CARIC-DA: Core Affinity with a Range Index for Cache-Conscious Data Access in a Multicore Environment

Fang Xi[1], Takeshi Mishima[2], and Haruo Yokota[1]

[1] Department of Computer Science, Tokyo Institute of Technology, Japan
[2] Software Innovation Center, NTT Japan
xifang@de.cs.titech.ac.jp, mishima.takeshi@lab.ntt.co.jp,
yokota@cs.titech.ac.jp

Abstract. In recent years,the number of cores on a chip has been growing exponentially, enabling an ever-increasing number of processes to execute in parallel. Having been developed originally for single-core processors, database (DB) management systems (DBMSs) running on multicore processors suffer from cache conflicts as the number of concurrently executing DB processes (DBPs) increases. In this paper, we propose CARIC-DA, middleware for achieving higher performance in DBMSs on multicore processors by reducing cache misses with a new cache-conscious dispatcher for concurrent queries. CARIC-DA logically range-partitions the data set into multiple subsets. This enables different processor cores to access different subsets by ensuring that different DBPs are pinned to different cores and by dispatching queries to DBPs according to the data partitioning information. In this way, CARIC-DA is expected to achieve better performance via a higher cache hit rate for each core's private cache. It can also balance the loads between cores by changing the range of each subset. Note that CARIC-DA is *pure* middleware, which avoids any modification to existing operating systems (OSs) and DBMSs, thereby making it more practical. We implemented a prototype that uses unmodified existing Linux and PostgreSQL environments. The performance evaluation against benchmarks revealed that CARIC-DA achieved improved cache hit rates and higher performance.

Keywords: multicore, OLTP, middleware.

1 Introduction

In the computer industry, multicore processing is a growing trend as single-core processors rapidly reach the physical limits of possible complexity and speed [1]. Nowadays, multicore processors are widely utilized by many applications and are becoming the standard computing platform. However, these processors are far from realizing their potential performance when dealing with data-intensive applications such as database (DB) management systems (DBMSs). This is because advances in the speed of commodity multicore processors far outpace advances in

S.S. Bhowmick et al. (Eds.): DASFAA 2014, Part I, LNCS 8421, pp. 282–296, 2014.
© Springer International Publishing Switzerland 2014

memory latency [2], leading to processors wasting much time waiting for required items of data.

The cache level therefore becomes critical for overcoming the "memory wall" in DBMS applications. Some experimental researches indicate that the adverse memory-access pattern in DB workloads results in poor cache locality and imply that data placement should focus on the last-level cache (LLC) [3,4]. However, as increasing numbers of cores are becoming integrated into a single chip, researchers' attention is being diverted from the LLC to other cache levels. For modern multicore processors, it is usual to provide at least two levels of cache for the private use of each processor core in addition to an LLC for data sharing between several cores. For example, the AMD Opteron 6174 [5], which has 12 physical processor cores, provides two levels of private cache for each core, namely a 64-KB level-one (L1) instruction cache, a 64-KB L1 data cache, and a 512-KB level-two (L2) cache. Recent research indicates that increasing the number of cores that share an LLC does not cause an inordinate number of additional cache misses, with different workloads exhibiting significant data sharing between cores [6]. As the cache levels become more complex, access to the LLC involves more clock cycles because LLC access latency has increased greatly during recent decades. These changes in cache levels indicate that it is increasingly important to bring data beyond the LLC and closer to L1. In this paper, we first analyze how various scheduling strategies for concurrent DB processes (DBPs) on different processor cores affect the performance of private-cache levels, which are closer to the execution unit than is the LLC. We then propose a middleware-based solution to provide efficient data access to the private-cache levels for concurrent OLTP-style transactions on a multicore platform.

1.1 Private-Cache Contentions

For multicore systems, all concurrent DBPs dealing with the various queries will be scheduled to run concurrently on different processor cores. However, a different DBP-schedule decision will lead to different cache performance.

For example, Figure 1 shows three queries of Q1, Q2 and Q3 need to be dispatched to run on the two processor cores Core1 and Core2. In Plan1 of Figure 1, Q1 and Q2 are dispatched to co-run on Core1. Q1 is executed first and the data in the range [1–50] are loaded into the private cache of Core1. After the execution of Q1, the context switches to Q2. The data needed for Q2, namely [1–40], are already cached at that cache level and the resulting cache hit is as desired. However, the situation for Plan2 of Figure 1 is different. If Q1 and Q3 are dispatched for the same core, consider the situation when the execution of Q1 has finished and Q3 starts to run. The existing data, in the range [1–50] are not used at all by Q3. The cache has therefore to reload a new data set [50–100].

For OLTP applications which have big database, the whole data set is not uniformly accessed, and there is access skew for different subsets. If the queries which accessing data in the same subset co-run on one core, cache hit rate becomes better. Furthermore, the appropriate co-running of multiple DBPs on the same core (Plan1) can restrict the data access for each core's private cache

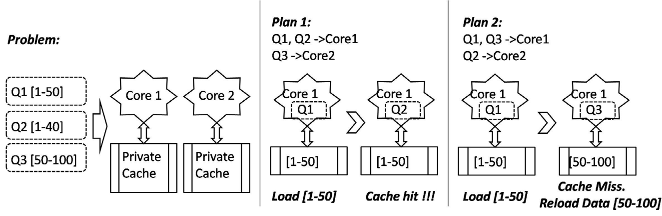

Fig. 1. Different query schedule strategies will cause different cache utilizations for private caches

to within a specific subset of the data in the whole DB, with the probability of cache hits thereby being improved.

The objective of our work is to provide a good co-running solution for concurrent queries to achieve higher private-cache utilization. The DBMS contains the data needs information for all its DBPs, and it has the ability to provide a good co-running strategy for the various DBPs. However, the scheduling of DBPs on processor cores is decided by the OS. The OS has no information about the co-running strategy provided by the DBMS.Therefore, in our approach, we achieve better private-cache utilization by importing OS functions into existing DBMSs.

1.2 The CARIC-DA Framework and Our Contribution

In this paper, we propose a framework—Core Affinity with Range Index for Cache-conscious Data Access (CARIC-DA)—that ensures that queries accessing the data set in a specific value range always run on the same processor core.

We use the core-affinity setting function provided by the OS to ensure that each DBP will always run on a specific processor core. The predefined value-range-based execute-core selection strategy serves as a range index for the CARIC-DA framework. CARIC-DA provides a function that enables the dispatching of all queries accessing the data in the same value range for execution by a specific DBP. The load balance between the processor cores is managed by a dynamic load-balancing process in CARIC-DA that adjusts the value range in the range index, according to the skew.

A major feature of our proposed solution is that all the functions provided by the CARIC-DA framework can be implemented as middleware over the existing OS and DBMS. Furthermore, CARIC-DA is *pure* middleware that does not require any modification to existing OSs and DBMSs, which is important because software for an existing DBMS is usually very large and complex, and any modification would be a time-consuming challenge.

The CARIC-DA is designed to optimize the performance for online transaction processing (OLTP) systems. A typical OLTP workload consists of 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 which are relatively small. In contrast,

the OLAP applications with queries involving aggregation and join operations on large amounts of data cannot benefit from our middleware. Although this may appear a limitation, it is not so in the context of database applications as there is no solution optimal for all applications [7].

There are different access skews in both of the transactions and data subsets. Some transactions occur much more frequently than other transactions. The frequent occurring transactions only access specific parts of some tables instead of all tables. Therefore, even the whole database for the real world OLTP applications might be in different sizes, the primary working set is not changed too much. The relatively modest primary working set can be captured by modern LLC [6]. As more cache resources are becoming integrated into a single chip, there is much possibility to cache the modest primary working set in the higher cache levels (private caches). To our knowledge, CARIC-DA is the first multicore-optimized middleware addressing the private-cache levels for concurrent queries. Furthermore, our paper gives several insights about how individual cache levels contribute towards improving performance through detailed experiments. Especially we highlight the role of the L2 cache. Experiments show that our proposal can reduce the L2 cache miss rate by 56% and reduce the L1 cache miss rate by 6%–10%. In addition, CARIC-DA can increase the throughput by up to 25% for a TPC-C workload [8].

2 Related Work

In recent years, enlarged LLC caches have been used in an attempt to achieve better performance by capturing larger working sets. Unfortunately, many DB operations have relatively modest primary working sets and cannot benefit from larger LLCs. Furthermore, larger LLC caches require more time to service a hit. Hardavellas *et al.* [6] noticed this problem and pointed out that DB systems must optimize for locality in high-level caches (such as L1 cache) instead of LLC. STEPS [9] improves the L1 instruction-cache performance for OLTP workloads, but improvement in data locality remains a problem for high cache levels. Our CARIC-DA focuses on both of the L1 and L2 caches (private-cache levels).

The DORA [10] is a typical work which focus on the optimization for OLTP workloads on multicore platforms. Furthermore it shares similarities with CARIC-DA at the idea of range partitioning. Rather than the CARIC-DA which optimizes the cache performance, DORA is designed to reduce the lock contentions. DORA decomposes each transaction to smaller actions and assigns actions to different threads. It may be very challenging for existing DBMSs to benefit from the DORA, as making the conventional thread-to-transaction DBMS to support multiple threads for one transaction is rather complex. On the other hand, our CARIC-DA does not touch any existing functions of existing DBMSs.

The MCC-DB [4] introduces functions from the OS to improve cache utilization for DBMSs. However, they introduce a cache-partitioning function, which is not supported by general purpose OSs. In contrast, we rely on the CPU-affinity function, which is well supported by most modern OSs.

The CPU-affinity function is already used for performance improvement in a variety of applications. Foong et al. [11] present a full experiment-based analysis of network-protocol performance under various affinity modes on SMP servers and report good gains in performance by changing only the affinity modes. However, their experiences and results are only limited to network applications and cannot be directly adopted by DBMS applications, which are much more complex and have intensive data accesses.

3 The CARIC-DA Framework

To provide good private-cache utilization for each processor core, we should consider dispatching those queries that access the same data set to enable execution on the same core. One straightforward approach would be to change the process-scheduling strategy in the existing OS. However, considering the complexity of existing OSs, this would be impractical. Our CARIC-DA offers a more practical approach which can be easily implemented as *pure* middleware over existing software, with no modification being needed in either the DBMS or the OS.

3.1 Design Overview

Different queries have different data needs, and this determines how much a query can benefit from accessing accumulated private-cache data. Therefore, we should co-run different queries according to their data needs. In principle, the following two points are critical to achieving good private-cache utilization. First, we make queries that access data in the same value range co-run on the same processor core. This strategy can enable a query to reuse the cached data loaded by a forerunner. Second, we make queries that access data in different value ranges run on different processor cores. This is because queries with the same data needs that co-run with a query with different data needs will cause a cache-pollution problem in which the frequently accessed data are replaced by one-time-accessed data. Our CARIC-DA framework broadly follows these two principles and achieves its goal by the two-step strategy described below.

1. Data Set and DB-process Binding. The first step is for CARIC-DA to associate each DBP with a disjoint subset of the DB and to ensure that queries that access data in the same subset are executed by the same DBP. As an example, consider a table with 3k lines. Suppose that there are three DBPs, namely DBP1, DBP2, and DBP3, in our system. Each DBP can access a disjoint subset of 1k lines, such as DBP1 accessing the lines numbered 1–1k. All queries that access lines numbered 1–1k will be dispatched for execution by DBP1, which can access the data in this subset, but queries with a data need for line number 1500 will be assigned to DBP2.
2. DB-process and Processor-core Binding. Second, we aim to force each DBP to run only on a specific processor core by setting the CPU affinity for each DBP. The core affinity setting is achieved via a function provided by the

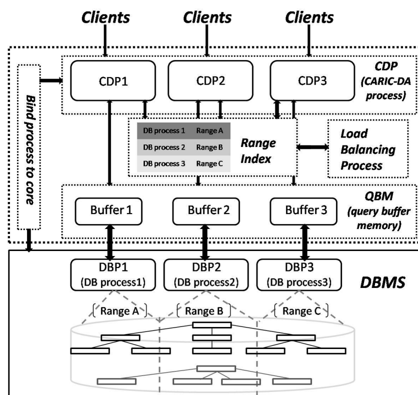

Fig. 2. Extensions of the CARIC-DA framework to conventional DBMSs

Linux OS called the CPU affinity. For example, if a CPU affinity of 1 is set for DBP1, this DBP will always run on processor core 1 and will never be scheduled for any other processor core by the OS.

In the first step, we create a binding between a data set and a DBP. In the second step, we create a binding between a DBP and a processor core. In this way, we reach the goal, namely bindings between data sets and cores. For the example, all queries that access lines numbered 1–1k are dispatched to run on processor core 1. In addition, all queries accessing data in different subsets are dispatched to run on other processor cores. This satisfies the two critical points needed to achieve the good private-cache utilization described above.

3.2 Core Components of CARIC-DA

Figure 2 shows an overview of the CARIC-DA framework and its main components, which extend an existing DBMS and OS by introducing a middleware subsystem comprising several subcomponents.

The binding between data set and DBP is achieved by a logical partitioning of the whole DB. A horizontal partition is adopted for each table. We first choose a suitable field for a table, which serves as its partition key. A mapping strategy for coupling the different partition key ranges with different DBPs is then formulated and stored in our CARIC-DA as the range index (RI). If there are multiple tables, the various tables are partitioned or overlapped, and mapped separately to all DBPs. That is, each process is mapped to several subsets for the different tables.

Theoretically any field of a table can be served as a partition key. But in practice the fields of the primary key of the table or only a subset of them can achieve good results as we have seen in our experiments. For multi-tables, it's better to choose the keys which can be commonly used to well partition most of the main tables (comparatively big and frequently accessed table). For example, the primary key of the Customers table of the TPC-C database consists of the

Warehouse id, the District id and the Customer id. The partition key fields may be the Warehouse id and the District id. The Warehouse id and the District id can also well be used to divide the other main tables. Furthermore, it is not necessarily to divide all of the tables in the database, as it is a trade-off between cache hit gain and query transfer cost.

The CARIC-DA processes (CDPs) use the mapping information provided by the RI to dispatch queries to DBPs with different data needs into an appropriate buffer in the query buffer memory (QBM). There is no physical partitioning of the DB or changes to the original DBMS, because the mapping work is done by the CDPs in terms of decomposing and dispatching the various queries according to the RI while considering access skews. Instead of obtaining queries directly from clients, each DBP only deals with queries in a specific buffer where the queries access only a specific data set.

3.3 Query Processing in CARIC-DA

In a traditional DBMS, all queries coming from the same client connection are always executed by the same DBP. In our CARIC-DA framework, CDPs communicate with the clients and dispatch the queries to different DBPs. For example, in Figure 2, CDP2 receives a "select" query from a client. CDP2 first checks the RI to identify the appropriate mapping between the data set related to this select query and the DBPs. Suppose DBP1 can access the data set this query requires. The query will then be put into Buffer1 in the QBM for checking by DBP1. Whenever DBP1 detects there is a query request, it picks up the query from Buffer1. The select query is executed and the answer is put back into Buffer1 by DBP1. CDP2 then transfers the answer in Buffer1 back to the relevant client.

When dealing with a range query that accesses a large data range, the query request can be divided into several subqueries that are transferred to several different buffers. If queries require data that are mainly mapped to one DBP, but that have a small portion (several lines of a table) being mapped to another DBP, we may prefer to dispatch this kind of query to one DBP instead of two different DBPs. This is because the query-and-answer transfer is an additional overhead for our platform, compared with traditional DBMS implementations. At the system level, if the query-dispatching function provided by the CARIC-DA framework can bring sufficient improvement to the whole system, the query transfer will be worth it. We can therefore suppose that a small imbalance will not affect the overall performance, because the imbalance will relate to only a few lines of the table and might not justify the extra query-transfer cost.

How to deal with cross-partition transaction is a challenge to our system as not all of the data sets can be well partitioned in many applications. For transactions which have lower isolation requirements, our middleware can decompose one transaction into several sub-transactions and dispatch them to different partitions. However we avoid transactions which need very high isolation level to cross different partitions. For the TPC-C benchmark, we assign the New-order transaction to a specific partition according to the district it related leaving the Item table no partitioned.

3.4 Processor-Core Binding for DBPs

Our DBP and processor-core binding is achieved by the CPU-affinity function, which is the Linux-based ability to bind one or more processes to one or more processor cores [12]. Other OSs, such as Windows Vista, have long provided a system call to set the CPU affinity for a process. On most systems, the interface for setting the CPU affinity uses a “bitmask.” Each bit indicates the binding of a given task to the corresponding processor core. In Linux, 11111111111111111111111111111111=4,294,967,295 is the default affinity mask for all processes. Because all bits are set to 1, the process can run on any processor core. Conversely, the bitmask 1 is much more restrictive. With only bit 0 being set, the process can run only on processor core 0.

In our framework, we set the CPU affinity for both of the DBPs and CDPs. For data-intensive applications, the DBPs make large data-access demands and mainly access the data in the DB. The CDPs make frequent accesses to the RI and QBM. We therefore bind the CDPs and DBPs that access different data sets to different processor cores to avoid cache-pollution problems.

3.5 Load Balancing

Our framework can not only maintain better cache locality, but also balance the loads across different processor cores. For example, there were 1,000 queries, with all of them using the same value range in a data set. Under the mechanism of static partitioning, all the queries would be dispatched to run on a single processor core, which would cause load imbalance between processor cores and result in poor performance. CARIC-DA attempts to balance the load between DBPs by modifying the value range mapped to different DBPs and to synchronize the CDPs in dispatching the queries, based on the new data-set mapping information of the RI. The load-balancing process regularly gathers the query numbers processed by each of the N DBPs as a measure of the load on each DBP (l_i) and then calculates the sum of l_i as l_{sum}. The DBP with the biggest load and the DBP with the smallest load are identified. The load difference is calculated as $l_{diff} = l_{big} - l_{small}$ and the skew is defined as $skew = \frac{l_{diff}}{l_{sum}}$. Tolerating a small load imbalance is achieved by omitting the rebalancing process whenever $skew$ is below a threshold. When the skew is bigger than a threshold, Algorithm 1 is invoked to calculate new data sets (R_i is the range of the ith data set) for the DBPs.

To describe the algorithm, let l_{ave} be the ideal load for each partition. Then the ideal load for each partition is: $l_{ave} = \frac{\sum_{i=1}^{N} l_i}{N}$. In the first part of the algorithm, we create N_{sub} subsets by dividing the ranges with $l_i > l_{ave}$ into subsets. We denote the load and range of each subset as sl_i and sR_i, respectively. We ensure that $sl_i < l_{ave}$ during the partition process and calculate the data range sR_i for the subsets. In the next part of the algorithm, we merge these subsets into N data sets that have $l_i = l_{ave}$. The new data set information will then be written back to the RI and all CDPs are synchronized to dispatch the queries according to the new RI.

Algorithm 1. Calculating new value ranges for DBPs

Create N_{sub} subsets by dividing sets with $l_i > l_{ave}$ into subsets
Let SN be the serial number of sub-sets, $SN \leftarrow 1$
for i in $[1, N]$ **do**
  if $l_i > l_{ave}$ **then**
    divide the data set into μ sub-sets where $\mu \leftarrow \lceil \frac{l_i}{l_{ave}} \rceil$;
    calculate the load and range for each sub-set SN as $sl_{SN} \leftarrow \frac{l_i}{\mu}$, $sR_{SN} \leftarrow \frac{R_i}{\mu}$, $SN \leftarrow SN + 1$;
  else
    $sl_{SN} \leftarrow l_i$, $sR_{SN} \leftarrow R_i$, $SN \leftarrow SN + 1$;
Merge N_{sub} subsets into N sets with $l_i = l_{ave}$, $j \leftarrow 1$
for i in $[1, N]$ **do**
  $l_i \leftarrow l_{ave}$, $R_i \leftarrow 0$
  while $l_i > sl_j$ **do**
    merge sub-set j into set i ($R_i \leftarrow R_i + sR_j$), $l_i \leftarrow l_i - sl_j$, $j \leftarrow j + 1$
  if $l_i > 0$ **then**
    merge a part of sub-set j into set i ($R_i \leftarrow R_i + \frac{l_i}{sl_j} \times sR_j$), $sR_j \leftarrow 1 - \frac{l_i}{sl_j} \times sR_j$

4 Performance Evaluation

We implemented the CARIC-DA-related functions in the C language as a middleware subsystem over the existing DBMS (PostgreSQL) and the OS (Linux). The Oprofile [13] was used to examine the hardware-level performance. We compared our CARIC-DA–PostgreSQL system against an unmodified PostgreSQL system (Baseline) to investigate the efficiency of our proposed framework.

We used a 48-core DB-server machine for answering clients' queries. The DB-server machine had four sockets, with a 12-core AMD Opteron 6174 processor per socket [5]. Each core had a clock speed of 2.2 GHz and had a 128-KB L1 cache and a 512-KB L2 cache. All 12 cores of a processor shared a 12-MB L3 cache. The server had 32 GB of off-chip memory, and two 500-GB hard-disk drives. Each of the four client machines had a Intel Xeon E5620 CPU and a 24-GB memory. To prevent the I/O subsystem from becoming a bottleneck, we set the value of *shared_buffers* to 20 GB for the PostgreSQL. This setting ensured that all DB tables in the following experiments could fit in main memory.

We first used a microbenchmark evaluation to isolate the effects and to provide in-depth analysis. We then used the TPC-C benchmark [8] to further verify the effectiveness of our proposal. Our evaluation covered seven areas.

(1)Different core-affinity strategies in our CARIC-DA system.

(2)The separate impacts of CARIC-DA on performance of select-intensive and insert-intensive workloads.

(3)The scalability of CARIC-DA.

(4)The advantages of CARIC-DA for different cache levels.

(5)The performance of CARIC-DA with data sets of different size.

(6)The efficiency of CARIC-DA when dealing with skewed data sets.

(7)The efficiency of CARIC-DA in handling the TPC-C benchmark.

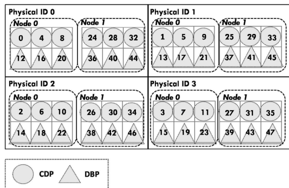

Fig. 3. Uniform mixing of CDPs and DBPs in each Node

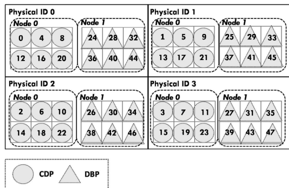

Fig. 4. Clustering CDPs or DBPs in each Node

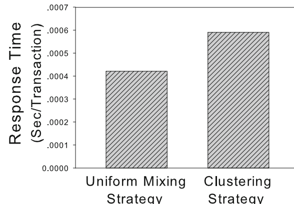

Fig. 5. Performance under different core-affinity settings

4.1 Core Affinity in CARIC-DA

We first decided how to set the core affinity for the various DBPs and CDPs in our CARIC-DA system. Figure 3 shows the physical location relationship for the different logical cores. There are four processors in our platform, with different physical IDs. In each processor, there are 12 cores with different logical IDs, shown as several squares located in the two Nodes [14]. We examined the performance under two core-affinity strategies for a system with 24 CDPs and 24 DBPs using the select-intensive transaction from the microbenchmark. Each transaction randomly accesses one line of a TPC-C stock table.

(1)Uniform Mixing: a mixture of DBPs and CDPs in one Node structure (Fig. 3).

(2)Clustering: all CDPs are bound to cores located in Node 0 and all DBPs are bound to cores located in Node 1 (Fig. 4).

The average response time with 24 concurrent clients is shown in Figure 5. The strategy involving the uniform mixing performs better than the strategy of clustering. Each Node is an integrated-circuit device that includes several CPU cores, up to four links for general purpose communication to other devices (such as an L3 cache, main memory interfaces). Compared with the CDP, the DBP is data intensive and makes frequent data requests to both cache and memory. In the clustering strategy, all six data-intensive DBPs are bound to the same Node, leading to intensive competition for the shared Node resources. The competition results in a longer L3-cache and memory-access latency and a comparatively worse performance. Therefore, we used the uniform-mixing strategy exclusively for the CARIC-DA system in subsequent experiments.

4.2 Microbenchmark Evaluation

In the select-intensive experiment, clients repeatedly send select transaction requests. The select transaction for the microbenchmark has only one query request for a specific line in the table. In a similar fashion, the insert transaction request comprises one record-insertion operation. We prepared a DB with only one table, namely a stock table from TPC-C comprising 100,000 lines. Both

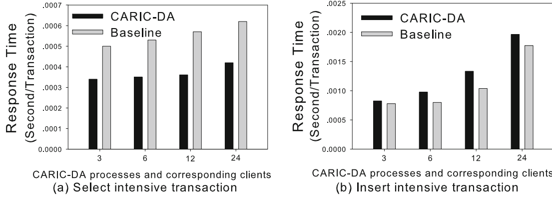

Fig. 6. Effectiveness of the CARIC-DA framework for separate select-intensive and insert-intensive transactions

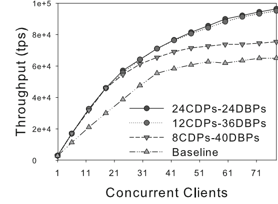

Fig. 7. Throughput of CARIC-DA systems

select and insert operations randomly generated search keys within the data range of the 100,000 lines. For CARIC-DA system, we separately set up 3, 6, 12, and 24 CDPs, and 3, 6, 12, and 24 DBPs. There were also 3, 6, 12, and 24 clients, with different clients connecting to different CDPs. In Baseline system, the clients connected directly to the PostgreSQL.

Figure 6 (a) indicates that the select operation can benefit significantly from our CARIC-DA architecture. When there were 24 concurrent clients, the select transactions are executed 33% faster under CARIC-DA. These results demonstrate the effectiveness of our CARIC-DA framework in providing better performance for select-intensive workloads. However, insert-intensive operations did not benefit from our CARIC-DA framework. For each insert operation, the related DBP has to access not only the table data but also some common data, such as index data and metadata. For example, two DBPs have to update the same node of the index if the index node is already cached by different DBPs. The data in one cache will be updated, but this update operation will invalidate the copy of the data in the other cache. This kind of cache-conflict problem, caused by accessing common data, cannot be avoided in either the Baseline system or the CARIC-DA system. The extra transmission cost of queries in CARIC-DA will then lead to a worse overall performance than for the Baseline system.

4.3 Performance of CARIC-DA under Different Loads

We set up 24 CDPs and 24 DBPs in the DB-server section and repeated the select-intensive experiment. We gradually increased the number of concurrent clients to 78 clients. For 78 concurrent clients, the throughput of the CARIC-DA system is 48% higher than the Baseline system (Fig. 7). The Baseline system can linearly scale to about 36 concurrent clients, with the rising trend of overall throughput greatly reducing when there are more than 36 concurrent clients. This is because the memory bandwidth limits the performance as the number of concurrent clients increases. In contrast, the 24CDPs–24DBPs system with its well-designed cache accesses does not show such a decrease in the rising trend in overall throughput. The CDP will generate one client thread to deal with the queries in one client connection, and we use the System-V semaphores to

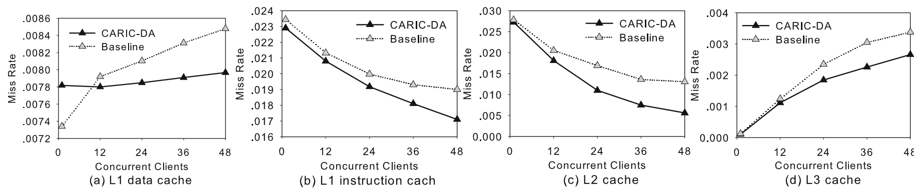

Fig. 8. Cache miss rates

synchronize the accesses to the QBM by these threads. With an increasing number of concurrent clients, any untimely scheduling of these semaphores will affect the system performance and result in a nonlinear throughput growth for the CARIC-DA system. We also set up two new CARIC-DA systems with 12CDPs–36DBPs and 8CDPs–40DBPs, respectively (Fig. 7). With reducing numbers of CDPs, there are more client threads bound to the same core. The performance of the CDPs limits the overall performance of the 8CDPs–40DBPs system. The select-intensive transaction is very short and the frequent query-and-answer transfers between clients and DBPs greatly stress the CARIC-DA middleware.

4.4 Cache Utilization

We repeated the select-intensive experiment for 24CDPs–24DBPs system, and measured the cache miss rate for the various cache levels separately to confirm that CARIC-DA is efficient because of its outstanding performance at the various cache levels. The miss rate is the percentage of misses per total number of instructions. We observed that the L1 data-cache miss rate increases from 0.73% to 0.84% as the number of concurrent client threads increases in the Baseline system (Fig. 8 (a)). However, for the CARIC-DA system, the L1 data-cache miss rate appears to be only slightly increased (from 0.78% to 0.79%). The CARIC-DA system can also improve the L1 instruction-cache performance by up to 10% (Fig.8 (b)). This is because all DBPs are restricted to running on a specific processor core, which can avoid frequent context switching in each core. The biggest cache performance improvement comes from the L2 cache, with an almost 56% reduction in cache misses compared with that for the Baseline system (Fig. 8 (c)). We also observe that the miss rate for the shared L3 cache can be reduced by 21% (Fig. 8 (d)).

4.5 Performance of CARIC-DA for Data Sets of Different Sizes

In this section, we describe the effectiveness of our proposal for a variety of data sets. We compared the average response time and cache utilization of the select-intensive application with 48 concurrent clients for a 24CDP–24DBP-based CARIC-DA system with those for the Baseline system, as shown in Figure 9. When increasing the data set to the much larger size of 1,130 MB (3,000,000 tuples), the performance gap between the CARIC-DA system and the Baseline system narrowed (Fig. 9 (a)). The diminished advantage of our proposal

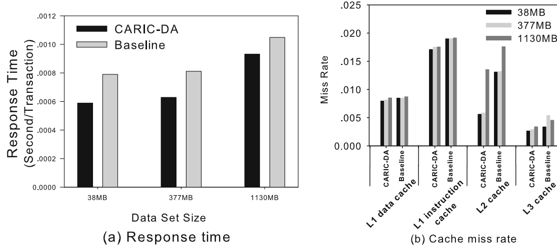

Fig. 9. Performance for data sets of different sizes

with very large data sets derives from our RI-based data-access strategy. In the CARIC-DA system, the L2 cache accesses a smaller subset, whereas the Baseline system's L2 caches each access the whole data set. However, as the size of the whole data set greatly increases to 1,130 MB, the subset size in our proposal is also greatly increased (to 47 MB). The sizes of both the whole data set and our subset are much beyond the capacity of the small L2 cache (512 KB). The size difference between the subset in our proposal and the whole data set in the Baseline system becomes less significant when compared with the size of the very small L2 cache, and the performance difference between the two systems decreases. For a real-world workload, data access is skewed and the frequently accessed data set is much smaller than the whole DB. Taking this data-access skew into account, our CARIC-DA system shows impressive advantages when dealing with GB-sized data sets, in comparison with the Baseline system.

4.6 Performance of CARIC-DA with Skewed Data Access

We use data sets of 3,000,000 tuples (table size 1,130 MB), 10,000,000 tuples (table size 3,766 MB), and 15,000,000 tuples (table size 5,649 MB). Fig. 10 plots the average response time for select-intensive transaction for the data set of 3,000,000 tuples. We calculate the average response time every 10 seconds. Initially, the distribution of the queries is uniform for the entire data set. However, at time point 50, the distribution of the load changes, with 50% of the queries being sent to 10% of the data set (Zipf (0.75) distribution). We did not use the dynamic load-balancing function of the CARIC-DA system until time point 100.

After the load change, the performance of the Baseline system improved slightly, while the performance of the non-load-balanced CARIC-DA system dropped sharply. After enabling dynamic load balancing in the CARIC-DA system, the performance improved dramatically, outperforming the Baseline system by 39%. For the other two data sets, the dynamic-load-balanced CARIC-DA system achieved improvements of 41% and 40% above the Baseline system's performance. These results confirm the efficiency of our proposal when dealing with skewed data sets and also substantiated the claim that skew favors the CARIC-DA system (the advantage is increased from 7.3% to 41%).

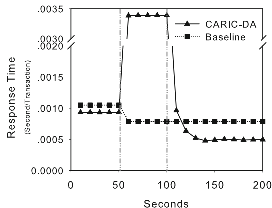

Fig. 10. Performance with skewed data access

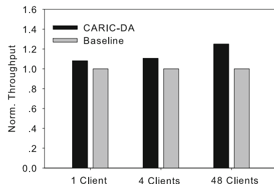

Fig. 11. Throughput for TPC-C benchmark

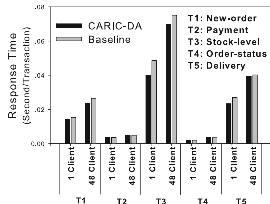

Fig. 12. Performance for different transactions

4.7 TPC-C Benchmarking

We used a 24-warehouse TPC-C data set (∼4.8 GB). For the CARIC-DA system, we set up 4 CDPs and 44 DBPs and evenly partitioned the 24 warehouses at the district level. For each transaction, the CDP has only to find the appropriate buffer in the QBM for the first query of the transaction, with any follow-up queries directly accessing the same QBM. By monitoring the CPU usage of CDPs we founded out that CDPs did not need a lot of CPU and 4 CDPs were sufficient for TPC-C transactions. As shown in Figure 11, the CARIC-DA system outperforms the Baseline system by 10% to 25%.

The average response times for the five kinds of transactions are shown in Fig. 12. The CARIC-DA system can optimize the execution time for New-order transactions by 7% and can greatly optimize the Stock-level transactions by 18%. The Stock-level transactions will retrieve item information for the most recent 20 orders in a specific district. In our system, a specific DBP only processes the New-order transactions, which will also access the item information for a specific district. When a Stock-level transaction follows a New-order transaction, there is a higher possibility of the item information for recent orders existing at the private-cache level. For the Baseline system, specific DBPs have to deal with the New-order transactions from all districts. For recent-order information in a specific district, therefore, the private-cache hit possibility will be relatively low. This explains why Stock-level transactions demonstrate great performance improvements for our proposed system. In the high-concurrency experiment, the New-order transactions can be further optimized by 11%. For concurrent transactions, more New-order transactions benefit from the CARIC-DA approach, compared with the no-concurrency situation. In addition, we observed that the CARIC-DA system can achieve lower abort rates for New-order transactions. The concurrent transactions which update the same data in a specific district may cause the serialization error in the Baseline system with the isolation level of serializable. In the CARIC-DA system, these concurrent transactions will be sequentially processed and lead to less serialization error. This is the other reason for the 25% throughput improvement with the CARIC-DA system.

5 Conclusions

In this paper, we introduced CARIC-DA, which is the first work addressing the problems for DBMSs of private caches on multicore platforms. Considering the different access skews in both of the transactions and data subsets, even the whole database for the real world OLTP applications might be in different sizes, the primary working sets are small and can be captured in modern LLC. We researched on the possibility to bring the primary working set beyond the LLC and closer to processor core (in the private cache levels). We give several insights about how individual cache levels contribute towards improving performance through detailed experiments and highlight the role of the L2 cache. CARIC-DA is implemented as *pure* middleware, enabling the existing DBMS to be used without modification. Not only can it maintain better locality for each core's private cache, but it can also balance loads dynamically across different cores. Experiments show that the L2 cache miss rate can be reduced by 56% for select-intensive transactions and the throughput can be improved by 25% for TPC-C transactions by using our CARIC-DA framework. In future work, we will use other hardware platforms in additional experiments to investigate the effectiveness of our framework. Providing better cache-access patterns for decision support systems will also be a future challenge.

References

1. Adee, S.: The data: 37 years of Moore's law. IEEE Spectrum 45, 56 (2008)
2. Cieslewicz, J., Ross, K.A.: Database optimizations for modern hardware. Proceedings of the IEEE 96, 863–878 (2008)
3. Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: DBMSs on a modern processor: Where does time go? In: VLDB, pp. 266–277 (1999)
4. Lee, R., Ding, X., Chen, F., Lu, Q., Zhang, X.: MCC-DB: Minimizing cache conflicts in multi-core processors for databases. In: VLDB, pp. 373–384 (2005)
5. AMD family 10h server and workstation processor power and thermal data sheet, `http://support.amd.com/us/Processor_TechDocs/43374.pdf`
6. Hardavellas, N., Pandis, I., Johnson, R., Mancheril, N.G., Ailamaki, A., Falsafi, B.: Database servers on chip multiprocessors: Limitations and opportunities. In: CIDR, pp. 79–87 (2007)
7. Salomie, T.I., Subasu, I.E., Giceva, J., Alonso, G.: Database engins on multicores, why parallelize when you can distribute? In: EuroSys, pp. 17–30 (2011)
8. Transaction processing performance council. TPC-C v5.5: On-line transaction processing (OLTP) benchmark
9. Harizopoulos, S., Ailamaki, A.: STEPS towards cache-resident transaction processing. In: VLDB, pp. 660–671 (2004)
10. Pandis, I., Johnson, R., Hardavellas, N., Ailamaki, A.: Data-oriented transaction execution. In: VLDB, pp. 928–939 (2010)
11. Foong, A., Fung, J., Newell, D.: An in-depth analysis of the impact of processor affinity on network performance. In: ICON, pp. 244–250 (2004)
12. Love, R.: Kernel korner: CPU affinity. Linux Journal (111), 8 (2003)
13. Oprofile: A system profiler for linux (2004), `http://oprofile.sf.net`
14. BKDG for AMD family 10h processors (2010), `http://support.amd.com/en-us/search/tech-docs?k=bkdg`