論文 / 著書情報
Article / Book Information

Title	PM-DB: Partition-based Multi-instance Database System for Multicore Platforms
Authors	Fang Xi, Takeshi Mishima, Haruo Yokota
Citation	Proc. of ICEIS2015, , , pp. 128-138
Conference name	17th International Conference of Enterprise Information Systems (ICEIS 2015)
ISBN	978-989-758-096-3
Pub. date	2015, 4

PM-DB: Partition-based Multi-instance Database System for Multicore Platforms

Fang Xi[1], Takeshi Mishima[2] and Haruo Yokota[1]
[1]*Department of Computer Science, Tokyo Institute of Technology, Tokyo, Japan*
[2]*Software Innovation Center, NTT Japan, Tokyo, Japan*
xifang@de.cs.titech.ac.jp, mishima.takeshi@lab.ntt.co.jp, yokota@cs.titech.ac.jp

Keywords: DBMS, Multicore, Middleware, TPC-W.

Abstract: The continued evolution of modern hardware has brought several new challenges to database management systems (DBMSs). Multicore CPUs are now mainstream, while the future lies in massively parallel computing performed on many-core processors. However, because they were developed originally for single-core processors, DBMSs cannot take full advantage of the parallel computing that uses so many cores. Several components in the traditional database engines become new bottlenecks on multicore platforms. In this paper, we analyze the bottlenecks in existing database engines on a modern multicore platform using the mixed workload of the TPC-W benchmark and describe strategies for higher scalability and throughput for existing DBMSs on multicore platforms. First, we show how to overcome the limitations of the database engine by introducing a partition-based multi-instance database system on a single multicore platform without any modification of existing DBMSs. Second, we analyze the possibility of further improving the performance by optimizing the cache performance of concurrent queries. Implemented by middleware, our proposed PM-DB can avoid the challenging work of modifying existing database engines. Performance evaluation using the TPC-W benchmark revealed that our proposal can achieve at most 2.5 times higher throughput than the existing engine of PostgreSQL.

1 INTRODUCTION

In recent years, the technology of multicore processors has been improving dramatically. On the multicore platform, software cannot benefit from the increasing of clock speed anymore, but has to improve performance by exploiting thread-level parallelism. Thus, greater emphasis is placed on the parallelization of software than ever before.

Most database management systems (DBMSs) were designed in the 1980s, when only uniprocessors were available. 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. Moreover, traditional DBMSs, which are dedicated to improving database performance through I/O optimization, fail to utilize modern processor and memory resources efficiently. 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 (Cieslewicz and Ross, 2008). Therefore, performance analysis and optimization for existing DBMSs on multicore platforms has become an important research topic (Hardavellas et al., 2007), (Ailamaki et al., 1999).

Tremendous efforts have been invested in the efficient utilization of many processor cores for different database applications. There are proposals for overcoming the memory–CPU gap for complex sort and join operations for OLAP applications (Kim et al., 2009), (Ye et al., 2011), and there has been research on solving contention in lock and log functions for concurrent updates in OLTP applications (Johnson et al., 2009). Moreover, the development of multicore platforms has motivated some work to analyze how and when to employ sharing for concurrent queries rather than optimizing each query independently for different kinds of workloads. For example, a proposal to provide work sharing between concurrent queries (Psaroudakis et al., 2013), and a solution for improving processor cache performance for concurrent queries (Xi et al., 2014) have been proposed.

However, mixed workloads such as those modeled in the TPC-W benchmark (Menasce, 2002) are different from both typical OLAP and OLTP applications (Giannikis et al., 2012), (Salomie et al., 2011). 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.

In this paper, we have analyzed the performance of a mixed workload on a modern Intel E7 multicore platform, and proposed new middleware called PM-DB to improve the performance for database systems on multicore platforms by treating the multicore machine as an extremely low latency distributed system. PM-DB provides a single database image to the users while coordinating query executions across several shared-nothing database instances deployed on a single multicore machine. Moreover, PM-DB can manage the binding relationships between the database engines and the underlying processor cores to improve the throughput further by reducing processor cache misses. Because it requires no modifications to existing DBMSs or OSs, our PM-DB is simpler and more practical than most existing proposals.

Our research was motivated by a performance analysis of the TPC-W workload on a modern multicore platform. The experiment showed that the shared-memory function of existing engines is becoming the system bottleneck. We therefore proposed a middleware of PM-DB to maintain multiple database engines running in parallel on the multicore machine, rather than a single database engine, to reduce the shared-memory contention and provide high scalability. The main contribution of this paper is to demonstrate how a partition-based multi-instance database system performs on modern multicore platforms. By introducing the system architecture that was originally widely used by distributed database systems to a single multicore platform, the contention in a single database instance is solved and the computing capacity of the multicore platforms is more fully exploited. Experiments show our proposal can provide much better throughput and scalability on multicore platforms.

Moreover, the PM-DB can further reduce the contentions in cache levels by maintaining better cache locality for each core's private cache. As the processor–memory gap is becoming larger on modern multicore platforms, we analyzed the possibility of optimizing the processor cache performance for concurrent queries. Optimizing cache performance to achieve better overall performance is a challenge, and it often requires very careful tuning of the data

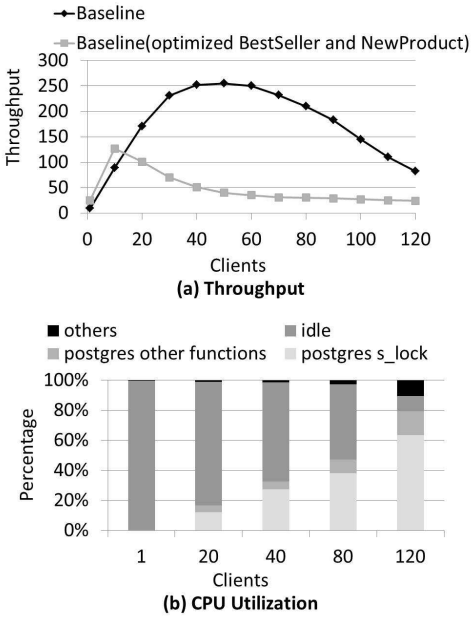

Figure 1: Scalability of TPC-W on a multicore platform.

structures and algorithms. However, we have proposed a solution to this problem that requires no changes to existing software implementations and we reduced the cache misses by improving the corunning strategies for better data sharing between concurrent queries in different processor cache levels. We pointed out that by assigning simple queries and complex queries to different cores, we could achieve much better cache performance and higher system throughput. Our detailed analysis of cache performance at different levels provides several insights on multicore processor cache optimization for mixed workloads.

The remainder of this paper is organized as follows: In Section 2, we describe our motivating experiments. In Section 3, we describe our proposed PM-DB in detail. In Section 4, we discuss the cache optimization solution for the basic PM-DB system. We conducted extensive experiments to show the efficiency of our proposal in different aspects and these are described in Section 5. Finally, Section 6 concludes the paper.

2 MOTIVATION

We analyzed the scalability of the mixed workload using the TPC-W benchmark with the Browsing workload mix on a modern Intel E7 multicore platform with 80 hardware contexts. A detailed description of the hardware platform and the workload can be found in Section 5. We evaluated the PostgreSQL DBMS

separately with the default query plan and the optimized query plan, in which we greatly reduced the response time for the "Best Seller" and "New Product" transactions, which are two relatively complex queries with high frequency of occurrence in the TPC-W benchmark. However, we must emphasize that our optimization is based on the execution of a single query as we usually do in traditional performance tuning. The throughput of these two Baseline systems while increasing the number of concurrent clients is shown in Figure 1 (a).

This figure shows that both systems encounter severe scaling problems. There are 80 hardware threads on the multicore platform, although the Baseline system could only scale to 40 concurrent clients. Moreover, the optimized Baseline system cannot achieve better throughput than the unoptimized system and it can only scale to 10 concurrent clients. We believe that the scaling problem is not caused by overload in the system, as the 80 hardware contexts are far beyond the needs of the software. The system must encounter some bottlenecks that waste CPU time or force the CPU to be idle. We further analyzed the CPU utilization for the optimized Baseline system using the performance monitoring tool Perf (de Melo, 2010), and the results are shown in Figure 1 (b). 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. For example, every page search in the memory buffer must access several critical sections: one to lock the hash bucket to prevent the page being moved to other buckets during the search, one to pin the page to prevent its eviction from the memory buffer, and finally one to request a latch on the page to prevent concurrent modification to this page. For hot pages (metadata and high-level index pages), the critical sections become the bottleneck, as many threads compete for them, even if they access the page in read mode (Johnson et al., 2009). Even though the "Best Seller" and "New Product" transactions have no update operation, they access a large number of pages when they perform their join operations. Moreover, these queries lock a large amount of memory-critical sections. With these queries running concurrently in the workload, the database becomes blocked from accessing the hot pages in the shared memory space. This problem was also observed by Salomie (Salomie et al., 2011) and Boy-Wickizer (Boyd-Wickizer et al., 2010). They reported similar bottlenecks for other database engines besides PostgreSQL or MySQL.

From these experimental results, 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. One solution to this problem is to rewrite existing database engines to fix all synchronization problems. The existing system SharedDB (Giannikis et al., 2012) takes this roadmap by creating a query plan for the whole workload, and the new engine merges the concurrent queries into one single query to reduce the concurrent access to the hot pages in the shared memory space. However, these kind of proposals require a complete rewrite of existing database engines. Considering the complexity of most commercial database engines, the rewriting will be a very challenging and time-consuming task. The other solution is to exploit 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 as the proposal of Multimed (Salomie et al., 2011). Salem et al. proposed introducing multidatabase instances on a single multicore platform to solve the single instance contention and evaluated their proposal as Multimed, which is a master–slave structure conventionally used in computer clusters. However, their proposal requires much memory space, as they hold a whole or partial copy of the database on each slave.

In this paper, we analyzed a different way to deploy the multi-instance architecture and introduced a partition-based multi-instance solution that was originally used in shared-nothing parallel database systems (Mehta and DeWitt, 1997). The proposed PM-DB has less contention in shared memory compared with the single instance database system since concurrent queries which compete for the shared memory critical sections are distributed to execute on several different instances. With less queries simultaneously accessing the critical sections in each single instance, the concurrent queries will not be blocked in the "s_lock" function in our PM-DB system. Our proposal requires much less memory space and avoids the complex work of maintaining consistency for master and slaves compared with the Multimed solution. Moreover, our solution can be implemented as middleware, thus avoiding the hard work of modifying existing database engines as required by the SharedDB proposal.

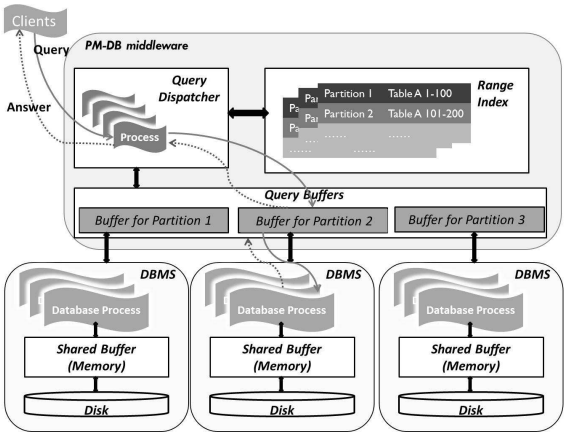

Figure 2: PM-DB system deployed on a single multicore platform.

3 THE PARTITIONED MULTI-INSTANCE SYSTEM

The multi-instance architecture was originally widely used in distributed systems and database clusters. However, in this paper, we are introducing this architecture into a multicore platform to overcome bottlenecks in existing database engines. We propose the PM-DB middleware to manage the database partition information in the different database instances, and to coordinate query execution between all of the underlying database instances. By distributing concurrent queries to different database instances, our proposal can ease the pressure on a single database engine.

3.1 Architecture Overview

The architecture of the PM-DB system deployed on a single multicore platform is shown in Figure 2. 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. The PM-DB offers a single system image to the clients, which do not require any knowledge about the partition information in the underlying database instances, and the only change to the client is that it communicates with the middleware instead of directly connecting to the database engine. The database instances then receive queries from the middleware and execute the queries in their dedicated query buffers.

The partitioned architecture has some advantages compared with the master–slave architecture used in the related research Multimed (Salomie et al., 2011). The biggest advantage is that the partitioned architecture saves memory space. In the master–slave architecture, some tables will be copied many times in the memory space, as each slave holds an independent copy in its local shared memory. This can easily cause the system to have insufficient memory. However, in the partitioned architecture, each database only holds part of the whole table and this saves a lot of memory space. On the other hand, in the master–slave architecture, the system must use snapshot isolation as a consistency criterion. Each update must be copied into all of the slaves. Queries are assigned a timestamp and dispatched to the appropriate slave that has all the updates committed up to that timestamp. This mechanism increases the complexity of the middleware. In the partitioned solution, this complex snapshot isolation mechanism is unnecessary, and we follow a simple two-phase commit protocol (Samaras et al., 1993) to manage update transactions in the multi-instances.

3.2 Query Processing in PM-DB

In our approach, the middleware manages multiple instances on a single multicore platform and each instance holds only part of the whole database. For a specific transaction, the required data are only stored in one specific database instance; therefore, in our middleware, we must dispatch the transactions to the instance that holds the required data; this is done by PM-DB's Query Dispatcher function. If we can partition the whole database cleanly between the database instances so that the data required by each transaction are all stored by a single instance, our middleware can simply dispatch each transaction to one specific instance. However, for some applications, it will be difficult to provide such a clean partition; that is, some transactions will require data that are stored in different instances. Therefore, our middleware must manage the transaction execution over multiple database instances. We offer a variety of optimized mechanisms to handle these cross-instance transactions, as any inefficient management of queries over multiple instances will cause large overheads and the loss of the advantage of the multi-instance architecture on multicore platforms.

First, for the update transactions that access multiple instances, we use 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. Second, even though we can achieve faster data transfer in multicore platforms, large amounts of data transmission will cause contention in the interconnections. For applications with frequent crossing-instance join operations, the data transfer between different instances will slow down the transactions and cause a system bottleneck. Therefore, our middleware provides data duplication to avoid intensive data transfers between different instances by redundantly storing the data for joins in different database instances. Moreover, the update queries accessing the redundant data will be propagated to all of the related database instances by our middleware, following the two-phase protocol according to the data duplication information. With this efficient management of concurrent queries over multidatabase instances, our middleware can achieve dramatic advantages compared with the Baseline system, with smaller overheads on the multicore platform.

The Query Dispatcher component in PM-DB uses these mechanisms to coordinate transaction execution between the different database instances. The Query Dispatcher process receives requests from the clients and finds the required data by parsing each query. Then the Query Dispatcher process refers to the database partition information stored in the Range Index and selects the appropriate instance to receive the query, and then queues the query in the database instance's Query Buffer. 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.

3.3 Database Partition

A horizontal partition is adopted for each table and a suitable field for a table is chosen as its partition key. Theoretically, any field of a table can serve as a partition key. In practice, however, our experiments have shown that the fields of the primary key of the table or even a subset of them can achieve good results. For multiple tables, it is better to choose common keys that can be used to partition most of the main (i.e., comparatively big and frequently accessed) tables. A good partition solution is to minimize the cross-partition update transactions and join operations.

For example, we partition the TPC-W database using the "Customer ID" and the "Item Subject". The "Customer ID" based partition can minimize the cross partition update operations. The order and order-line information related to a specific customer will be stored in a single database partition. Adding a new customer only requires access to a single partition. On the other hand, the "Item Subject" based partition can distribute the concurrent complex queries with join operations to different database instances. The two queries with join operations, which are the "Best Seller" and "New Product" transactions, have as their selection criteria the column Subject of the Item table. If we partition this table according to the column Subject, the "Best Seller" and "New Product" queries can be distributed into different database instances and each query can be answered using a single database instance. With less concurrent complex queries in each database engine, the shared memory contention can be reduced in each instance.

Different partition solutions may result in different performances. As the TPC-W benchmark monitors the typical application of an online book store, a large range of similar Web applications can follow the same partition method as we used in the TPC-W example. Moreover, we can find much help with achieving good partition solutions from the existing research on distributed database systems (Mehta and DeWitt, 1997), (DeWitt and Gray, 1992), (DeWitt et al., 1990), (Chen et al., 2013).

For the "Best Seller" and "New Product" queries, the Item table must join with the Author table and the Order-line table. Therefore, we must redundantly store part of the Author table and the Order-line table in all of the partitions to avoid cross-partition joins. Moreover, the "New Order" query, which updates the Order-line table, will accordingly be sent to all of the partitions.

Even though we must redundantly store some tables in our system, we still require much less memory space than does the existing Multimed (Salomie et al., 2011), with its master–slave architecture. In Multimed for the same TPC-W benchmark, in the memory-optimized setting, each slave requires 5.6 GB of memory for a database with 20 GB of data. If Multimed uses a four-instance system, it requires a total of 36.8 GB of memory, which is 84% more memory than is required for the original single-instance system. In our architecture, for a 10 GB database, the four-instance system requires 11 GB of total memory, which is only 10% more memory space. As a partitioned-based solution can avoid to redundantly store the big tables several times, our proposal needs less memory space compared with the Multimed for nearly all of the applications. Therefore, with the same memory size, our system has the advantage to handle much more database instances and provide the possibility to higher scalability and throughput.

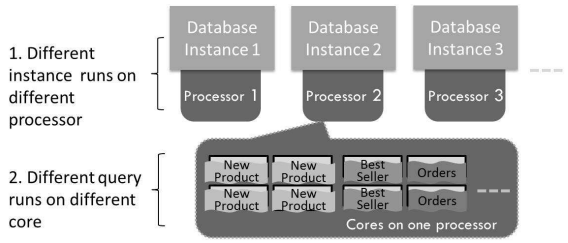

Figure 3: Cache-optimized query scheduling strategy.

4 CACHE OPTIMIZATION

Memory sizes have increased rapidly along with microprocessor performance, and memory can now play the role of the disk for many applications. However, memory bandwidth has not kept pace with the increasing number of cores and has become another bottleneck for higher performance. Therefore, the cache levels become critical for overcoming the processor–memory gap. Cache-efficient solutions for arranging the execution of concurrent queries on multicore platforms are very attractive, as better data sharing in cache levels can reduce the time-consuming memory access operations and improve the system performance. We analyzed how to improve cache performance by binding different database processes to different processor cores (Muneeswari and Shunmuganathan, 2011). The optimized core binding strategy is shown in Figure 3.

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.

With modern multicore processors, it is usual to provide two levels of cache for the private use of each processor core (private cache levels) in addition to a shared LLC. The simple queries which access several table lines, usually do an index based data search. 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. Therefore, we suggest to assign queries with different types to different cores. As shown in Figure 3, we assign the of “New Product” query, “Best Seller” query, “Orders” query to different processor cores.

We create separate queues for queries of different types, and assign a group of database processes to execute only the queries in a specific queue. We then bind the database processes that access the different query queues to different processor cores. The efficiency of these cache-efficient core binding strategies are analyzed in the experiment section in detail.

5 PERFORMANCE EVALUATION

In this section, we analyze the efficiency of our proposed PM-DB system on a modern Intel multicore platform. We implemented our middleware in the C language for the Linux operating system and the PostgreSQL DBMS, and compared the performance with an unchanged PostgreSQL system that serves as the Baseline system. Both systems used the TPC-W benchmark. Our experiments cover different data sets with different sizes and workloads with different scenarios. The detailed hardware, software settings and results are introduced in the following subsections.

5.1 Experimental Setup

We introduce the hardware, software and benchmark used in the following experiments in this subsection. The hardware of the DB server is a 40-core Intel Xeon system that serves as 80 virtual CPUs with Hyper-Threading. It has four processor sockets and one 10-core Intel Xeon E7-4860 processor per socket (Intel, 64). Each core runs at a clock speed of 2.27 GHz with a 64 KB L1 cache (32 KB data cache + 32 KB instruction cache) and a 256 KB L2 cache. All 10 cores of one processor share a 24 MB L3 cache. The server that we used to evaluate the benchmark has 32 GB of off-chip memory, and a 200 GB Hard Disk Drive. The client part is four machines and each machine has an Intel Xeon E5620 CPU and 24 GB of memory.

We use the TPC-W benchmark throughout this paper, which monitors the Web application of an online book store (Menasce, 2002). There are typical OLTP transactions of ordering new books and simple analysis transactions to collect detailed information

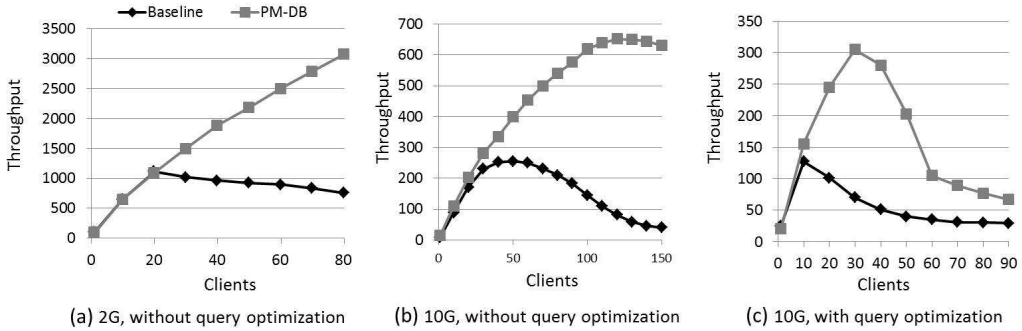

Figure 4: Throughput for the Baseline and the PM-DB systems.

about the latest new books and best-selling books on a specific topic (the "New Product" and "Best Seller" transactions). The benchmark specifies three workload mixes: the Browsing mix (5% updates), Shopping mix (20% updates) and Ordering mix (50% updates). The Ordering mix with heavy concurrent update operations stresses the log and I/O of the whole system before the system faces the s_lock contention. Therefore, we focus on the two scenarios of Browsing mix and Shopping mix. Our evaluation used data sets of 2 GB and 10 GB. The benchmark specifies both the application and the database level and we implemented only the database level to avoid the influence of the application layers. The system throughput and transaction response time are all calculated by the client machines.

5.2 Efficiency of the PM-DB

In this experiment, we set up a four-instance PM-DB system to demonstrate the effectiveness of our approach. The experiments cover two different data sets, and optimized and unoptimized Baseline systems. In the optimized system, we optimized the "Best Seller" and "New Products" queries as we have done in the motivation experiments. By creating appropriate indexes and tuning the query plan, we greatly reduced the execution time for both queries. However, we did not do any cache-efficient core binding for either the Baseline or the PM-DB systems. Figure 4 shows the throughput results with varying loads of the three different systems for the Browsing mix.

Our proposal shows dramatic advantages over all of the different settings. Figure 4 shows that our proposal can provide better scalability and higher throughput. In Figure 4 (a), the Baseline system scales to 20 concurrent clients, while our four-instance system can scale to 80 concurrent clients. Moreover, the PM-DB system can improve the

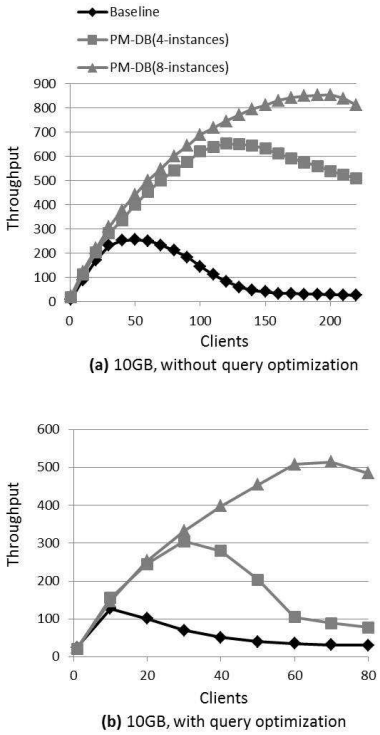

Figure 5: PM-DB systems with different database instances.

throughput by 1.8 times compared with the Baseline system. We can also observe the dramatic advantage of our proposal for the larger data set of 10 GB in Figure 4 (b) and Figure 4 (c).

As we observed in the motivation section, the traditional single-query optimization cannot provide better performance (comparing the Figure 4 (b) and Figure 4 (c)). When more queries run together concurrently, the interactions between the queries become severe and the optimization should consider all the concurrent queries. The traditional single-query-based optimization may negatively impact on other concurrent queries and slow down the whole system.

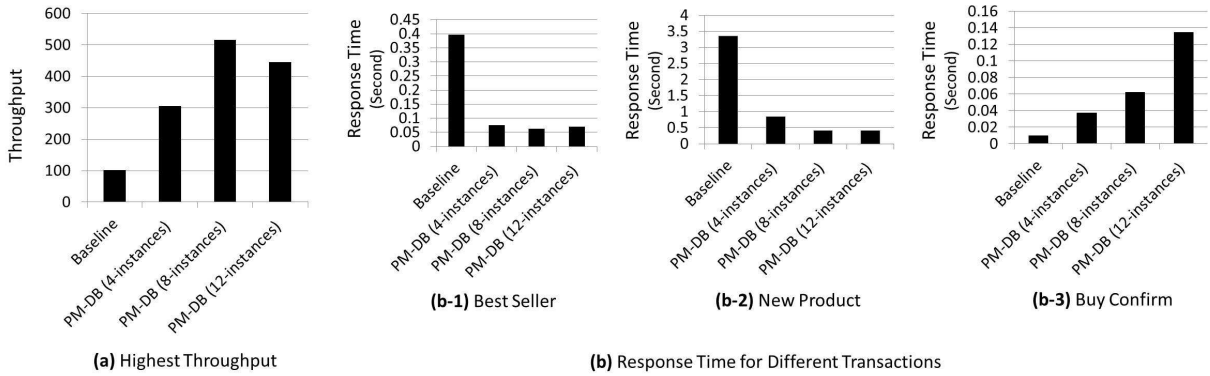

Figure 6: Highest throughputs and response times.

We do not limit the maximum number of connections in our PM-DB system as it is the same situation in the Baseline system. Therefore, even though our four-instance system can ease the pressure on the single database engine, the original contention in the shared memory of each instance will also affect the bottleneck when there are too many concurrent queries. This is why we observed a performance decrease in our PM-DB system when there were more than 40 concurrent clients in Figure 4 (c).

5.3 Increased Database Instances in PM-DB

The above results show that a four-instance PM-DB system achieved both higher scalability and throughput than the Baseline system, demonstrating the effectiveness of our proposal. However, in the early experiments, we observed that the four-instance system could not handle large numbers of clients for some settings. Therefore, in this subsection, we introduce eight-instance and 12-instance systems to explore the possibility of achieving better performance.

Figure 5 shows that more database instances can provide further improvement to the scalability and the throughput. The throughput of the PM-DB eight-instance systems are separately 3.3 times (Figure 5 (a)) and 3.5 times (Figure 5 (b)) as high as that of the Baseline system.

We then further analyzed the throughput of a 12-instance PM-DB system with the 10 GB data set and optimized queries (Figure 6 (a)). The 12-instance system does not achieve higher throughput than the eight-instance system. The "Best Seller" and the "New Product" transactions, which access a large amount of the table and index data, benefit greatly from our proposal (Figure 6 (b-1) and Figure 6 (b-2)). However, we observed that the queries that access multi-instances following the two-phase commit protocol take much more time in the 12-instance system and these queries slowed down the system. For example, the typical update transaction of "Buy Confirm", which adds a customer's new order to the database, must be copied to all of the database instances. With more database instances, the "Buy Confirm" transaction requires much more execution time as shown in Figure 6 (b-3).

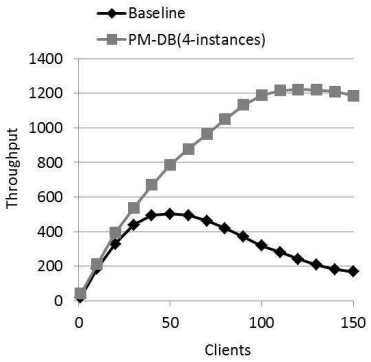

Figure 7: Performance for the scenario Shopping mix.

These experimental results indicate that increasing the number of database instances in the PM-DB system can further reduce the contention in each single instance and increase the possibility of achieving higher throughput and scalability. On the other hand, increasing the number of instances in the PM-DB system will increase the middleware overhead. Therefore, more instances do not necessarily equate to higher performance in our PM-DB system, and the users should find the most appropriate settings based on their specific applications.

5.4 Different Scenarios

We changed the scenario from the Browsing mix to the Shopping mix, which has more update operations. We evaluated the throughput of the Shopping mix

with the 10 GB data set and unoptimized transactions. Figure 7 shows that our proposal retains its advantage with increased update operations and the four-instance PM-DB system increased the throughput by 1.4 times over the Baseline system.

5.5 Cache Optimizations

In this section, we analyze the efficiency of our cache-efficient core binding strategies. The experiments are based on the four-instance PM-DB system. We bound the database processes of the four different database instances to four processors and each of those processed the queries from a specific database instance. With the 10 GB database without query optimization, we observed that the "New Product" transaction occurs very frequently and it has a longer response time than the other transactions. That is, the "New Product" transaction is relatively more complex than the other transactions. Therefore, we bound the "New Product" transaction and other transactions to be executed to different processor cores.

The appropriate core numbers for different transactions are strongly related to the specific application. We analyzed the core binding (10, 10) system in which the "New Product" transactions were bound to 10 virtual CPUs and another 10 virtual CPUs were used for the other transactions in each processor. We also analyzed two other core binding systems with increased core numbers assigned to the "New Product" transaction; these are the core binding (14, 6) and core binding (16, 4) systems.

The throughput with 60 concurrent clients is shown in Figure 8. We found that our cache-efficient core binding strategies further improved the throughput for the naive PM-DB system by 21% (with Core binding (14, 6)). The response times of the "New Product" and the "Best Seller" transactions are shown in Figure 9. We observed that the response time of the "Best Seller" transaction could be greatly reduced by the cache-efficient solution. This is because, by assigning this relatively simple query to run on different cores from the complex "New Product" query, much more reusable data could be cached in the private cache levels and the "Best Seller" query benefited greatly from this improved data caching.

The cache performances are shown in Figure 10, which was calculated using Oprofile (Levon and Elie, 2004). We evaluated the cache miss ratio (the percentage of cache misses in the total number of cache requests) for the different cache levels. As we expected, the cache miss ratio in the private cache of the L2 cache is reduced by 18.37%. By binding the database processes of different database instances to

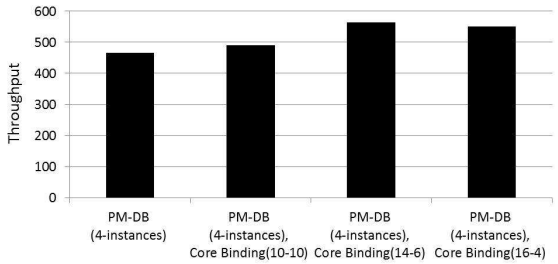

Figure 8: Efficiency of cache-efficient core binding strategies.

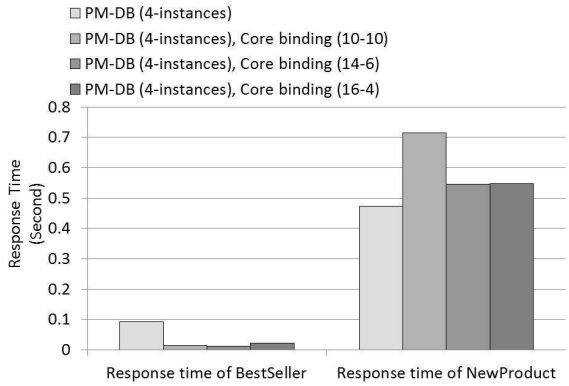

Figure 9: Response times for different cache-efficient core binding strategies.

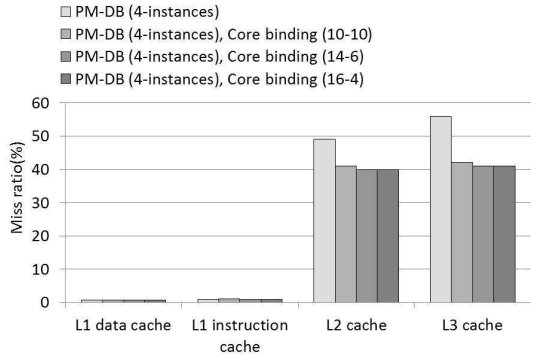

Figure 10: Cache performance for different cache-efficient core binding strategies.

different processor sockets, the miss ratio in the L3 cache was also reduced by 26.78%.

Comparing the different core binding systems, we found the response time of the "New Product" transaction becomes worse in the core binding (10, 10) system (Figure 9). This is because 10 cores are not sufficient for dealing with so many concurrent "New Product" transactions in the system. As we used more cores to process the "New Product" transactions in the core binding (14, 6) system, the response time of the "New Product" transaction was greatly reduced.

The proper setting of the core numbers for different queries can be found by performing load balancing on our system. However, we leave these studies to future work.

6 CONCLUSIONS

In this paper, we have proposed the middleware of PM-DB to improve the performance of database systems on modern multicore platforms. The strongest advantage of our proposal is that it can be implemented as middleware using existing OSs and DBMSs, so it is more practical than some other proposals, which require existing database engines to be rewritten. Moreover, the shared-nothing architecture used in PM-DB requires much less memory than the solutions using the master–slave architectures as Multimed. The PM-DB middleware can provide a partition-based multi-instance architecture on a single multicore platform to solve the contention in shared memory functions for existing database systems. The middleware can efficiently manage query execution over multiple instances by offering the two-phase commit protocol and partial data replication. By distributing the concurrent queries to different database instances, our proposal can ease the pressure on a single database instance and provide better utilization of modern multicore platforms. Moreover, PM-DB can further improve the performance of database applications through cache-efficient query scheduling on multisocket multicore platforms. The introduced experiences can provide a useful reference as optimizing a variety of database applications on modern and upcoming multicore platforms.

By setting up more database instances, the contention in the single database engine can be reduced and experiments using the typical mixed workload of TPC-W benchmark show that our proposal achieved at most 2.5 times higher throughput than the Baseline system. However, increasing the number of database instances will not always lead to higher system performance, as we observed the middleware overhead in synchronizing the transactions across different instances will also be increased by setting up more database instances. Moreover, the cache-efficient query dispatching for concurrent queries provided by the PM-DB can further improve the system throughput by 21% and result in 26.78% less L3 cache miss.

In future, we plan to introduce load-balancing solutions into our middleware, and provide a dynamic performance tuning function to automatically generate the proper settings for different applications. We believe not only the PostgreSQL, but also other database engines can benefit from our proposal and for another future work, we intend to verify the advantage of PM-DB with different DBMSs.

REFERENCES

Ailamaki, A., DeWitt, D. J., Hill, M. D., and Wood, D. A. (1999). DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277.

Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. (2010). An analysis of linux scalability to many cores.

Chen, S., Ng, A., and Greenfield, P. (2013). A performance evaluation of distributed database architectures. *Concurrency and Computation: Practice and Experience*, 25(11):1524–1546.

Cieslewicz, J. and Ross, K. A. (2008). Database optimizations for modern hardware. *Proceedings of the IEEE*, 96(5):863–878.

de Melo, A. C. (2010). The new linux perf tools. In *Slides from Linux Kongress*.

DeWitt, D. and Gray, J. (1992). Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98.

DeWitt, D. J., Ghandeharizadeh, S., Schneider, D. A., Bricker, A., Hsiao, H.-I., and Rasmussen, R. (1990). The gamma database machine project. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):44–62.

Giannikis, G., Alonso, G., and Kossmann, D. (2012). SharedDB: killing one thousand queries with one stone. In *VLDB*, pages 526–537.

Hardavellas, N., Pandis, I., Johnson, R., Mancheril, N. G., Ailamaki, A., and Falsafi, B. (2007). Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, pages 79–87.

Intel, I. (64). IA-32 architectures software developers manual volume 3b: System programming guide. *Part*, 1:2007.

Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., and Falsafi, B. (2009). Shore-MT: A scalable storage manager for the multicore era. In *EDBT*, pages 24–35.

Kim, C., Kaldewey, T., Lee, V. W., Sedlar, E., Nguyen, A. D., Satish, N., Chhugani, J., Blas, A. D., and Dubey, P. (2009). Sort vs. hash revisited: fast join implementation on modern multi-core cpus. In *VLDB*, pages 1378–1389.

Levon, J. and Elie, P. (2004). Oprofile: A system profiler for linux.

Mehta, M. and DeWitt, D. J. (1997). Data placement in shared-nothing parallel database systems. *The International Journal on Very Large Data Bases*, 6(1):53–72.

Menasce, D. (2002). TPC-W: A benchmark for e-commerce. *Internet Computing, IEEE*, 6(3):83–87.

Muneeswari, G. and Shunmuganathan, K. L. (2011). A novel hard-soft processor affinity scheduling for mul-

ticore architecture using multi agents. *European Journal of Scientific Research*, 55(3):419–429.

Psaroudakis, I., Athanassoulis, M., and Ailamaki, A. (2013). Sharing data and work across concurrent analytical queries. In *VLDB*, pages 637–648.

Salomie, T. I., Subasu, I. E., Giceva, J., and Alonso, G. (2011). Database engines on multicores, why parallelize when you can distribute? In *EuroSys*, pages 17–30.

Samaras, G., Britton, K., Citron, A., and Mohan, C. (1993). Two-phase commit optimizations and tradeoffs in the commercial environment. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 520–529. IEEE.

Xi, F., Mishima, T., and Yokota, H. (2014). CARIC-DA: Core affinity with a range index for cache-conscious data access in a multicore environment. In *DASFAA*, pages 282–296.

Ye, Y., Ross, K. A., and Vesdapunt, N. (2011). Scalable aggregation on multicore processors. In *DaMoN*, pages 1–9.