A Cache Topology-Aware Multi-Query Scheduler for Multicore Architectures

Umut Orhan [\$], Wei Ding [#], Praveen Yedlapalli [#], Mahmut Kandemir [#], Ozcan Ozturk [*]
[\$]*Amazon Inc.*, [#]*The Pennsylvania State University, USA*, [*]*Bilkent University, Ankara, Turkey.*
{uxo101, wzd109, praveen, kandemir}@cse.psu.edu, ozturk@cs.bilkent.edu.tr

I. Introduction

Growing performance gap between processors and main memory has made it worthwhile to consider off-chip data accesses in multi-query processing [2], [1], [3]. 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.

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 improve the overall throughput of the system. We address two main concerns in optimizing multi-query scheduling: *affinity* and *load balancing*. 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 affinity domains (in our case, a set of on-chip caches depending on the underlying cache topology). These assignments can improve data locality on shared caches and lead to reduced number of off-chip data accesses.

Towards that, we identify common data retrieval operations in multi-query workloads and build *affinity relations* as undirected weighted graphs between queries that represent possible data sharing at runtime. Edge weights are calculated from the query plan estimations provided by the query optimizer. Using this graph, we invoke a *hierarchical clustering algorithm* to generate *query-to-affinity domain mappings*. An *affinity domain* in this context refers to a particular cache structure bounded by a specific level of the cache hierarchy. According to the generated mappings, each query is executed only on the cores that are connected to the corresponding affinity domain. In Figure 1, we give the high-level view of our automated approach to cache topology aware query scheduling.

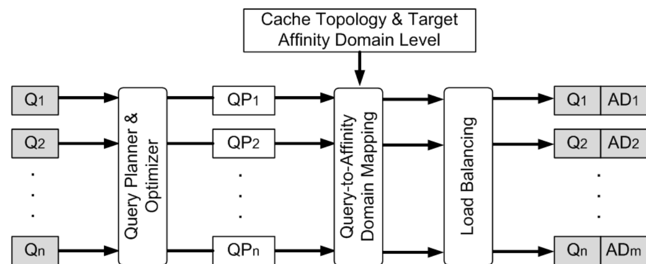


Fig. 1: High level sketch of our cache topology-aware query scheduling approach.

II. Proposed Scheduler

Let us consider three different mappings of four queries(Q) from TPC-H [4] on a dual-socket Intel IvyBridge-EN based architecture. The first mapping maps all queries to one of the sockets; the second mapping maps Q1 and 2 to the first socket and Q3 and 4 to the second socket; and the third mapping maps Q1 and 3 to the first socket and Q2 and 4 to the second socket. The query execution times (normalized to query executed in isolated fashion) of mappings are plotted in Figure 2.

We can see that, although each mapping uses the same number of cores, the execution time of a given query exhibits significant variances depending on the mapping used, indicating that cache performance plays a critical role. Further, when all queries are executed in the same socket, we see that the performance of each query suffers to varying degrees. This is expected due to contention in the last level cache. However, when we move to the second mapping, we see that the performances of Q1 and Q2 improve over isolated executions. This is because of the data (table) sharing between these two queries. When we look at the results with the third mapping, we see that they are very similar to those of the first mapping. Overall, these results show that careful mapping of queries to cores can improve query execution times.

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 affinity domain. It tries to evenly distribute the queries among available cores while maximizing possible data sharings through shared caches.

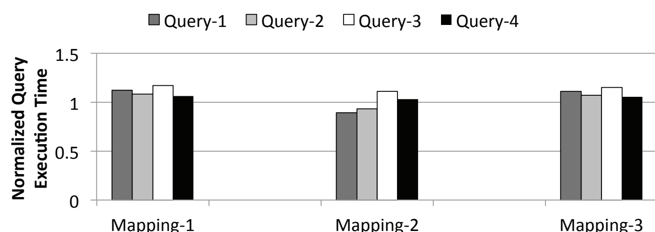


Fig. 2: Query execution times with different mappings.

[\$] This work has been done when the author was at Penn State University

Estimating the Amount of Shared Data between Two Queries The *reuse distance* of shared data is an important concern because if the distance between two scan operations which read tuples of the same relation is significantly large, then the 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 order to address this, we enhance our data sharing model by considering the selectivity of each scan operation, as two scan operations having similar selectivities are more likely to share tuples brought into a cache. This information is extracted by parsing the query execution plan where scan operations are associated with estimated costs and the number of the resulting tuples.

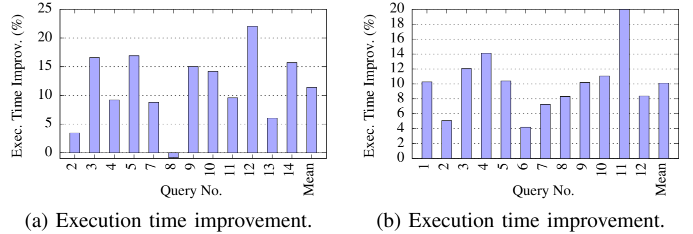
Estimating the Working Memory Sizes We estimate the peak working memory size of a query as:

$$\begin{aligned} H &= max(\bigcup |h|), \\ P &= |k_i| + |a|, \\ WMS &= max(H, P), \end{aligned} \quad (1)$$

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.

Query-to-Affinity Domain Mapping: Our proposed scheduling scheme consists of three components, namely *graph builder, graph partitioner*, and *load balancer*. 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. To avoid cache thrashing effects we consider *vertex weights* representing the working memory sizes of queries. 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. 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. 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. The goal of this partitioner is to minimize the sum of inter-partition edge weights that span more than one partitions. Next, we minimize the cutsizes of the partitions and balance the sum of the vertex weights in each partition. Finally, the partitioner tries to obtain roughly equal partitions according to the sum of vertex weights while minimizing the edge-cut.

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 affinity domains. For this, we adopt a 0-1 ILP (integer linear programming) based formulation to balance the query loads mapped onto affinity domains.



(a) Execution time improvement. (b) Execution time improvement.

Fig. 3: WL-1 and WL-2 results with 12 clients.

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

Results We tested our querying scheduler using an Intel IvyBridge-EN multicore system with PostgreSQL. To perform our experiments, we used two query workloads.

Figure 3a gives the improvements in query execution times of the WL-1 and WL-2 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%. This is due to the fact that our proposed mapping scheme reduces L2 and L3 cache misses. In WL-1, L2 and L3 misses were reduced on average by 5% and 13%, respectively.

We repeated similar performance analysis experiments with the WL-2 workload as well. These improvements clearly underline the success of our strategy in exploiting the underlying cache hierarchy.

III. Conclusions

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. In order to manage and exploit hardware design differences, we propose and show benefits with an architecture aware multi-query scheduling scheme.

References

[1] A. Ailamaki *et al.*, “DBMSs on a modern processor: Where does time go?” in *Proceedings of the VLDB ’99*, 1999.

[2] A. Anastasia, “Embarrassingly scalable database systems,” in *Proceedings of the ICDE ’11*, 2011.

[3] N. Hardavellas *et al.*, “Database servers on chip multiprocessors: Limitations and opportunities,” in *CIDR*, 2007.

[4] TPC-H. (2010) http://www.tpc.org/tpch/. Available: http://www.tpc.org/tpch/