A Research on Efficient and Scalable DBMS for Many-core Based Platform

メニーコアプラットフォームのための効率的かつスケーラブルなDBMSに関する研究

Fang XI[1]

In recent years, dramatic improvements have been made in computer hardware. In particular, 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 management systems (DBMSs) cannot take full advantage of the parallel computing that uses so many cores. Therefore, this thesis analyzes the possibility of optimizing the performance of DBMS on modern multicore platforms and proposes two approaches to break the bottleneck in database engines and to provide cache-efficient query scheduling for concurrent queries for different database applications on modern multicore platforms. All of these proposals are pure middleware, which avoid any modification to existing DBMSs, thereby making them more practical. This is important because the source code for existing DBMSs is large and complex, making it very expensive to modify. The performance evaluation through benchmarks revealed that these proposals can improve throughput and scalability on typical modern multicore platforms for different database applications.

1. Introduction

Database management systems (DBMSs) are widely used to help users and applications control their data over the decades, by providing both efficient data update, data retrieval and durable, structured data storage. Since the birth of the DBMS, how to provide fast and timely data access has become into the challenge for researchers and the DBMS vendors. Most DBMSs were designed in the 1980s, when only uniprocessors were available and limited main memory space. The data are stored on the disks, and the main memory is used to buffer the usually accessed data in order to reduce the time consuming disk accesses. Research focused on improving the system performance through efficient buffer pool management and fine-grained multiplexing concurrent transactions to hide disk latency, as the disk I/O was the dominated bottleneck. Moreover, queries were optimized and executed independently of each other in a query-at-a-time processing model with few of the queries actually running simultaneously on the traditional platforms.

Nowadays the database systems are facing a different environment where the modern multicore processors are providing powerful parallel computing capabilities than traditional uniprocessors. Recently micro-processor manufacturers find that it has become increasingly difficult to make CPUs go faster due to size, complexity. So they continue the performance curve. That is, putting multiple CPUs on a single chip and relying on the parallelism ability to get higher performance gain which brings the computing world into so-called “multi-core era”. The multi-core processor is the mainstream now and there will be many-core processors with more cores on one die.

As the processor speed increased as a much higher speed than the development in the memory access speed, the multicore processors are costing more cycles in waiting intended data fetching from memory. The performance of the memory subsystem is greatly affecting the whole system for data intensive applications and unoptimized memory access pattern has the possibility to slow down the whole system. Microprocessor manufactures have also recognized the data starving problem on modern multicore platforms and are putting a much bigger on-chip cache space to address this performance gap. Therefore, efficient cache utilization and reducing the time consuming memory accessing operation become critical for efficient DBMS on modern multicore platforms. Optimizing cache performance often requires very careful tuning or the total re-writing for the data structures and algorithms. However, considering the complexity of modern commercial database engines, any modification is a challenge and time consuming work. Therefore, a more practical cache optimizing solution which can avoid the challenge work of modifying existing implementations of DBMSs will be welcomed and expected.

The increasingly powerful concurrent processing ability bought by multicore processors is stressing the database engines. A traditional approach of getting higher performance of processors is to increase the clock speed of them, since a faster CPU can finish one task quickly then switch to the next. In the multicore area, we can not benefit from the increasing of clock speed anymore, but we have to efficiently utilize the parallelism ability brought by increasing of a core number instead. That is, the software must have scalability and provide high parallelism to keep the multicore processor busy. The original concurrency ability which is mainly used to overlap delays in traditional DBMS cannot exploit heavy parallelism. The database applications can exhibit concurrently both inter and intra queries, and the database engines perform well on the platform with less than 10 cores. For the sudden boost of modern multicore and many core platforms with 40 to 100 hardware threads, the existing database engines are clearly not yet ready. Existing research pointed out some functions inside the database engines are becoming into new bottleneck and prevent the system scale to a high level on the multicore platform. Considering the complicity of DBMS, with

[1] Fang XI（西方）received her Ph.D in Computer Science from Tokyo Institute of Technology. She is engaged in research on data engineering and database systems.

increasing the concurrency of transactions, there will be many more potential bottlenecks exposed. For example, the concurrent update to the same table will cause severe contentions in the lock functions. Experiments to find the potential bottlenecks and removing these bottlenecks from database engines is very important for achieving high scalability and high throughput.

Motivated by these challenges on modern multicore and manycore platforms, this thesis analyzed the possibility to provide efficient and scalable DBMSs through optimizing the concurrent queries as a whole, and analyzed how to employ sharing for concurrent queries rather than optimizing each query independently for different kinds of workloads.

We proposed the CARIC-DA (Core affinity with a range index for cache-conscious data access in a multicore environment) [1] to optimize the high level data cache performance for OLTP applications on multicore platforms. We dispatch the concurrent database queries to run on different cores according to the different data needs of the concurrent queries. By co-running the queries with the same query needs onto the same processor core, these concurrent queries can share data in private cache levels and reduces the cache miss and the time consuming memory accesses.

Furthermore, with the other proposal of PM-DB (Partition-based multi-instance database system for multicore platforms) [2], we solved the contention in memory space management function of existing database engines for concurrent queries, by managing multiple database engines on a single multicore platform. By distributing the concurrent queries to different database engines, the original contention for a single engine can be eased and the whole system achieved higher scalability.

2. The CARIC-DA System

The data intensive applications of DBMSs are far from taking full advantage of the parallel processing capabilities provided by modern multicore platforms. This is largely due to the fact that the advance in the speed of multicore processors far outpaces that in memory latency, leading to data cannot be delivered fast enough to be consumed by the processor. The CPU cache, which is used to buffer the main memory data blocks to speed up accesses to frequently needed data, becomes critical for overcoming the "memory wall".

2.1 Related Works

Ailamaki et al. analyzed the memory hierarchy performance of commercial DBMSs and pointed out the importance of the last-level cache (LLC) [3]. A study on MCC-DB [4] pointed out the conflicts in the shared cache for concurrent queries on the multicore platform.

As the cache levels become more complex and the last-level cache size is scaled, the access to the LLC involves an increasing number of clock cycles. These changes in cache levels indicate that it is increasingly important to bring data beyond the LLC and closer to L1. Hardavellas et al. proposed STEPS [5], which minimizes instruction misses in the L1 cache based on the StagedDB design. However, reducing the data misses in higher cache levels is still a major challenge.

Therefore, in this approach, we first analyze how various scheduling strategies for concurrent DB processes on different processor cores affect the performance of private-cache levels, which are closer to the execution unit than the LLC. We then propose a middleware-based system to provide efficient data access to the private-cache levels for concurrent OLTP-style transactions on multicore platforms.

2.2 Motivation of CARIC-DA

A typical OLTP workload consists of a large number of concurrent short-lived queries, each accessing a small fraction of a large dataset. Furthermore, all concurrent DB processes (DBPs) dealing with various queries should be dispatched to run concurrently on different processor cores. However, a different DBP-dispatch decision will lead to different cache performance. For example, Figure 1 shows two query-dispatching strategies for four concurrent queries on two processor cores. In the cache-efficient solution (Schedule 1 in Figure 1), the two queries that access the same data are dispatched to run on one processor core. Q1 can reuse the cache data <1-50> which are already loaded into the cache after the execution of Q2. However, in the cache-inefficient solution (Schedule 2), Q1 runs after Q3, and it cannot reuse the cache data loaded by Q3. There will be more cache misses in the cache-inefficient solution than in the cache-efficient solution. Furthermore, the cache-efficient solution can restrict the data access for the private cache of each core to within a smaller subset, with the probability of cache hits thereby being increased.

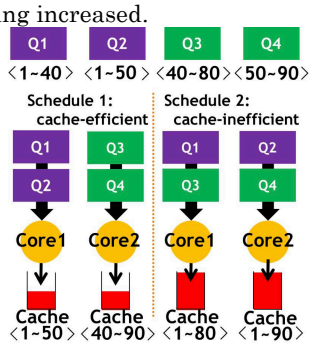

Figure 1: Different query-dispatching strategies and related private-cache performance

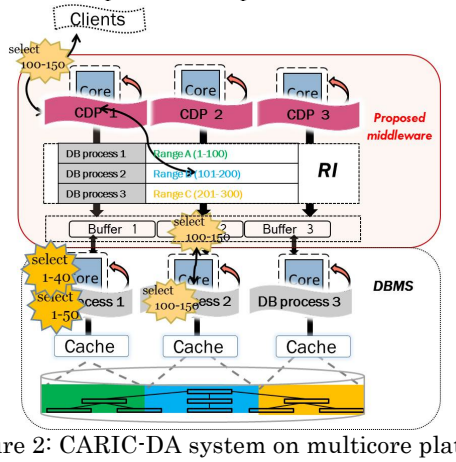

Figure 2: CARIC-DA system on multicore platform

2.3 Framework of CARIC-DA

To provide efficient private-cache utilization for each processor core, the queries that access the same dataset should be dispatched to run on the same core. However, the DBMS cannot ensure that, as the scheduling of DBPs

on processor cores is decided by the OS. CARIC-DA offers a practical approach with no modification needed in either the DBMS or the OS. It achieves its goal by a two-step strategy.

(1) Dataset and DB-process binding: CARIC-DA associates 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.

(2) DB-process and processor-core binding: CARIC-DA forces 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 Linux OS called the CPU affinity [6]. In this way, we reach the goal, namely, binding between datasets and cores.

The architecture of the CARIC-DA system deployed on a multicore platform is shown in Figure 2. 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. We introduce some middleware processes to dispatch the queries to be processed by different database processes according to the pre-defined database partitioning information. Therefore, we can make sure each processor core only deals with the queries accessing the data in a specific sub data set.

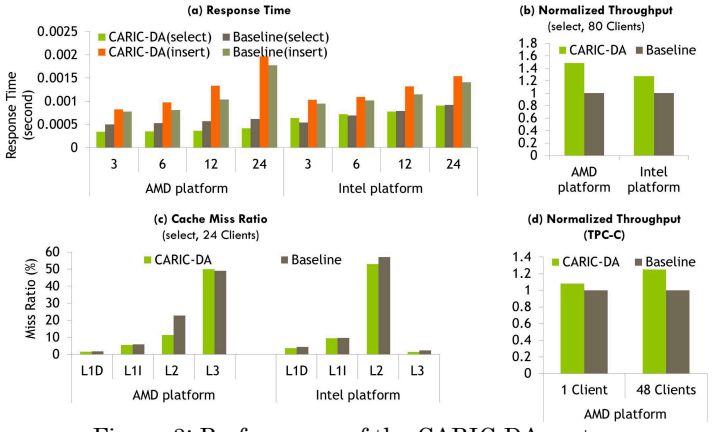

Figure 3: Performance of the CARIC-DA system

2.4 Experiments

The CARIC-DA-related functions were implemented in the C language over the PostgreSQL and the Linux. We used the micro benchmark to isolate the effects, and then we used the more complex TPC-C. The response time and scalability of the CARIC-DA compared with the Baseline system on different platforms are shown in Figure 3. The proposal can reduce the miss ratio for the L2 cache for about 50% on the AMD platform (48 cores) [7], while on the Intel platform [8], the L2 miss ratio is reduced by 7%. For the TPC-C benchmark with data set of 4.8GB on AMD platform (40 cores/80 vCPUs), the CARIC-DA system can achieve 25% throughput improvement with 48 clients.

3. The PM-DB System

The increasingly powerful concurrent processing ability of multicore platforms is stressing these DBMSs. An increasing number of concurrent database processes share resources both at the hardware (caches and memory) and at the software (locks) levels, and any inefficient resource sharing will create new bottlenecks in database systems. Mixed workloads such as those modeled in the TPC-W benchmark [9] are different from both typical OLAP and OLTP applications. There are neither as many complex queries as in OLAP applications, nor as many severe update operations as in typical OLTP workloads. To this end, the optimization of mixed workloads on multicore platforms is still a challenge for existing database engines.

We analyzed the scalability of the mixed workload using the TPC-W benchmark on a modern Intel E7 multicore platform with 80 hardware contexts. Results show that the system encounters severe scaling problems. There are 80 hardware threads on the multicore platform, although the system could only scale to 40 concurrent clients. We further analyzed the CPU utilization for the system using the performance monitoring tool Perf. As the number of concurrent clients increases, the "s_lock function" in PostgreSQL becomes the system bottleneck. Each database engine holds one shared buffer in memory and the concurrent loads access the shared memory space through specific synchronization functions. The "s_lock" function is the hardware-dependent implementation of a spin lock and it is used to control access to the shared memory-critical sections in PostgreSQL [10][11]. Therefore, we can conclude that the contention for the shared memory significantly degrades the scalability of the database engine on a multicore platform that could otherwise offer rich true concurrency not provided on single-core platforms. Considering the complexity of most commercial database engines, the rewriting of existing database engines will be a very challenging task [12]. Therefore, we proposed the PM-DB, a partition-based multi-instance solution that was originally used in shared-nothing parallel database systems. PM-DB exploits the parallelism offered by multicore through the combined performance of a collection of unmodified database engines, rather than through the optimization of a single engine modified to run on multicores.

3.1 Architecture Overview

The architecture of the PM-DB system deployed on a single multicore platform is shown in Figure 4. We set up several database instances rather than one on the multicore platform and partition the whole database into the different database instances. The detailed database partition information is stored in the local data structure as a Range Index. Each database instance's process monitors its Query Buffer, and takes the query from the buffer when it arrives. After execution, the query result will be written back into the query buffer and the Query Dispatcher then returns the answer to the client.

For some applications, it will be difficult to provide a clean partition; that is, some transactions will require data that are stored in different instances. The PM-DB's Query Dispatcher function can dispatch the transactions to several instances that hold the transaction required data. Moreover, for the update transactions that access multiple instances, we provide the two-phase commit protocol to ensure that either all the instances are updated or none of them; therefore, the database instances can remain synchronized with each other. The multicore-based two-phase commit has much lower overhead than that in traditional distributed

environments, as the data communication is much faster through the interconnections in the single multicore platform than in network communications. For applications with frequent crossing-instance join operations, the data transfer between different instances may slow down the transactions. Therefore, our middleware can provide data duplication to avoid intensive data transfers between instances by redundantly storing some data in several instances.

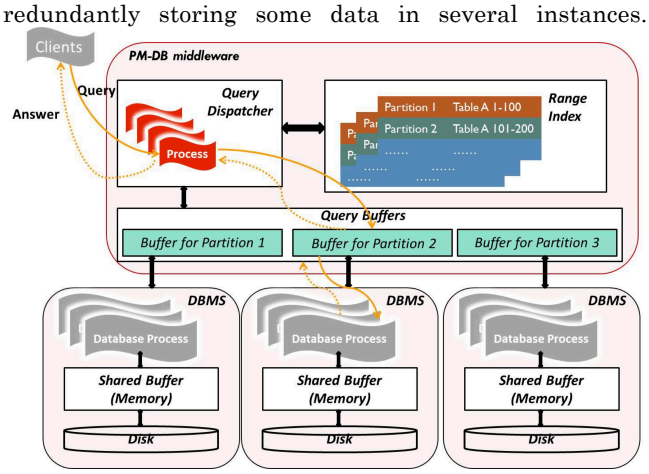

Figure 4: PM-DB system deployed on a multicore platform

3.2 Cache Optimization

We consider a modern multisocket and multicore platform, and propose two strategies to manage the running of the concurrent database processes on the multicore platform. First, we propose to run the database processes that access the data in different database instances on different processors. This strategy can avoid forcing queries that access data in different data partitions to compete for Last Level Cache (LLC) resources. Second, for the database processes in each database instance, we propose to separate different types of queries to run on different processor cores. This strategy can increase the performance of private cache levels. If we run simple queries together on the same processor core, there will be a higher possibility that these queries can share high-level index data in the private cache levels. However, if complex queries (with join operations) run together with simple queries on the same core, the one-time accessed hash data or a big range of table data of complex query will evict the frequently used index data of simple queries from the private cache.

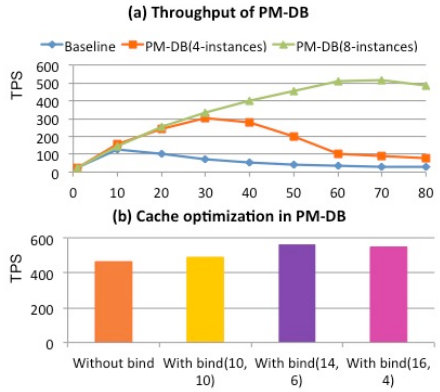

Figure 5: Performance of the PM-DB system on Intel platform

3.3 Performance Evaluation

We analyzed the efficiency of our proposed PM-DB system on a modern Intel multicore platform using the TPC-W benchmark. The hardware of the DB server is the Intel Xeon E7 system that serves as 80 virtual CPUs with Hyper-Threading. By setting up more database instances, the contention in the single database engine can be reduced and experiments show that our proposal achieved at most 2.5 times higher throughput than the Baseline system (Figure 5 (a)). Moreover, the cache-efficient query dispatching provided by the PM-DB can further improve the system throughput by 21% (Figure 5 (b)).

4. Conclusion

In this thesis, I research on the performance and opportunity for providing an efficient and scalable DBMS on modern multicore and many-core platforms. Two novel approaches which are CARIC-DA and PM-DB are proposed, for overcoming the processor memory gap and non sacalable bottlenecks in database engine for OLTP and mixed applications. Experiments show that the CARIC-DA and PM-DB systems achieved higher performance on both of modern AMD and Intel platforms.

References

[1] 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.

[2] F. Xi, T. Mishima, and H. Yokota, " PM-DB: Partition-based multi-instance database system for multicore platforms," ICEIS, pp. 128-138, 2015.

[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] 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.

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

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

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

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

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

[10] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: A scalable storage man- ager for the multicore era," In EDBT, pp. 24–35, 2009.

[11] T. I. Salomie, I. E.Subasu, J. Giceva, and G. Alonso, "Database engines on multicores, why paral- lelize when you can distribute?" In EuroSys, pp. 17–30, 2011.

[12] G. Giannikis, G. Alonso, and D. Kossmann, "SharedDB: killing one thousand queries with one stone," In VLDB, pp. 526–537, 2012.