first mapping. The marginal improvement (around 1 percent) over the first mapping is due to reduced contention. Overall, these results suggest that careful mapping of queries to cores can improve query execution times. In particular, we would prefer the queries that share data to share cache as well, and similarly, the queries that do not share much data should be mapped to cores that do not share any cache. In the rest of this paper, we present a multi-query scheduling algorithm driven by these goals.

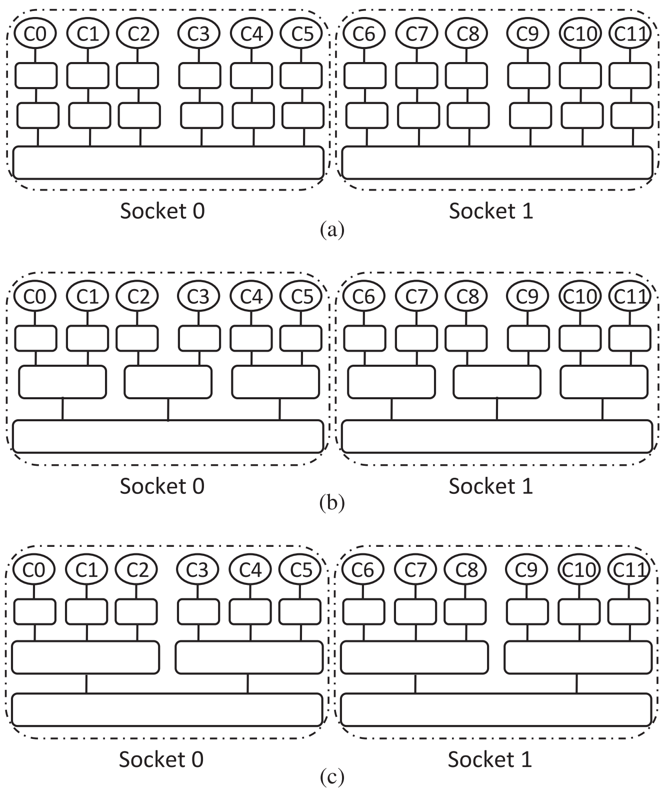


Fig. 2. Different cache topologies with the same number of cores and sockets.

3 Related Work

Query Processing. Several studies are presented for making query processing and database operators aware of on-chip cache spaces in the context of both single core machines [34] and multicore architectures [11], [12], [39]. Acker et al. [1] present an approach that encapsulate parallelism for relational database query execution, which strives for maximum resource utilization for both CPU and disk activities. Liknes [24] investigates database algorithms and methods for modern multi-core processors in main memory environments. Stonebraker et al. [35] and Boncz et al. [7] introduce tuple access and storage optimizations in order to cope with the memory access bottleneck. Albutiu et al. [4] devise a suite of new massively parallel sort-merge (MPSM) join algorithms that are based on partial partition-based sorting. These MPSM algorithms are NUMA-affine as all the sorting is carried out on local memory partitions. Duffy and Essey [13] review the goals of the PLINQ technology, where it fits into the broader .NET Framework and other concurrency offerings. Lee et al. [23] specifically target database queries sharing same on-chip cache structures in multicore architectures. They introduce an OS-level cache partitioning scheme which is based on data access patterns and working memory requirements of the given workload queries. As compared to these works, our approach considers a batch of queries running concurrently (instead of a single query) and exploits data locality opportunities in a global fashion.

Data Sharing. Harizopoulos et al. [22] present a pipelined query engine where a single data retrieval operation serves more than one query in parallel. Petrides et al. [29] propose different representative data-parallel versions of the original database scan and join algorithms to exploit the benefits of using on-chip clustered many-core architectures, and study the impact on the performance when on-chip memory, shared among all cores, is used as a prefetching buffer.

In [31], [32], [41], work sharing opportunities through exploiting common operators across concurrently-running queries are discussed. The goal of our multi-query scheduling scheme is similar to these work sharing approaches from the data locality perspective. However, we focus more on the issues arising due to shared caches and different on-chip cache topologies. Extending our approach with these expert work sharing based approaches can further improve data locality through all levels of the on-chip cache hierarchies in multicores.

In addition, batch scheduling and resource allocation problems have been studied in the scope of parallel database systems [25], [40]. In comparison, our work specifically targets emerging multicore platforms and focuses on the problem of optimizing data locality in shared on-chip cache hierarchies.

4 Multicore Architectures and Data Reuse

Chip multiprocessors paved the way to alternative cache topologies, which means that cache memories can be connected to on-chip cores in a multi-leveled fashion by exhibiting various different patterns. Intel's Dunnington [17] and Harpertown [18] architectures are good examples of this diversity. Dunnington has six on-chip cores whereas Harpertown has four cores. Both architectures have an L1 cache per core and L2 caches are shared by a pair of cores. However, Dunnington adds one more level to Harpertown's cache hierarchy and introduces L3 cache. On the other hand, architectures such as Intel Nehalem [19] can have a completely different topology with private L2 caches. All these three multicore machines have distinct on-chip cache hierarchies which are shared across different number of cores. Today, a server rack can contain more than one of these chips, resulting in parallel systems with large number of cores. Representative cache topologies, each with 12 cores spread over two sockets, are depicted in Fig. 2. A distinguishing characteristics these multicore architectures is the existence of *shared on-chip caches*. Shared caches are preferable to their private alternatives especially when we consider (i) efficient utilization of cache space and (ii) avoiding data redundancy across caches. In particular, depending on their data access/sharing patterns, cache sharing among two processes/threads can be constructive or destructive [6], [8], [9]. Shared caches can cause co-runner applications running on different cores to contest for the available space. In other words, an application, process or thread executing on a particular core can be slowed down by a *co-runner* which uses the same cache space at the same time through a different core. As a result, one can expect that scheduling decisions on multicore architectures can dramatically change the overall system performance. In order to avoid such contentions, one must find an appropriate match of processes. This challenge is often referred to *the application-to-core mapping* problem and has become an active research area [9], [14], [36].

Shared caches make use of the property of *data reuse* in applications. Data reuse is an access to a memory location that has already been accessed previously. The ability of a cache in converting a data reuse into a *cache hit* depends on (i) the parameters of the cache (e.g., capacity, associativity and block size) and (ii) the distance at which the reuse

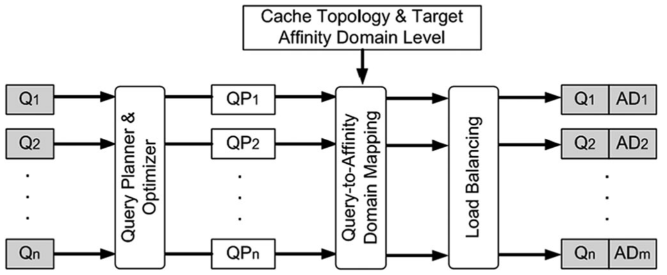

Cache Topology & Target Affinity Domain Level
Q1
Q2
Qn
Query Planner & Optimizer
QP1
QP2
QPn
Query-to-Affinity Domain Mapping
Load Balancing
Q1 AD1
Q2 AD2
Qn ADm

Fig. 3. High level sketch of our cache topology-aware query scheduling approach. Q_i indicates the specific query, whereas QP_i indicates the query plan generated by the query optimizer. After applying our approach, we generate the new query execution, Q_i, on a specific domain affinity AD_i.

occurs, namely, the *reuse distance*. Reuse distance is defined as the number of "unique" memory locations accessed between two contiguous accesses to the same memory location. Smaller a reuse distance is, higher the chances for catching the reused item in the cache (i.e., converting reuse into locality). More specifically, converting the reuse into locality can happen if two queries are accessing the same set of data (or sharing data) and using the same memory location to access this data. Moreover, timing of accesses should also match for locality. In essence, our target with data reuse is to achieve reuse of shared data between different queries. Note that, in our approach, we assume that the queries are completely independent. Our approach can be converted to support such cases but this would require additional constraints and bring other limitations.

5 Proposed Scheduler

5.1 Problem Definition and High-Level View

Our goal in this paper is to present and evaluate a scheduling algorithm which assigns queries of a given batch job to the domain affinities in the target multicore architecture in a cache conscious manner. This *cache hierarchy-aware* scheduler can reduce potential cache contentions among concurrent queries and improve the overall throughput of the system. We define this query-to-domain affinity mapping problem more formally as follows. A query (q_i) to domain affinity (D_j) mapping at level L is defined as

$$M(L) = \{(q_i, D_j) \mid 1 \leq i \leq n_q, 1 \leq j \leq n_L\}, \quad (1)$$

where n_q denotes the number of queries and n_L denotes the number of caches at level L of the target cache topology.

Our scheduling algorithm takes two inputs: a set of query plans to be executed and the underlying cache topology of the target multicore architecture where these queries are processed. The main goal behind the algorithm is to decide which query should be executed on which domain affinity. It tries to evenly distribute the queries among available cores while maximizing possible data sharings through shared caches.

In Fig. 3, we give the high-level description of our automated approach to cache topology aware query scheduling. In the first step, we invoke the PostgreSQL Query Planner and Optimizer. We then analyze the generated query plans to extract possible data sharing opportunities across queries and estimate the amount of memory space to be consumed by each query. In this step, we build a *graph structure* representing the data sharing relationships among queries with respect to cache behavior, and then partition this graph considering the target architecture and the domain affinity level. Finally, we try to balance the load on each domain affinity according to the estimated query execution times. More specifically, we apply a decoupled approach where we define affinity relations between queries and assign closely related queries into similar domain affinities statically. In the second phase, we apply our load balancing technique to consider dynamic modulations across workloads of different processors.

Note that it may not be possible to extract query features for all platforms and query types. Our goal in this work is to show that this would be possible in an environment where the execution plan, estimated cost for each operator/plan, and target multicore platform information is available. While target platform information is easier to obtain, cost for operators and execution plan are harder to estimate. In our implementation, we use the Query Planner and Optimizer module of PostgreSQL to extract the execution plan. Similarly, we estimate the cost of each operator by running experiments with various queries that use the given operator. For example, we apply *Hash Join* with different queries and different data sizes to estimate the unit cost.

5.2 Assumptions

Relational database management systems use various operators to perform required tasks on data. Data is organized as a set of tuples $(d_1, d_2, \dots, d_n)$, where each element d_j is a member of D_j, a data domain. These set of tuples are called relations which enable structured collection of data. One of the major operators used as part of these systems is the *join* operator. More specifically, join generates the set of all combinations of tuples in R and S that are equal on their common attribute names. In mathematical terms, join is a binary operator expressed as $R \bowtie S$, where R and S are relations. For this study, we employ *hash join* as our default join operator. This is because, instead of using nested loop or sort merge joins, PostgreSQL mostly prefers hash joins for executing TPC-H queries when no indices are introduced to the system. Further, employing hash join allows us to test our approach in the presence of private data structures generated by queries at run-time such as hash tables. Each hash table belongs to a particular query and is not shared with other queries. These in-memory tables tend to have short reuse distances during join processing, thus, besides aggregations, they can easily jeopardize the potential benefits brought by on-chip caches by causing contention especially when the cache is used by other hash joins at the same time [23], [31].

A conventional hash join operation consists of two consecutive steps: *building* and *probing*. In the building phase, a hash table is created from the rows of the smaller relation or from the results of another join. Afterward, the other relation is scanned and suitable rows are joined with the ones found in the hash table. The building phase is materialized in the classical hash join method, i.e., probing step is started right after finishing the construction of the hash table.

Despite their drawback of extra memory consumption, we can take advantage of hash join operations for join processing in exploring data sharing opportunities. Specifically, with a query optimizer favoring the left-deep query plans, the materialized nature of a hash join operation can be exploited to maximize data reuse between concurrent queries that are working on same relations. In such a case, scan operations within a query are likely to be executed in the reverse order of the cardinalities of the relations that they scan. When the *selectivities*[1] on shared relations are similar, chances of finding the data in an on-chip cache, which was once brought in by another query, can be improved. Using selectivities when performing joins will result in same hash join orders which will also mean to access the data in a similar order. More specifically, data accesses will follow similar patterns in different queries while creating the joins, thereby increasing the chances of utilizing the data in the cache. As an example, in Fig. 4, query plans of TPC-H querys 5 and 8, which are generated by PostgreSQL query optimizer, are given. One can figure out that these two different query plans have same hash join ordering decisions for same relations.

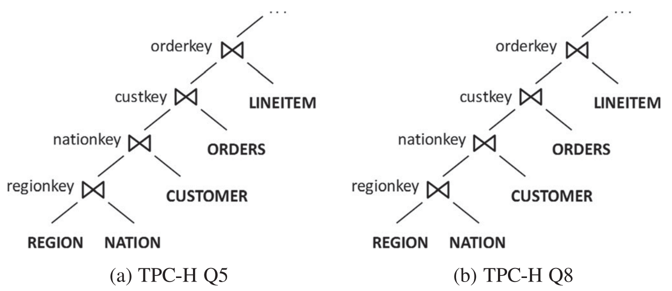


Fig. 4. Ordering in hash join chains depends largely on relation cardinalities. In this example, we have $|LINEITEM| > |ORDERS| > |CUSTOMER| > |NATION| > |REGION|$.

In this study, we statically assign queries to domain affinities and do not handle changes that might happen over time. The proposed technique is dependent on the query optimizer of the database system. The results can be hindered by the wrong selectivity or execution time estimates. Especially, in highly concurrent environments, not only the selectivities but also the execution frequencies of the queries might change at run-time. As part of our future work, we are planning to consider the dynamic nature of the query execution as well.

5.3 Estimating the Amount of Shared Data Between Two Queries

A query reads the data stored on a database management system through scan operations. For example, a sequential scan operation fetches all tuples of a relation starting with the first tuple. Therefore, we can represent the data that a query reads during its lifetime as a set of relations R:

$$\mathcal{R} = \bigcup r, \tag{2}$$

where r denotes a scanned relation, and $\bigcup$ denotes "set union" operation. At this point, one can approximate the total amount of data shared between two queries as follows

$$DataSharing = \sum_{r_i \in \mathcal{R}_1, \mathcal{R}_2} |r_i|, \tag{3}$$

1. Selectivity in a scan operation is defined as the percentage of the filtered tuples over the total number tuples in the relation. In other words, it indicates the percentage of different rows selected as a result of the scan operation.

where $\mathcal{R}_1$ and $\mathcal{R}_2$ represent two set of relations read by distinct queries, and r_i is the relation that scanned by both of these queries. Note that, we calculate the amount of data sharing in terms of *tuples*, instead of using the actual size of the stored data in bytes.

With an unlimited cache size (capacity), *reuse distance* of shared data would not be of any concern. Consequently, once a tuple is brought into the on-chip cache, it would not be kicked out due to a miss and, after the very first miss, any read request for this tuple would be a hit. However, in real-world settings, we must consider the distance between two read operations to the same tuple, as cache capacities are limited. If the distance between two scan operations which read tuples of the same relation is significantly large, then leading scan may displace all existing tuples from the cache and replace them with newer ones before the lagging scan can access them. As a result, these queries may not benefit from data sharing. In other words, if the same data is read by two queries at completely different points in time, then the amount of data shared between the queries might be zero.

In order to address this timing issue, we enhance our initial data sharing model by considering the selectivity of each scan operation. During our experiments, we observed that the execution time of a scan operation is related to its selectivity and, in fact, two scan operations having similar selectivities are more likely to share tuples brought into a cache. On the other hand, even if we ensure that join orderings for the same tables are the same, scan operations may not necessarily occur at the same level of the associated query plan trees. For example, one of the queries might work on a completely different data first, and compute a join among shared relations as the rest. To consider such cases, we calculate vertical differences between scan operations and enable this information for data sharing estimations. More specifically, we capture the level of each scan operation (η) in the query plan, and use these levels in estimating the data sharing. Accordingly, we change our data sharing model to:

$$\begin{aligned} \Delta(LevelDifference) &= 1/(1 - |\eta_1 - \eta_2|), \\ DataSharing(revised) &= \sum_{r_i \in \mathcal{R}_1, \mathcal{R}_2} (|r_i| * \Delta * (1 - |\sigma_1 - \sigma_2|)), \end{aligned} \tag{4}$$

where σ_1 and σ_2 represent the selectivity of two scan operations, and each η gives the order of a scan operation according to the post-ordered query plan tree. As can be seen from this expression, we first capture the amount of data in relations $r_i \in \mathcal{R}_1, \mathcal{R}_2$ read by distinct queries, and scale this with both $\Delta(LevelDifference)$ and selectivity difference, that is $|\sigma_1 - \sigma_2|$. Both of these effect the data sharing inversely. In our framework, we extract the selectivity information by parsing the query execution plan where scan operations are associated with estimated costs and the number of the resulting tuples.

```
                            QUERY PLAN
-------------------------------------------------------------
(1) Sort  (cost=329380.96..329380.97 rows=1 width=27)
     Sort Key: lineitem.l_shipmode
(2) -> HashAggregate(cost=329380.93..329380.95 rows=1 width..
(3)  -> Hash Join(cost=68238.00..329179.33 rows=26879 width..
(4)   Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
(5)   -> Seq Scan on lineitem(cost=0.00..252082.75 rows=268..
       Filter: ((l_shipmode = ANY ('{MAIL,RAIL}'::bpchar[...
(6)    -> Hash(cost=41431.00..41431.00 rows=1500000 width=20)
(7)     -> Seq Scan on orders(cost=0.00..41431.00 rows=150...
```

Fig. 5. A sample query plan for TPC-H Q12.

5.4 Estimating the Working Memory Sizes

For achieving good shared cache performance, it is critical to reduce the amount of memory stalls experienced due to cache misses. Even when several co-runner queries with data sharing are executed, *cache thrashing* may offset the potential benefits of this data sharing. Cache thrashing occurs when the data structures required by each query, such as aggregations and hash tables, overflow the cache. Thus, any data sharing optimization in concurrent query execution needs to target at reducing thrashing effects of non-shared data structures as much as possible.

In order to minimize the cache thrashing effects of the working memory, we estimate the amount of memory space needed by a query during its lifetime. We perform this estimation by exploiting query plan definitions produced by the query optimizer. A node in the PostgreSQL's query plan is associated with the estimated execution time, the number of tuples returned, and the width of a returned tuple in size of bytes. Hence, we can have a general idea about the size of working memory allocated for hash table and aggregation table nodes individually by multiplying the cardinality of the returned tuple set and the corresponding width value. Cache thrashing is more likely to occur during the peak memory consumption periods, and therefore, we estimate the upper bounds of the working memory.

Since Query Planner and Optimizer gives us the query plan after the submission, we can estimate the peak working memory size in the worst case by summing up the estimated working memory sizes of the stages in the query plan. This query plan only indicates the operations that will be performed on the database which will not be sufficient to measure the amount of data or its contents until the actual stage of the plan is executed. In most of our benchmark queries, pipeline stages consist of an aggregation and a generation of the intermediate results that are supplied to this aggregation. These intermediate results are typically generated after a hash join. We can therefore estimate the peak working memory size of a query as

$$H = max\Big(\bigcup |h|\Big), P = |k_i| + |a|, WMS = max(H, P), \tag{5}$$

where P denotes the sum of aggregation table size ($|a|$) and its inputs (k_i), H is the size of the largest hash table created among all other hash tables (hs), and WMS is the estimated working memory capacity demanded by this query.

Consider, as an example, the query plan given in Fig. 5, produced by PostgreSQL's query optimizer for TPC-H Q12. In this query, a hash table on *ORDERS* relation is built first. As indicated in the plan node (*Hash* node at line 6), this table has 1.5 M rows, each of which is 20 bytes, resulting in a table size of nearly 28.6 MB. In the pipelined execution, results fetched in the join operation are provided to the aggregation operation. The sum of the working memory required in the pipelined stages is calculated and found to be less than the size of the hash table generated in the beginning. Consequently, WMS in this case is equal to 28.6 MB.

```
1:  $QP = \{q_0, ..., q_n\}$: query plans
2:  $T$: cache hierarchy tree
3:  $K$: number of available cores/domain affinities
4:  $G$: a graph consisting queries as vertices
5:  $CS = \{cs_0, ..., cs_{k-1}\}$: cluster set (affinity groups)

6:  procedure SCHEDULER($QP, T, K$)                    ▷ Main Routine
7:      $G \leftarrow BuildGraph(QP)$
8:      $CS \leftarrow Partitioner(G, T)$
9:      $CS \leftarrow LoadBalancer(CS, QP, K)$        ▷ ILP Solver
    return $CS$
10: end procedure
```

Fig. 6. Cache topology aware query scheduling.

As mentioned before, we use cache topology as an input to our query mapping approach. The ability of a memory hierarchy in converting a query into a cache hit depends on the data reuse distance and the parameters of the cache such as capacity, associativity and block size. While our approach is designed to consider any memory hierarchy given as a tree representation, we do not include optimizations which depend on the exact cache configuration in terms of block size and associativity, i.e., we do not attempt to answer the question how queries can exploit the associativity, block size, or other cache parameters. However, as an extension to our current implementation we are planning to extend our scheme to include such optimizations using a cache configuration parameter as a three-element tuple $< capacity, size, line >$. In our approach, currently, we are only using the size of the cache as part of the memory hierarchy tree configuration.

We also need to mention that, to reduce energy consumption, the proposed approach can be used with some modifications, at minimizing energy as well (instead of improving query performance). For example, our query scheduler applies a load imbalance coefficient to limit the disparities across the loads of different cores. While this is a desirable property for performance, it is not always true for energy consumption since idle cores can save energy if they are in sleep mode or shut down. However, this option is not evaluated in this paper.

6 Query-to-Domain Affinity Mapping

In this section, we first describe our cache topology-aware multi-query scheduling scheme that uses the estimated amount of data shared among queries. We next enhance this scheme through minimizing cache conflicts by considering the working memory sizes of queries and balancing the loads across different domain affinities. A pseudo-code for the proposed scheduler is given as Fig. 6.

6.1 Exploiting Data Locality and Avoiding Cache Conflicts

We start with building an undirected weighted graph where each query is represented as a vertex. An edge between two vertices has a weight equal to the estimated amount of data sharing using the technique presented in Section 5.3.

To avoid cache thrashing effects of overflowed working memories on a shared cache as much as possible, one has to consider the total amount of memory space allocated to the co-runner queries assigned to a particular domain affinity. In order to model working memory requirements, we slightly modify our graph structure and introduce *vertex weights* representing the working memory sizes of queries. A pseudo-code for how we build this graph structure is given as Fig. 7.

```
1:  $QP = \{q_0, ..., q_n\}$: query plans

2:  procedure BUILDGRAPH($QP$)
3:      $V \leftarrow \emptyset$
4:      $E \leftarrow \emptyset$
5:      for all $q \in QP$ do
6:          $|V_q| \leftarrow Work\_Mem(q)$
7:          $V \leftarrow V + \{V_q\}$
8:      end for
9:      for all $v_i \in V$ do
10:         for all $v_j \in V$ do
11:             $sharing \leftarrow Data\_Shared(v_i, v_j)$
12:             if $sharing > 0$ then
13:                 $E_k \leftarrow AddEdge(v_i, v_j)$
14:                 $|E_k| \leftarrow sharing$
15:                 $E \leftarrow E + \{E_k\}$
16:             end if
17:         end for
18:     end for
19:     $G \leftarrow \{V, E\}$
    return G
20: end procedure
```

Fig. 7. Building graph structure.

After representing queries and the potential data sharing opportunities as a graph, we next cluster the vertices/queries based on the cache topology of the underlying multicore machines. An on-chip cache topology can be modeled using a *tree* where the last level on-chip cache is the root and the first level caches are the leaves. For a two-socket system with two last level caches such as the ones depicted in Fig. 2, a virtual root is used. Our clustering algorithm partitions queries starting from the root level moving towards the leaf level caches. At each level, a *k-way* partitioning takes place where *k* is equal to the number of child nodes. In other words, the number of generated partitions in each level is equal to the number of child nodes in the cache hierarchy tree. When the algorithm terminates, we have the same number of partitions as the number of domains available at the target affinity level and each query is assigned to a particular partition. A *k-way* graph partitioning problem [21] can be expressed in formal terms as follows:

For a given graph $G(V, E)$, find a set of graphs such as $P = \{G_0(V_0, E_0), \ldots, G_{k-1}(V_{k-1}, E_{k-1})\}$, where $\bigcup_0^{k-1} V_i = V$ and $\forall i, j, i \neq j \rightarrow V_i \cap V_j = \emptyset$.

A *k-way* graph partitioning problem is typically associated with a cost function. The goal of this partitioner is to minimize this cost function. One common cost function is the sum of inter-partition edge weights that span more than one partitions. In our approach, we try to group queries which are working on the same data more than the others. We achieve our goal through representing the amount of inter-query data sharing as edge weight and minimizing the cost function.

```
1: $T$: cache hierarchy tree
2: $G$: a graph consisting queries as vertices
3: $CS = \{cs_0, ..., cs_{k-1}\}$: cluster set (affinity groups)

4: procedure PARTITIONER($G, T$)
5:     $CS \leftarrow \emptyset$
6:     if $isLeaf(T)$ then
7:         $V \leftarrow Vertices(G)$
8:         for all $v \in V$ do
9:             $CS \leftarrow CS + \{v, T\}$
10:        end for
11:    else
12:        $k \leftarrow NumClusters(T)$
13:        $Partitions \leftarrow MultiLevelPartitioning(G, k)$
14:        for all $p \in Partitions$ do
15:            $t \leftarrow LevelUp(T)$
16:            $CS \leftarrow CS + Partitioner(p, t)$
17:        end for
18:    end if
       return $CS$
19: end procedure
```

Fig. 8. Graph partitioning.

For the implementation of *k-way* partitioning, we employ a well-known graph library based on the multi-level recursive bisectioning algorithm presented in [26]. In brief, a multi-level partitioning algorithm can be divided into three distinct phases. The first phase, called *coarsening*, groups the connected vertices of the graph into a bigger vertex to form a coarser graph which contains a smaller number of vertices than the original graph. Coarsening is performed iteratively at multiple levels and the graph is shrunk at each level. At each level, coarsening is done by finding a maximal matching with the help of a *heavy-edge matching* algorithm. Coarsening is finished when it reaches the smallest graph, called the top-level graph. In the second phase, a *two-way partitioning* is applied to this top-level graph. Finally, starting from the top-level, each partition is *projected* to upper levels. Coarsening, top-level partitioning, and refinement are all tunable and can be performed using different strategies including local optimization of the partition using the available topological information or similarly using an uncoarsening approach as has been done in [10].

After associating weights with vertices in our original graph along with the edge weights, we then modify our cost function in order to minimize the cut sizes of the partitions and balance the sum of the vertex weights in each partition. The multi-level recursive bisection algorithm handles weighted vertices as a balancing constraint in the top-level partitioning phase. Vertices are ordered according to their weights and assigned to partitions satisfying the balancing constraint. Next, the partitioner tries to obtain roughly equal partitions according to the sum of vertex weights while minimizing the edge-cut. While it is possible to also consider the shared data sizes within a partition while obtaining the total cost of that partition, we did not explore this as it also complicates the partitioning algorithm. Based on our preliminary analysis, we did not see a significant benefit in applying such an extension. A pseudo-code for the proposed graph partitioning approach is given as Fig. 8.

TABLE 1
The Constant Terms Used in Our ILP Formulation

Constant	Definition
M	Number of cores
T	Number of affinity domain
I	Number of queries for affinity domain
$L_{t,q}$	Load for a given affinity domain t and query q
Q	Load imbalance coefficient

These are either architecture specific or workload specific.

6.2 Load Balancing

Although an *k-way* partitioning heuristic is able to produce k nonempty partitions, it cannot guarantee *balanced query workloads*. Thus, we need to balance the loads (i.e., the average number of cycles to process queries assigned to each partition) explicitly across domain affinities. For this, we adopt a 0-1 integer linear programming based formulation to balance the query loads mapped onto domain affinities.

Table 1 gives the constant terms used in our ILP formulation. Note that, the loads given in this table are normalized using the minimum amount of processing load that can be allocated to an affinity domain. Load imbalance coefficient is used as an upper limit for the difference between the amount of work assigned to two affinity domains. This value can be obtained through experimentation, query types, data being accessed, and history of the executions. Moreover, it is also possible to devise an adaptive technique where this coefficient gets adjusted according to a function of current state and history.

For each query, we define 0-1 variables to specify the assignment of a query to an affinity domain. Specifically, we define:

- $X_{t,q,m}$: to indicate whether affinity domain t and instance q of that domain is assigned to core m.

We use a variable for each one of the possible assignments. If this 0-1 variable is 1, this indicates that the corresponding query can be assigned to core m. If this size is 0, then we conclude that this assignment does not exist.

We use another 0-1 variable to indicate (after final assignment) whether two different queries of the same affinity domain are assigned to the same core or not:

- S_{t,q_1,q_2} : indicates whether query q_1 and q_2 of affinity domain t can be assigned to the same core.

We use AL, a non 0-1 variable, to express the total assigned query load assigned to each core:

- AL_m : indicates the amount of load assigned to core m.

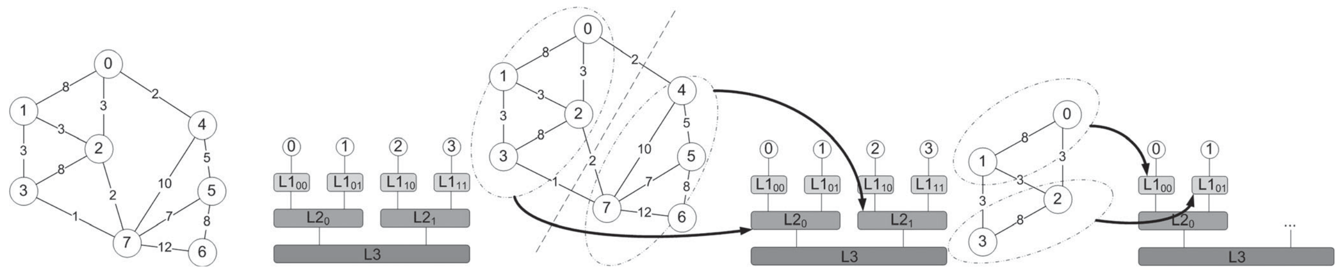
After defining the variables in our ILP formulation, now we explain the necessary constraints to be satisfied.

Each query must be assigned to a particular core, captured by the constraint:

$$\sum_{k=1}^{M} X_{t,q,k} = 1, \quad \forall t, q. \tag{6}$$

Also, two queries are said to be assigned to the same core if the following constraint holds:

$$S_{t,i_1,i_2} >= X_{t,i_1,m} + X_{t,i_2,m} - 1, \quad \forall t, i_1, i_2, m, \text{where} i_1 \neq i_2. \tag{7}$$



(a) Initial graph and target architecture. (b) First level partitioning. (c) Second level partitioning.

Fig. 9. Example application of our scheme.

If both i_1 and i_2 queries(affinity domain t) are assigned to the same core (m), then 0-1 variable S_{t,i_1,i_2} will be forced to have a 1 value.

A necessary constraint is related to the load balancing in the query mapping between affinity domains which will prevent overloading of a core-pair with running related queries. To capture this, we use variable AL_m to indicate the total assigned query load onto the core m. The estimated load of a particular query can be extracted from the associated query plan derived by the query optimizer. As explained earlier, we use the estimated cost (execution time) for each operator and generate the total estimated cost (execution time) for a certain query execution plan

$$AL_m = \sum_{i=1}^{T} \sum_{j=1}^{I} X_{i,j,m} \times L_{i,j}, \quad \forall m. \tag{8}$$

This expression essentially sums up all the assigned query loads to generate the total load of the core. This variable is then used for limiting the disparities across the loads of different cores. More specifically

$$AL_{m1} - AL_{m2} < AL_{m2} \times Q, \forall m1, m2, \text{where } AL_{m1} > AL_{m2}. \tag{9}$$

Note that, the load imbalance coefficient (Q) is given as a percentage in Table 1. Having specified the necessary constraints in our ILP formulation, we next give our objective function

$$max \sum_{i=1}^{T} \sum_{j=1}^{I} \sum_{k=1}^{I} S_{i,j,k}, \text{ where } j \neq k. \tag{10}$$

Based on the above expression, our 0-1 ILP problem can formally be defined as one of "maximizing the objective function under constraints (6), (7), (8), (9), and (10)."

After the clustering phase, we analyze the query-to-domain affinity mappings and check whether there is an overloaded domain affinity or not. *Overload* simply refers to the case when the difference between the amount of loads assigned to two domain affinities are greater than the fixed load balance threshold. To calculate the load on a domain affinity, we use the sum of query execution time estimations extracted from the corresponding query plans. When we detect an overloaded domain affinity, we try to group it with the domain affinity that has the minimum amount of load in order to exchange queries between domains. If these grouped domain affinities are not overloaded after query transfers/exchanges, then load balancing is considered to be successful and we update query-to-domain affinity mappings according to these new assignments. Otherwise, we leave the overloaded domain affinity as it is, and move to the next overloaded domain affinity to try to apply the same logic. A pseudo-code for the load balancer is provided as Fig. 10.

Note that, our approach can potentially degrade the performance of the system if load balancing is applied aggressively. This will cause higher number of context switches increasing the overheads. Moreover, certain load imbalance coefficients may lead to idle cores under-utilizing the system. This may even cause higher energy consumption.

Apart from actual query execution times, query planner and optimizer will be more sophisticated in order to analyze queries and execute them accordingly. But the overhead associated with the analysis phase is orders of magnitude less than the actual execution time. This is due to the fact that query planner and optimizer timing mostly depend on the depth of the hash-join chains, which usually is very limited. Moreover, there is no additional hardware overhead associated with our approach.

6.3 Example

Consider the query graph shown in Fig. 9a. For clarity purposes, we take only affinities into consideration and have no weights assigned to the vertices. Edges are weighted according to the amount of data sharing between queries. If two queries, such as 2 and 4, do not work on common records,

```
1: $QP = \{q_0, ..., q_n\}$: query plans
2: $K$: number of available cores/domain affinities
3: $CS = \{cs_0, ..., cs_{k-1}\}$: cluster set (affinity groups)

4: **procedure** LOADBALANCER($CS, QP, K$)
5:     $results \leftarrow runILPSolver(CS, QP, K)$
6:     $CS \leftarrow updateAssignments(CS, results)$
7:     **for all** $cs_i \in CS$ **do**
8:         **if** $isOverloaded(cs_i, K) > 0$ **then**
9:             $cs_{min} \leftarrow getMinimum(CS)$
10:            $grouped \leftarrow group(cs_i, cs_{min})$
11:            $trial \leftarrow runILPSolver(grouped, QP, K)$
12:            **if** $hasSolution(grouped, QP, K) > 0$ **then**
13:                $CS \leftarrow updateAssignments(CS, trial)$
14:            **end if**
15:        **end if**
16:    **end for**
       **return** CS
17: **end procedure**
```

Fig. 10. Load balancer.

TABLE 2
Important Features of Our Target Multicore System

Number of Cores	6 cores (per socket)
Number of Sockets	2
Socket Type	lga-1,356
Cache Parameters	L1: 32 KB, core private, 3 cycles L2: 256 KB, core private, 14 cycles L3: 10 MB per socket (shared by 6 cores), 36-44 cycles
Off-Chip Latency	~330 cycles

we have no edge between them. It is to be noted that, in this example, our target affinity level corresponds to L1 caches; consequently, we need to carry out a two-level partitioning.

We now go over our hierarchical query clustering scheme. Since the L3 cache is shared by all cores and is the root of the cache hierarchy tree, the first step is to cluster the query groups for the L2 cache. The graph and the assignment after the first level of clustering and query mapping are shown in Fig. 9b. Next, the query distribution is applied to each of the two clusters formed in the previous step. After this second and final level of clustering and load balancing, the query clusters are assigned to the target domain affinities, as illustrated in Fig. 9c.

7 Experimental Evaluation

7.1 Setup

We tested our querying scheduler using an Intel IvyBridge-EN multicore system. The important characteristics of this system are given in Table 2. To perform our experiments, we prepared two query workloads listed in Table 3. The benchmarks in these workloads are compiled from the TPC-H suite [33], which is an industry standard to simulate a decision support system. While TPC-H specifies the tables, relationships, and characteristics of the database, it does not enforce a specific DBMS. More specifically, TPC-H requires and industry-level DBMS to implement a database that consists of eight separate and individual tables. The relationships between columns of these tables are explicitly defined as well. There are 22 queries defined as part of TPC-H to stress the database in different ways. Similarly, TPC-H enforces the data populating the database to examine large volumes of data. Single-core performance statistics of these queries are given in Table 4, where distinct workload characteristics have been explored [33]. We ran these workloads on a data set size of ~1 GB with various numbers of clients. All experiments are repeated five times and the presented results represent the average values of these multiple runs. In our experimental setting, each client fires only one query from the corresponding workload. Workloads are provided to the system in batches. In a closed-queuing network, system requests a new workload after each job in the previous batch is terminated.

TABLE 3
Our Workloads (Query Mixes)

ID	Queries (TPC-H)	Description
WL-1	2,3,4,5,7,8,9,10,11,12,13,14	Join bound
WL-2	1,2,3,4,5,6,7,8,9,10,11,12	Scan - Join Mixed

TABLE 4
Performance Evaluation Parameters of the Benchmark Queries

Query	# of L2 Misses	# of L3 Misses	Query Exec. Time
Q1	705,303,888	1,512,924	61.0 sec
Q2	1,934,521,657	82,781,055	497.2 sec
Q3	76,269,478	5,203,913	11.8 sec
Q4	64,830,065	4,282,042	16.9 sec
Q5	112,732,882	9,448,113	26.3 sec
Q6	37,888,791	437,269	31.4 sec
Q7	92,209,962	4,882,086	28.4 sec
Q8	146,264,841	22,806,660	9.6 sec
Q9	236,333,814	13,392,932	24.5 sec
Q10	98,513,105	5,249,866	32.0 sec
Q11	21,355,776	571,485	59.4 sec
Q12	46,423,994	2,413,886	24.2 sec
Q13	89,780,792	1,180,228	14.5 sec
Q14	29,125,660	425,294	4.9 sec

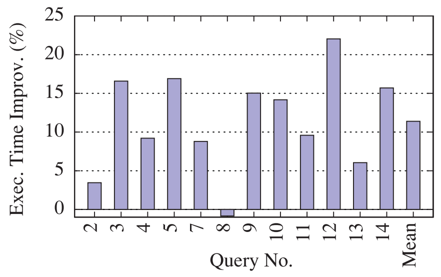
Each column gives the absolute values collected when the query is executed on a single core of our IvyBridge based system using the default Linux scheduler.

We used PostgreSQL 8.4 installed in Linux 2.6 kernel with its default configuration. PostgreSQL handles each client as a separate process, and heavily relies both on its buffer pool and the underlying operating system's file I/O cache. Therefore, we warmed up these buffers before starting to collect our results. In our experiments, the load imbalance coefficient mentioned in the given ILP formulation is set to 10 percent. A commercial tool, XpressMP [38], is used to formulate and test the ILP-based approach. Xpress-MP takes the problem as a Mosel description which is a plain text file with descriptions of binary variables, constraints, and objective function. The results are collected using perfmon2 [28] from the hardware counters. Since we use a database system running a process per query, query mappings onto affinity domains are forced through *taskset()* system call on particular processes. Results presented in this section are all *normalized* with respect to the results obtained by using the standard Linux-based scheduler on each multicore architecture. Absolute performance counter values with the Linux scheduler are given in Table 4.

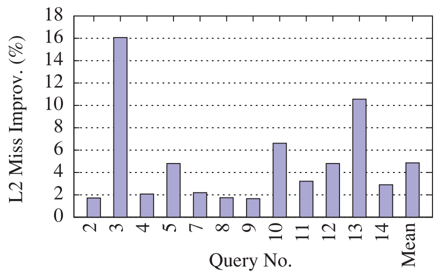
7.2 Results

Fig. 11a gives the improvements in query execution times of the WL-1 workload, brought by our approach over the default Linux scheduler. We observe that, with this workload, the average performance improvement per query is about 11.4 percent. This is due to the fact that our proposed mapping scheme reduces L2 and L3 cache misses on average by 5 and 13 percent, respectively (see Figs. 11b and 11c for improvements in the L2 and L3 miss rates). We repeated similar performance analysis experiments with the WL-2 workload as well. Recall from Table 3 that, as opposed to WL-1, this workload includes *both* join and scan bound queries. One can see that, with this workload, the execution time improvements our approach brings range between 4 and 20 percent, averaging on 10.1 percent. The coresponding L2 and L3 miss rate reductions for this workload are plotted in Figs. 12b and 12c. these improvements clearly underline the success of our strategy in exploiting the underlying cache hierarchy.

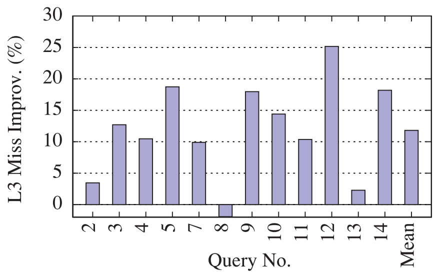
We next double the number of queries per core in each of our multicore machines. In doubling the number of queries, we replicated the original workload. The goal behind these



(a) Execution time improvement.

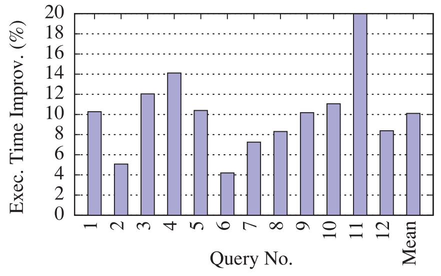


(b) L2 miss rate improvement.

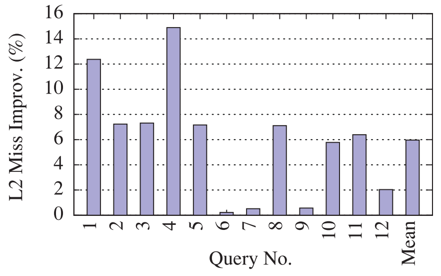


(c) L3 miss rate improvement.

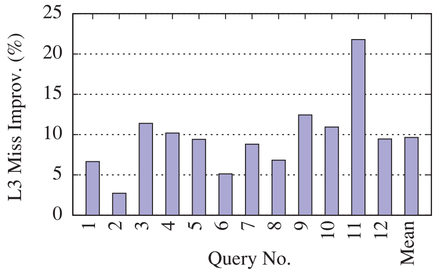
Fig. 11. WL-1 results with 12 clients.



(a) Execution time improvement.



(b) L2 miss rate improvement.



(c) L3 miss rate improvement.

Fig. 12. WL-2 results with 12 clients.

experiments is to measure the performance of our load balancing algorithm (see Algorithm 10). When we double the number of clients (i.e., we move from 12 clients to 24 clients), the corresponding average execution time improvements per query is 17.8 percent, as presented in Fig. 13 for the case of WL-1.

In order to compare the *throughput* of our approach with the default scheduler, we devised a closed-queuing network where the system requests a new batch of queries whenever all queries from the previous batch are finished. Each new batch is randomly composed of queries from WL-1 and WL-2 with different number of clients. As can be observed from Fig. 14a, when we cumulatively process 60, 84, 108 and 132 clients, the overall throughput improves 15-19 percent over the default OS scheduler.

When we look at the performance improvement values per query basis, we see that, except in very few cases, our approach improved the individual query execution time. We see some performance degradations with Query 8 of

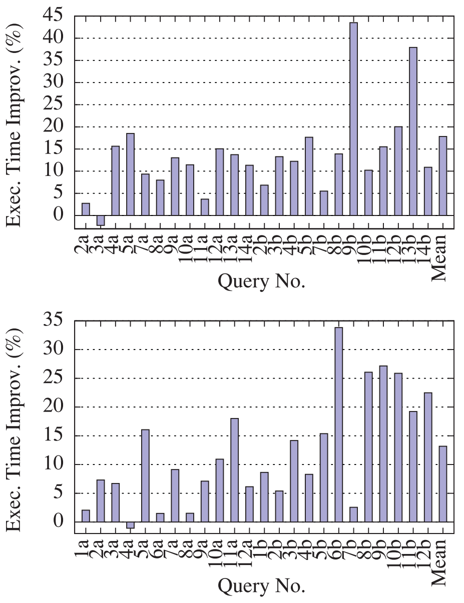
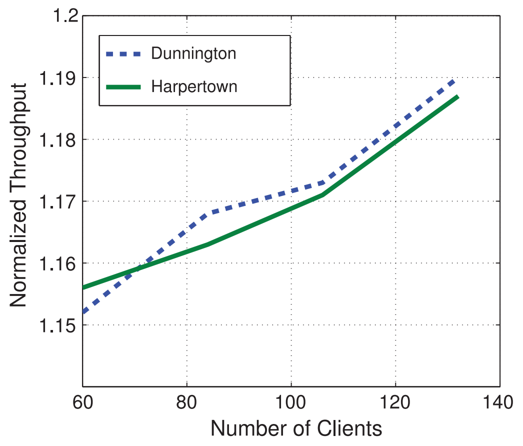
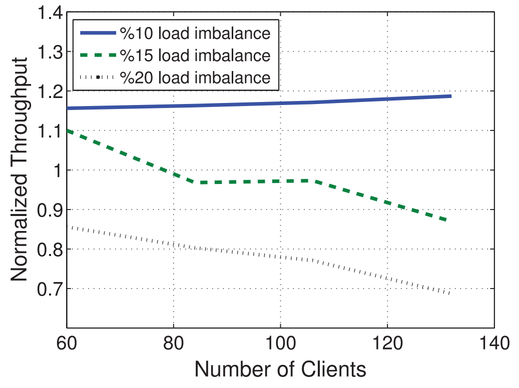


Fig. 13. WL-1 and WL-2 results with 24 clients.



(a) Throughput with different number of clients.



(b) Regression analysis on load imbalance coefficient.

Fig. 14. (a) Throughput with different number of clients. (b) Regression analysis on load imbalance coefficient.

WL-1 with 12 clients and Query 2 of the same workload with 24 clients. These results are mostly due to other processes which are spawned by the Linux kernel. They interfere with affinity domains without our control and bring unshared data into affinity domains along with additional load to be processed, causing extra context-switches, and ultimately resulting in performance degradation.

One can also see that all queries have the same data access pattern or interactions with other, co-runner queries. This is because, query plans can exhibit different characteristics at run-time. In particular, when it is not possible to build appropriate neighborhood (i.e., tandems) among queries, then mutual benefits between these queries become limited. To address this issue, instead of scheduling each query plan as a whole, it may be more beneficial to split query plans into fine-grained stages and schedule these stages individually [16].

7.3 Regression Analysis

In this experiment, we run our throughput analysis under various different load imbalance coefficients. The load imbalance coefficient is used in ILP formulation and provides an upper limit for the difference between the amount of work assigned to two affinity domains. Results are provided in Fig. 14b. We figure out that increasing load imbalance coefficient can lead to have idle cores at runtime i.e., a core connected to an affinity domain can finish its jobs and become idle while other cores are still processing. Typically, in such a case, OS-based schedulers can identify this core as idle at runtime and can assign a new task to the idle core if there is a task waiting in another core's task queue. Thus, our mapping strategy can be suppressed by dynamic OS scheduler if the load imbalance between query-to-affinity domain assignments cannot be compensated by the achieved data locality optimizations.

8 Conclusion

In this paper, we address one of the problems of *multi-query scheduling on emerging multicore architectures*. We show that singularities across on-chip cache topologies designed for different multicore architectures further complicate scheduling decisions beyond the traditional resource allocation and load balancing concerns. Eventually, how a scheduler utilizes on-chip cache topology becomes an important factor of runtime performance. In order to manage and exploit hardware design differences, we propose an architecture aware multi-query scheduling scheme. Our implementation of this scheme provides up to 25 percent improvement in individual query execution times and 15-19 percent improvement in throughput as demonstrated by our experiments on an Intel IvyBridge based multicore. Our future work includes developing intra-query, shared cache-aware parallelization strategies, and investigating interactions between query parallelization and locality optimization.

Acknowledgments

A preliminary 2-page version of this paper appears in the Proceedings of 2014 IEEE International Symposium on Workload Characterization (IISWC) [27]. This work has been done when U. Orhan, W. Ding, and P. Yedlapalli were at Pennsylvania State University.

References

[1] R. Acker, C. Roth, and R. Bayer, "Parallel query processing in databases on multicore architectures," in *Proc. 8th Int. Conf. Algorithms Archit. Parallel Process.*, 2008, pp 2–13.

[2] R. Agrawal, et al., "The Claremont report on database research," *Commun. ACM*, vol. 52, pp. 56–65, Jun. 2009.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a modern processor: Where does time go?" in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 266–277.

[4] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *Proc. VLDB Endowment*, vol. 5, no. 10, pp. 1064–1075, Jun. 2012.

[5] A. Anastasia, "Embarrassingly scalable database systems," in *Proc. Int. Conf. Data Eng.*, 2011, pp. 1–1.

[6] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proc. 41st Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2008, pp. 318–329.

[7] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in MonetDB," *Commun. ACM*, vol. 51, pp. 77–85, Dec. 2008.

[8] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," *ACM SIGARCH Comput. Archit. News*, vol. 34, pp. 264–276, May2006.

[9] S. Chen, et al., "Scheduling threads for constructive cache sharing on CMPs," in *Proc. 19th Annu. ACM Symp. Parallel Algorithms Archit.*, 2007, pp. 105–115.

[10] C. Chevalier and I. Safro, *Learning and Intelligent Optimization*, T. Stützle, Ed. Berlin, Germany: Springer, 2009, pp. 191–205.

[11] J. Cieslewicz, W. Mee, and K. A. Ross, "Cache-conscious buffering for database operators with state," in *Proc. 5th Int. Workshop Data Manage. New Hardware*, 2009, pp. 43–51.

[12] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *Proc. 33rd Int. Conf. Very large data bases*, 2007, pp. 339–350.

[13] J. Duffy and E. Essey, "Running queries on multi-core processors," 2007. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc163329.aspx

[14] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2005, Art. no. 26.

[15] N. Hardavellas, I. Pandis, R. Johnson, N. G. Mancheril, A. Ailamaki, and B. Falsafi, "Database servers on chip multiprocessors: Limitations and opportunities," in *Proc. Conf. Innovative Data Syst. Res.*, 2007, pp. 79–87.

[16] S. Harizopoulos and A. Anastassia, "StagedDB: Designing database servers for modern hardware," in *Proc. IEEE Data Eng. Bulletin*, vol. 28, 2005, pp. 11–16.

[17] Intel-Dunnington, 2010. [Online]. Available: http://ark.intel.com/Product.aspx?id=36941

[18] Intel-Harpertown, 2010. [Online]. Available: http://ark.intel.com/Product.aspx?id=33085

[19] Intel-Nehalem, 2010, [Online]. Available: http://ark.intel.com/Product.aspx?spec=slbf5

[20] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. J. Irwin, and Y. Zhnag, "Cache topology aware computation mapping for multicores," in *Proc. 31st ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2010, pp. 74–85.

[21] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Proc. ACM/IEEE Conf. Supercomputing*, 1998, pp. 1–13.

[22] G. Kun, S. Harizopoulos, I. Pandis, V. Shkapenyuk, and A. Ailamaki, "Simultaneous pipelining in QPipe: Exploiting work sharing opportunities across queries," in *Proc. 22nd Int. Conf. Data Eng.*, 2006, Art. no. 162.

[23] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, "MCC-DB: Minimizing cache conflicts in multi-core processors for databases," *Proc. VLDB Endowment*, vol. 2, pp. 373–384, Aug. 2009.

[24] S. Liknes, "Database operations on multi-core processors," M.S. thesis, Dept. Comput. Inf. Sci., Norwegian Univ. Sci. Technol., Trondheim, Norway, 2013.

[25] M. Mehta, V. Soloviev, and D. J. DeWitt, "Batch scheduling in parallel database systems," in *Proc. 9th Int. Conf. Data Eng.*, 1993, pp. 400–410.

[26] METIS, 2010. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/views/metis

[27] U. Orhan, W. Ding, P. Yedlapalli, M. Kandemir, and O. Ozturk, "A cache topology-aware multi-query scheduler for multicore architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2014, pp. 86–87.

[28] Perfmon2, 2010. [Online]. Available: http://perfmon2.sourceforge.net/

[29] P. Petrides, A. Diavastos, C. Christofi, and P. Trancoso, "Scalability and efficiency of database queries on future many-core systems," in *Proc. 21st Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2013, pp. 24–28.

[30] PostgreSQL, 2010. [Online]. Available: http://www.postgresql.org/

[31] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman, "Main-memory scan sharing for multi-core CPUs," *Proc. VLDB Endowment*, vol. 1, pp. 610–621, Aug. 2008.

[32] K. A. Ross, "Optimizing read convoys in main-memory query processing," in *Proc. 6th Int. Workshop Data Manage. New Hardware*, 2010, pp. 27–33.

[33] M. Shao, A. Ailamaki, and B. Falsafi, "DBmbench: Fast and accurate database workload representation on modern microarchitecture," in *Proc. Conf. Centre Adv. Stud. Collaborative Res.*, 2005, pp. 254–267.

[34] A. Shatdal, C. Kant, and J. F. Naughton, "Cache conscious algorithms for relational query processing," in *Proc. 20th Int. Conf. Very Large Data Bases*, 1994, pp. 510–521.

[35] M. Stonebraker, et al., "C-store: A column-oriented DBMS," in *Proc. 31st Int. Conf. Very large data bases*, 2005, pp. 553–564.

[36] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 47–58.

[37] TPC-H, 2010. [Online]. Available: http://www.tpc.org/tpch/

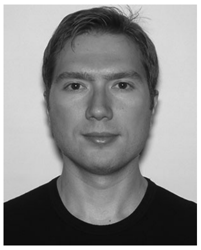
[38] XPress-MP, 2015. [Online]. Available: http://www.fico.com/en/products/fico-xpress-optimization-suite

[39] D. Yadan, J. Ning, X. Wei, C. Luo, and C. Hongsheng, "Hash join optimization based on shared cache chip multi-processor," in *Proc. 14th Int. Conf. Database Syst. Adv. Appl.*, 2009, pp. 293–307.

[40] P. S. Yu, D. W. Cornell, D. M. Dias, and B. R. Iyer, "On affinity based routing in multi-system data sharing," in *Proc. 12th Int. Conf. Very Large Data Bases*, 1986, pp. 249–256.

[41] M. Zukowski, S. Héman, N. Nes, and P. Boncz, "Cooperative scans: Dynamic bandwidth sharing in a DBMS," in *Proc. 33rd Int. Conf. Very large data bases*, 2007, pp. 723–734.

Ozcan Ozturk is an associate professor in the Department of Computer Engineering, Bilkent University. His research interests include many-core architectures, parallel computing, and computer architecture. Prior to joining Bilkent, he worked in Cellular and Handheld Group, Intel and Marvell. He also held positions in NEC Labs and Arizona State University. His research has been recognized by Fulbright, Turk Telekom, IBM, Intel, HiPEAC, Tubitak, and European Commission.



Umut Orhan received the BS and MS degrees in computer engineering from Middle East Technical University, in 2006 and 2009, respectively. He received another MS degree in computer science and engineering from Pennsylvania State University, in 2011. Right now, he is working with Amazon Seller Services. He is designing and developing scalable web services to provide Amazon customers with the largest, highest quality and most up to date retail selection in the world. His current research interests include computer architectures, optimizing compilers, databases, and distributed systems.

Wei Ding received the PhD degree in computer science from Pennsylvania State University, in 2014. He is currently a senior software engineer with Qualcomm Innovation Center Inc., California. His research interests include compilation optimization, data locality optimization, and high performance computing on multi-cores and mobile devices.

Praveen Yedlapalli received the PhD degree from Pennsylvania State University under the guidance of Dr. Mahmut Kandemir. His PhD thesis titled "A Study of Parallelism-Locality Trade-offs Across Memory Heirarchy", does an in-depth analysis of processor-memory bottlenecks in modern CMPs. He is working in VMware specifically in the CPU and Memory performance team for ESX hypervisor. His research interests include computer architecture, operating systems, and distributed systems.

Mahmut Taylan Kandemir is a full professor in the Computer Science and Engineering Department, Pennsylvania State University. He is the author of more than 80 journal publications and more than 350 conference/workshop papers in optimizing compilers, manycore architectures, and storage systems. He served in the program committees of more than 50 conferences and workshops. His research is/was funded by US National Science Foundation, DOE, DARPA, SRC, Intel, and Microsoft. He received the US National Science Foundation Career Award and the Penn State Engineering Society Outstanding Research Award. He is a fellow of the IEEE.