DBSJ Journal, Vol.13, No.1
March 2015

Query Partitioning for Mixed Workloads in Multicore Environments

Fang XI[1] Takeshi MISHIMA[2]
Haruo YOKOTA[3]

The current generation of computer hardware has brought several new challenges for the underlying software. The number of cores on a chip has grown exponentially, enabling an ever-increasing number of processes to execute in parallel. The efficient utilization of the full range of concurrent processing capabilities offered by such a multicore platform is critical to achieving good system performance. As the number of cores on a chip increases, the increasing processor–memory gap is the bottleneck for most data-intensive applications. We therefore propose a cache-efficient CARIC-DA framework for arranging the execution of concurrent database queries on multicore platforms. This achieves improved database management system (DBMS) performance by improving cache utilization for concurrent queries. Our middleware optimizes the performance of the private-cache levels by providing query-needs-aware dispatching for concurrent online transaction-processing queries to run on different processor cores. By considering both the operating system and the DBMS application, our proposal achieves higher cache utilization for various cache levels. In this paper, we demonstrate how the middleware of CARIC-DA manages a mixed workload, where complex queries with join operations cannot share data with other queries in caches. We describe strategies that enable the middleware to partition complex queries and dispatch concurrent queries to different processor cores. The performance of the extended CARIC-DA for the TPC-W benchmark is evaluated on modern Intel and AMD multicore platforms.

1. Introduction

Over the years, great attention has been paid to providing fast and timely data access in database management systems (DBMSs) [1], [2]. However, the emergence of multicore platforms is now challenging DBMSs in several ways. Microprocessor manufacturers have hit a frequency-scaling "wall", whereby a single CPU cannot easily be designed to operate faster. They are therefore seeking improved performance via an alternative route: namely, the development of dual-core and multicore processors. A modern multicore platform increases the overall speed for those programs amenable to parallel computing by integrating multiple processing cores in a single chip to form chip-level multiprocessors. Further parallelization can be achieved by putting multiple multicore CPUs into a single server. Traditional DBMSs are based on decades-old designs that were intended to run on uniprocessors. Such systems benefit directly from improvements in the underlying uniprocessors, where higher processor frequency means higher throughput. Moreover, these systems achieved improved performance through I/O optimization. However, these DBMSs are far from realizing their potential performance when multicore platforms are used. This is because the advance in processing ability by multicore processors has far outpaced improvements in memory latency, leading to processors wasting much time waiting for required data items, particularly for data-intensive applications such as DBMSs. Therefore, the efficient use of the new hardware resources has become a hot topic for DBMS researchers.

Existing work evaluating the performance of database applications on modern multicore platforms indicates that the processor–memory gap is becoming a bottleneck, with the CPU wasting much time waiting for desired data to be loaded from the main memory for a variety of database applications [3]. With ever more cores integrated into a single chip, this processor–memory gap will further slow down the whole system. Therefore, the processor caches, which are smaller, faster memory subsystems used to store copies of data from frequently used main-memory locations, will play an important role in overcoming this "memory wall". By making cache-conscious proposals to change the data structures and algorithms into cache-friendly patterns, the cache performance can be improved for some database queries [4]. However, this research focuses on the problem only for single-process execution models. Employing data sharing for concurrent queries, rather than optimizing each query independently, remains a challenge for optimizing DBMS execution on multicore platforms. In fact, on multicore platforms, database systems are facing a different environment, where modern multicore processors are providing more powerful parallel-computing capabilities than traditional uniprocessors have been able to offer. An increasing number of concurrent database processes share resources at different cache levels between the processor cache and main memory. Any inefficient resource sharing will result in cache conflicts.

The MCCDB proposal [5] analyzed the problem of data sharing between different queries in the last-level cache (LLC), pointing out that LLC data sharing is a double-edged sword. Some frequently used data could be evicted from the LLC by a one-time-accessed big-data structure, resulting in poor data sharing in the LLC. This proposal combined the cache-coloring function of the operating system (OS) with an existing DBMS to avoid data sharing between different database queries in the LLC. In another proposal, STEPS [6] optimized the cache performance for concurrent queries by reducing cache misses in the instruction caches. STEPS breaks each

[1] Department of Computer Science, Tokyo Institute of Technology.
xifang@de.cs.titech.ac.jp

[2] Software Innovation Center, NTT Japan.
mishima.takeshi@lab.ntt.co.jp

[3] Department of Computer Science, Tokyo Institute of Technology.
yokota@cs.titech.ac.jp

request to be executed into stages and processes a group of subrequests at each stage, thereby exploiting work commonality.

However, as ever more cores are being integrated into a single socket, the microprocessor manufacturers have also recognized the data-starving problem for modern multicore platforms and are allocating a much-increased on-chip cache space to address this performance gap. For modern multicore processors, in addition to an LLC for data sharing between all concurrent processes, it is usual to provide at least two levels of cache for the private use of each processor core. With caches becoming larger and more complex, LLC access latency has increased greatly during recent decades. Therefore, it is increasingly important to bring data closer to the execution unit and optimize data sharing at the higher cache levels. On the other hand, the software of an existing DBMS is usually very large and complex, and any modification would be a time-consuming challenge. Therefore, a solution to improving cache sharing for concurrent database processes that does not involve changes to existing software would be highly attractive for DBMSs.

Core affinity with a range index for cache-conscious data access (CARIC-DA) was the first proposal to address the cache problem by properly arranging the execution of concurrent database queries for different processor cores in DBMSs on multicore platforms [7], [8]. Here, we analyzed the possibility of improving the cache-hit rate for data access in the private-cache levels by involving an effective collaboration between the DBMS and the OS. CARIC-DA assigns those queries that access the data in the same data range to run on the same processor core, while makes sure that queries accessing different data are executed on different processor cores. Using this query-aware concurrent-query dispatching, the private cache of each processor core has to cache only a small subdataset of the whole database, thereby achieving a high cache-hit rate. An advantage of CARIC-DA is that it is implemented as middleware, with the existing DBMS and OS being used without modification. This makes our proposal more practical than other approaches to the effective utilization of multicore environments that would need to modify the existing DBMS or OS.

Performance evaluation of CARIC-DA using TPC-C [7], [9], which is a typical online transaction-processing (OLTP) benchmark, proved the efficiency of our proposal for optimizing OLTP applications. A typical OLTP application comprises a large number of concurrent, short-lived transactions, each accessing a small fraction (ones or tens of records) of a large dataset. Furthermore, the smaller cache footprints of these transactions make the data sharing between sequences of transactions possible in the private cache levels that are relatively small. When we come to other applications with mixed workload, such as those modeled in the TPC-W benchmark [10], scheduling complex queries with join operations becomes a new problem. Complex queries that have join and aggregation operations on large amounts of data will evict the frequently accessed data for short queries from the private-cache levels. Therefore, in this paper, we propose an extended version of CARIC-DA that can manage mixed workloads. We demonstrate join-partitioning functions and scheduling for concurrent queries for mixed workloads with the TPC-W benchmark. By partitioning the complex queries into several smaller subqueries, we can achieve better data sharing for all concurrent queries. However, we also observed that, because queries with join and aggregation in mixed workloads are less complex than those in online analytical processing (OLAP) applications, the join partitioning is not costless and improper partitioning might well reduce system performance.

The remainder of this paper is organized as follows. Section 2 reviews our CARIC-DA proposal and Section 3 describes the join-partitioning extension for CARIC-DA. Section 4 gives the details of the benchmark and multicore platform used for the experiments. Section 5 discusses the results of the experiments. Section 6 summarizes the paper.

2. The CARIC-DA System

On a multicore platform, the DBMS has to run concurrently a number of queries, aiming to utilize the parallel-computing ability of several processor cores. These queries are dispatched to the processor cores by the OS. The OS has no information about what data the query is accessing. Furthermore, it has no information about the whole database of the application. Therefore, the OS may assign queries that access different datasets to run together on the same processor core. We have observed this problem and investigated whether we can provide a cache-efficient concurrent-query dispatching solution for DBMSs on multicore platforms by combining information about the whole database and the data needs of different queries.

2.1 Framework of CARIC-DA

We observed that different query-dispatching strategies result in different private-cache utilizations, even though they may share the LLC. Private caches are the cache levels dedicated to each processor core, and each core-related private cache is only shared by processes allocated to co-run on the specific processor core. We propose to dispatch concurrent queries according to the specific data needs of different queries, and co-run the queries that access the same data on the same processor core. For example, suppose there are three queries, Q1, Q2, and Q3. Q1 and Q2 access lines 1–40 of a table, and Q3 accesses lines 100–150 of the table. Without any data-needs information about the concurrent queries, the OS may schedule Q1 and Q3 to co-run on the same processor core (say, Core 1), while dispatching Q2 to another core (Core 2). With this dispatching strategy, the private cache of Core 1 has to cache both lines 1–40 and lines 100–150. However, if we used data-needs-based query dispatching, different queries can share data at the private-cache levels. We would co-run Q1 and Q2 on the same processor core (Core 1) and dispatch Q3 to another processor core (Core 2). Q1 and Q2 could then share the cache data and the cache-hit rate would be improved. Furthermore, dispatching the queries to different cores based on the data needs of different queries can ensure

that a private cache only accesses a specific subset of the whole database. Therefore, there is a higher probability that intended data will give cache hits at the private-cache levels.

CARIC-DA partitions the whole database between different processor cores, and ensures that those queries that access data in a specific partition will be executed by a specific processor core. In a naïve database system, the private cache of each processor core might access data from anywhere in the whole database. However, in the proposed system, by restricting the data access by each core's private cache to a specific subset of the data in the whole database, the probability of cache hits is improved.

In the naïve database system, the OS manages the query dispatching between processor cores. Changing the query-dispatching strategy would usually involve changing the functions in the existing OS. However, we take a more practical approach. We do not dispatch the database processes to different processor cores directly. Instead, we bind the database processes to different processor cores by setting a "core affinity" for each database process. We then introduce processes that dispatch queries to be processed by different database processes according to the predefined database-partition information. Even though our proposal involves extra time for performing the query dispatching, our proposal can be applied easily to existing systems without the time-consuming and challenging work of modifying the OS or the DBMS.

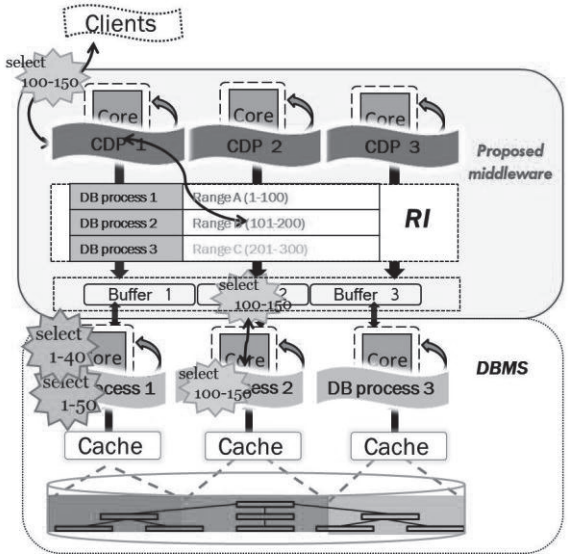

Figure 1: The CARIC-DA system deployed on a multicore platform

2.2 Architecture of CARIC-DA

The architecture of the CARIC-DA system deployed on a multicore platform is shown in Figure 1. We extend an existing DBMS in terms of a middleware level between the DBMS and the clients. CARIC-DA offers a single system image to the clients, and the clients do not need to know about the data-partitioning information in the database system. For the client, the only change is that of communicating with the middleware instead of connecting directly to the database engine. The database processes also communicate with the middleware, receiving and executing the queries in their dedicated query buffers.

In the CARIC-DA system, we partition the whole database logically into several subdatasets. We map the different subdatasets to different database processes. The data structure of the range index (RI) stores the subdataset and the database-process mapping information, as shown in Figure 2. In this example, a specific table is partitioned into three date ranges, and the three ranges are mapped to three different database processes.

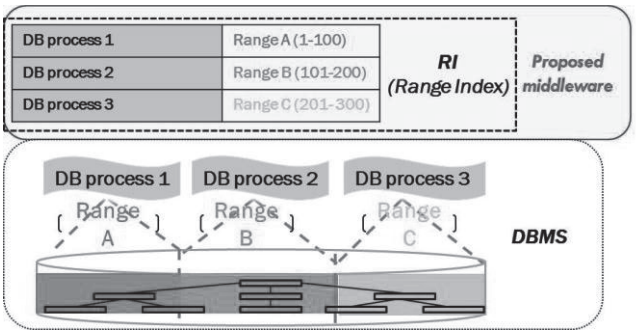

Figure 2: Logical partitioning of the database in CARIC-DA

A principal function of the middleware is to dispatch the queries to be executed by different database processes based on the RI and we introduce processes to do this work (the "CDPs" in Figure 1). These middleware processes receive queries from the client and parse each query to obtain information about the query's intended data. By referring to the RI, the target database process is determined primarily according to the query's intended data. The query will then be put into the buffer of the target database process. For example, as shown in Figure 1, a query that accesses a table between lines 100 and 150 will be scheduled for execution by DB process 2. Meanwhile, any query that accesses data outside this range will be dispatched to other DB processes. With this middleware function of intended-data-based query dispatching, a specified DB process will execute those queries that access data in the specified subdataset.

We use the function of CPU affinity [11] provided by the OS to ensure a specified DB process runs only a specified processor core. With this assignment, queries will be dispatched to run on different processor cores according to the data the query will need. The private cache of a specific processor core will then access only a specified subdataset of the whole database. The CPU-affinity function provided by a variety of OSs from Linux to Windows is used to bind one or more processes to one or more processor cores. On most systems, Linux included, the interface for setting the CPU affinity uses a "bitmask". A bitmask is a sequence of n bits, where each bit indicates a logical processor core, and setting a specific bit to 1 means the process can be scheduled to run on the corresponding processor core. We restrict a database process to run on a specific core by setting the specific

core-related bit to 1 and the other bits to 0 in the bitmask.

3. Partitioning the Join Queries

The smaller cache footprints of the OLTP transactions make the data sharing between sequences of transactions possible in the private cache levels that are relatively small. However, complex queries with join and aggregation operations have to access a large dataset during their execution. It is difficult for these complex queries to share data with other queries (simple or complex) in the private caches. The extensive runtime data of complex queries are likely to occupy the entire private cache of the processor core and evict the shareable data. Therefore, improving the cache performance for mixed workloads is a big challenge. We propose to partition each complex join query into several smaller subqueries, as shown in Figure 3. By reducing the runtime data size, we may restore data sharing in the private caches. In the following subsections, we describe this query partitioning and the dispatching of queries for a mixed workload in the middleware of CARIC-DA.

3.1 Middleware-based Query Partitioning

In the naïve approach, several complex queries may run concurrently on the different processor cores, with runtime data structures such as the index used in a nested loop join being beyond the size of a private cache. Therefore, it would be difficult to share data between different join queries, particularly for join operations that access the same data. Therefore, in the proposed approach, we run only one join query at a time on the multicore system. We partition this join query into several subqueries and then run the various subqueries on different processor cores concurrently. The smaller runtime data structures that result, such as the index data used by each subquery, has a greater probability of fitting the cache size, and the possibility of data sharing between different join queries becomes more likely.

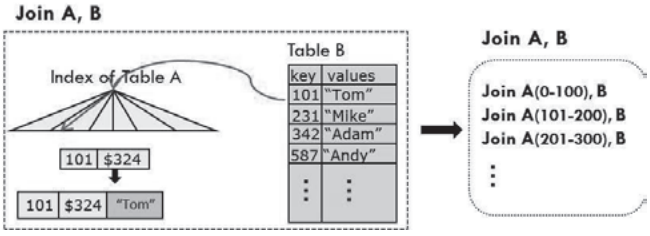

Figure 3: Partitioning the join queries

We partition each join query by rewriting the join query at the middleware level and adding some selection criteria to a specific table in each subquery. For example, in the TPC-W benchmark, the "Best Seller" transaction is a typical complex query with the join operation involving three tables: "item table", "author table", and "order line table". We can partition the Best Seller query into several Sub-BestSeller queries using the key of "item id" by adding the selection criteria for item id over a specific data range to the Best Seller query, as illustrated in Figure 4.

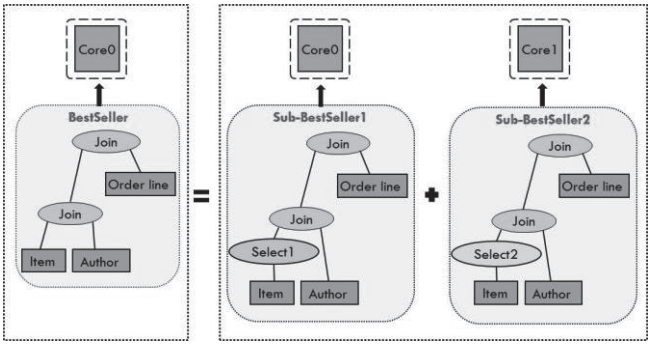

Figure 4: Partitioning the Best Seller transaction for the TPC-W benchmark

The complex queries in the mixed workload, such as those modeled in the TPC-W benchmark, are different from the transactions in typical OLAP applications. Even though there are join operations in both, the queries in OLAP applications are much more complex and involve the processing of large amounts of data in fact tables for a data warehouse. However, the join queries in the TPC-W are small by comparison and use the index-based nest-loop join method instead of the hash join. Therefore, for TPC-W, there is a greater possibility of sharing cached data between the join queries.

3.2 Query Dispatching for Mixed Workloads

The middleware maintains the query pattern of the complex queries that are to be partitioned into subqueries for a specific workload, and partitions complex queries automatically into several subqueries. To optimize a specific application with mixed workloads, we should first target the complex queries for partitioning through an initial pre-analysis of the whole application. A target query would be a complex query that involves join operations requiring long response times in comparison with other queries. In addition, queries with a high frequency of occurrence in the whole application should be targeted, because optimizing a query with a low frequency of occurrence will have little effect on the overall performance.

The middleware processes parse each query received from the clients and partition the target complex queries into several subqueries. Algorithm 1 describes how the middleware partitions and dispatches a complex query. The middleware process rewrites each query into N_{sub} subqueries, placing them in different middleware buffers. The database processes will be bound to different processor cores and obtain queries from different buffers in the middleware, as shown in Figure 1. Therefore, the different subqueries dispatched to different buffers will be executed on different processor cores by different database processes. After dispatching the N_{sub} subqueries for execution by the different processor cores, the middleware process awaits N_{sub} answers for the N_{sub} subqueries. Finally, the middleware process merges the N_{sub} answers as the final answer for the original query and returns the final answer to the client.

Algorithm 1: Managing the partitioning of a complex query in each middleware process

```
Nsub defines the number of subqueries for each query
Ans defines the answer to the query

Receive a query from the client ;
Parse the query ;
If (the query needs to be partitioned)
{
    For i in [1, Nsub]
    {
        Rewrite the query into subquery_i ;
        Put the subquery in a specific buffer ;
        (the subquery will be executed on a specific core)
    }
    For i in [1, Nsub]
    {
        Get information that the answer to subquery_i is ready ;
        Get the answer to subquery_i ;
    }
    Merge Nsub answers for the Nsub subquery as Ans ;
}
else
{
    Put the query to a specific buffer ;
    Get information that the answer to the query is ready ;
    Get the answer to the query as Ans ;
}
Return Ans to the client ;
```

The number of subqueries produced by partitioning a query has a great impact on performance. It is not a good solution to partition a single query into as many subqueries as possible because the execution time of each subquery does not reduce linearly as the number of subqueries increases.

The middleware dispatches complex queries and simple queries to different processor cores. It dispatches simple queries according to the data needs of the different queries to different processor cores based on a logical database partition, as shown in previous work on CARIC-DA for OLTP applications [7], [8]. For a complex query, we partition the query into a predefined number of subqueries and dispatch the subqueries to run concurrently on different cores. With the query-dispatching strategy for simple queries having been analyzed in previous CARIC-DA work [7], [8], we now present a detailed analysis of a dispatching strategy for complex queries. The number of cores to allocate to complex queries should be decided for specific applications according to the work intensity of its complex queries. We use the TPC-W benchmark as an example application to demonstrate the partitioning and dispatching of join queries for mixed workloads.

TPC-W is a transactional Web e-Commerce benchmark. The TPC-W database contains eight tables. Two tables contain information about books on 24 different topics and the book-related author information. Two tables contain ordering information and detailed information about each item in the order, which is the order line table. Other tables store customer-related information such as customer addresses and credit-card information. This benchmark defines a complete Web-based bookstore for searching, browsing, and ordering books. The most complex query, which has a very high frequency of occurrence, collects detailed information about best-selling books in the most-recent 3333 orders on a specific topic (the Best Seller transaction) and it has a join operation over three tables. Therefore, for the TPC-W benchmark, we would seek to optimize the Best Seller query.

There are 40 physical processor cores on our Intel multisocket multicore platform. However, using hyper-threading, the 40 cores will be seen as 80 virtual CPUs (vCPUs) by the OS. Hyper-threading [12] has been available in Intel processors for many years. By enabling the hyper-threading function on the Intel multicore processor, each core will be taken to be two vCPUs by the OS, each sharing the same private caches. We set up two different query-dispatching strategies on the Intel platform to evaluate both the hyper-threading effects and the subquery numbers for each join query. We allocated 20 vCPUs to processing the Best Seller transaction, using two dispatching strategies and separately partitioning each join query into 10 subqueries, as shown in Figure 5, and five subqueries from Sub-BestSeller1 to Sub-BestSeller5, as shown in Figure 6.

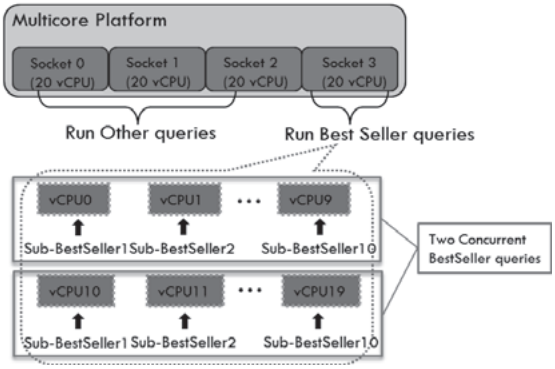

Figure 5: Query dispatching in CARIC-DA-1

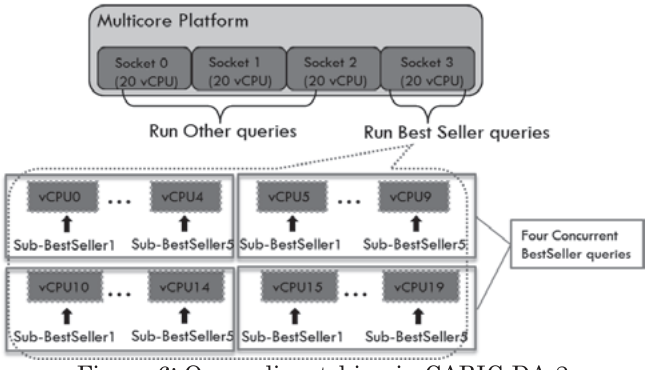

Figure 6: Query dispatching in CARIC-DA-2

For the CARIC-DA-1 strategy, we partitioned each Best Seller query into 10 subqueries and dispatched the 10 subqueries from Sub-BestSeller1 to Sub-BestSeller10 to the 10 vCPUs from vCPU0 to vCPU9, as shown in Figure 5. Note that vCPU0 to vCPU9 are associated with five physical processor cores. Therefore, on the 20 vCPUs dedicated to processing the Best Seller transaction, we can concurrently run two Best Seller queries (20

Sub-BestSeller tasks) in CARIC-DA-1.

For the CARIC-DA-2 strategy, we partitioned each Best Seller query into five subqueries from Sub-BestSeller1 to Sub-BestSeller5 and dispatched the five subqueries to five vCPUs, as shown in Figure 6. Therefore, there are four Best Seller queries running concurrently in CARIC-DA-2. However, the 10 vCPUs from vCPU0 to vCPU9 are associated with five physical processor cores. That is, vCPU0 and vCPU5 belong to the same physical processor core. Therefore, the two Sub-BestSeller1 tasks dedicated to vCPU0 and the vCPU5 from two different Best Seller queries share the same private caches for the same physical processor core. That is, there is private-cache sharing between two different Best Seller queries in CARIC-DA-2.

Because there is no hyper-threading on the 48-core AMD platform, we only considered the subquery numbers for each join query. For the CARIC-DA-1 system, we partitioned each Best Seller query into six subqueries. For the CARIC-DA-2 system, we partitioned each join query into three subqueries.

4. Experimental Setup

The functions of CARIC-DA are implemented in the C language as middleware between PostgreSQL [13] and the Linux OS. We compare the CARIC-DA-PostgreSQL-based DBMS against a PostgreSQL baseline system to evaluate the efficiency of our proposed framework. In this section, we will introduce the multicore platforms and TPC-W benchmark used in our experiments.

The hardware of the database servers is shown in Table 1. There are 80 hardware threads on the Intel platform, and 48 hardware threads on the AMD platform. The client environment comprises four machines, with each machine containing an Intel Xeon E5620 CPU and 24 GB memory.

We used the TPC-W benchmark in the experiments and evaluated the workload mixes for the "Browsing mix" (5% updates, 95% selects). These are typical OLTP transactions about ordering new books and simple analytical transactions to collect detailed information about best-selling books on a specific topic (the Best Seller transaction). We used a dataset with 1,000,000 items and 2,880,000 customers. To reduce I/O contention and avoid I/O-subsystem bottlenecks, we set the value for shared_buffers to 20 GB for PostgreSQL. This setting ensured that all database tables in our experiments could fit in main memory, enabling I/O contention to be eliminated as a performance issue.

Table 1: Database Server Parameters

Processor	Intel Xeon E7-4860 [14]	AMD Opteron 6174 [15]
Sockets	4	4
Cores/Socket	10	12
Frequency	2.26 GHZ	2.2 GHZ
HW Contexts	80	48
L1D (per core)	32 KB	64 KB
L1 I (per core)	32 KB	64 KB
L2 (per core)	256 KB	512 KB
L3/LLC (shared)	24 MB	12 MB
Memory	32 GB	32 GB

5. Performance Evaluation

We compared the performance of the proposed CARIC-DA-1 and CARIC-DA-2 strategies with the PostgreSQL system that served as the baseline system. The throughput as a function of the number of concurrent clients is shown in Figures 7 and 8. We observed that the CARIC-DA systems performed better than the baseline system, On the Intel platform, we improved the throughput by 12%, whereas the AMD platform achieved a 6% improvement.

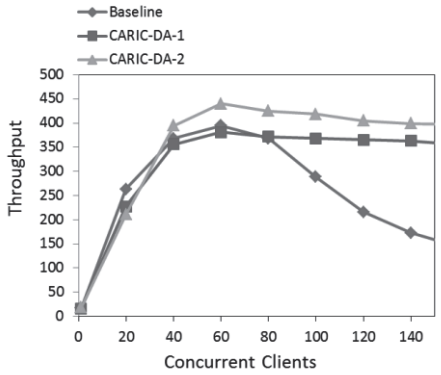

Figure 7: Throughput of different systems on the Intel multicore platform

For the Intel platform, the CARIC-DA-1 system performs a little worse than the baseline system when there are less than 80 concurrent clients. This is because partitioning each query into 10 subqueries involves excessive overhead. The database engine has to parse and create a proper query plan for 10 subqueries, whereas, for the baseline system, each join query is analyzed only once by the database engine. For small queries, the time used by the query parser and in creating the query plan cannot be ignored when compared with the query execution time in the database engine.

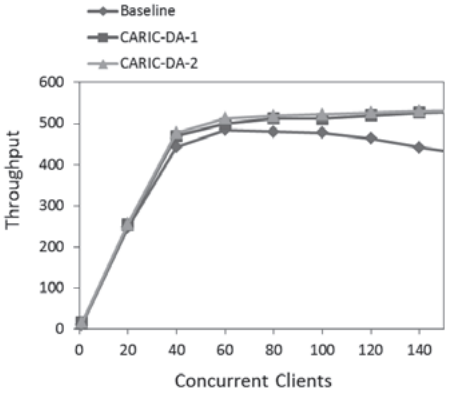

Figure 8: Throughput of different systems on the AMD multicore platform

In the baseline system, there is a dramatic performance decrease when setting up additional concurrent clients. However, we are setting up a fixed number of concurrent database processes equal to the number of the hardware threads supported by the specific multicore platform.

Therefore, the CARIC-DA system can avoid the performance decrease caused by too many concurrent database processes. However, we did observe a nonscalability problem for all the CARIC-DA systems and for the baseline system. We speculate that some components of the database engines are becoming bottlenecks and we would therefore seek to identify and resolve these bottlenecks in our future work.

6. Conclusion

To optimize concurrent-query execution on modern multicore platforms for DBMSs, we have proposed a middleware solution, "CARIC-DA", for OLTP applications. By dispatching the concurrent queries to appropriate processors, our proposal achieved higher performance by improving the private-cache utilization. However, for mixed workloads, such as those modeled in the TPC-W benchmark, it is difficult to provide cache data sharing for concurrent complex queries with join operations. Therefore, in this paper, we analyzed how to use the middleware of an extended CARIC-DA to manage the partitioning for complex queries and to dispatch the concurrent queries for mixed workloads on a multicore platform. We compared the performance of our proposed system with a pure PostgreSQL system on both Intel and AMD multicore platforms. Our proposed system achieved higher throughput on both platforms. However, we observed nonscalability for a mixed workload on modern multicore platforms. Therefore, improving system scalability will be investigated in our future work.

References

[1] J. Cieslewicz and K. A. Ross, "Database optimizations for modern hardware," Proceedings of the IEEE, vol. 96, no. 5, pp. 863–878, May 2008.

[2] N. Hardavellas, I. Pandis, R. Johnson, N. G. Mancheril, A. Ailamaki, and B. Falsafi, "Database servers on chip multiprocessors: Limitations and opportunities," CIDR, pp. 79–87, 2007.

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

[4] J. Rao and K. A. Ross, "Making B+tree cache conscious in main memory," SIDMOD, pp. 475–486, 2000.

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

[6] S. Harizopoulos and A. Ailamaki, "STEPS towards cache-resident transaction processing," VLDB, pp. 660–671, 2004.

[7] F. Xi, T. Mishima, and H. Yokota, "CARIC-DA: Core affinity with a range index for cache-conscious data access in a multicore environment," DASFAA, pp. 282–296, 2014.

[8] F. Xi, T. Mishima, and H. Yokota, "Optimizing concurrent query execution on modern multisocket multicore platform," FTSIS, pp. 125–130, 2014.

[9]Transaction processing performance council. TPC-C v5.5: On-line transaction processing (OLTP) benchmark.
http://www.tpc.org/tpcc/

[10] D. Menasce, "TPC-W: A benchmark for e-commerce," Internet Computing, IEEE, vol. 6, no. 3, pp. 83–87, 2002.

[11] R. Love, "Kernel korner: CPU affinity," Linux Journal, no. 111, p. 8, July 2003.

[12] L. Gilbert, J. Tseng, R. Newman, S. Iqbal, R. Pepper, O. Celebioglu, J. Hsieh, and M. Cobban, "Performance implications of virtualization and hyper-threading on high energy physics applications in a grid environment," in 19th IPDPS, p. 32a, 2005.

[13] PostgreSQL: The world's most advanced open source database.
http://www.postgresql.org/.

[14] Intel 64 and IA-32 Architectures Optimization Reference Manual.
http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[15] BKDG for AMD family 10h processors. http://support.amd.com/en-us/search/tech-docs?k=bkdg.

Fang XI

She is currently a Ph.D. student at the Tokyo Institute of Technology. She is engaged in research on data engineering and database systems.

Takeshi MISHIMA

He received his D.Eng. degree from Tokyo University in 2010. He has been with the NTT Corporation since 1996. His research areas include database systems and cloud computing.

Haruo YOKOTA He is currently a Professor in the Department of Computer Science at the Tokyo Institute of Technology. His research interests include the general research areas of data engineering, information storage systems, and dependable computing.