An Investigation of Performance Bottlenecks in a Main-Memory Database Management System

Xin Jia
xin@cs.brow.edu
Brown University

ABSTRACT

The rise of Internet-based applications means that database management systems (DBMSs) need to support a large number of concurrent clients issuing transaction requests. But the speed in which traditional DBMS are able to access data is insufficient because of the limited bandwidth of disk storage devices. Main memory database management systems (MMDBMS) [16] store data entirely in memory to overcome the high latency of these storage devices.

Furthermore, a DBMS running on a single-core processor system reaches the computation limitation to process multiple requests efficiently because of the contention overhead from many concurrent requests. The arrival of multi-core processors makes parallel computing possible in a single system. But although multi-core processors have more computational power, executing transactions in the correct order makes it difficult for a DBMS to take advantage of them.

In this paper, we explore the different types of bottlenecks found in MMDBMS when it is running on a multi-core system. Particularly, we use the H-Store MMDBMS as our test bed for this work. We identify three key issues, namely, excessive allocated objects, lock contentions, and recovery logging overhead as the main factors in preventing H-Store from scalability. We describe how to fix these issues, evaluate the performance of H-Store before and after these fixes, and show that we are able to improve throughput by 70%.

1. INTRODUCTION

The previous decade saw the emergence of separate, dedicated DBMS for online transaction processing (OLTP). OLTP applications focus on executing short-lived, small-footprint transactions with high throughput and strong consistency guarantees. Most of these transactions read or write a few records at a time, execute low latency without "user stalls", and require lightweight concurrency control mechanism. Most of the total size of OLTP databases is small and growing slowly. [1]

Traditional DBMSs are predicated on the idea that database's primary storage location is on a disk. This is because disks were more affordable than memory at that time. The limited data access speed of disk-oriented storage, however, is a bottleneck for many OLTP applications. Accessing data from main memory is several order of magnitude faster than from disk. Today, main memory with several Gbytes has become common and affordable. The increasing size of the memory at an affordable price makes it possible for many OLTP applications to store all of their data in main memory. A system that is designed for running entirely on main memory can eliminate the data access bottleneck[4].

As we enter the multi-core era, the future computational abilities of single-core processors are limited. Thus, the only way to improve throughput is to take advantage of new multi-core processors. The increasing computational power of multi-core processors provides one solution to this problem. Ideally by dividing an application across cores, the overall performance can improve in proportion to the number of cores. But, in modern database applications, this scalability depends on whether the workload can be easily divided in this manner. [2,3]

Based on the two features mentioned above, DBMSs using main memory architecture running on multi-core CPUs can execute modern transaction processing workloads efficiently. Applications developers expect DBMS to scale-up as more hardware resources are provided to it. But achieving this is difficult.

In this paper, we investigate why this is difficult. Our work focuses on the H-Store OLTP DBMS. [13] H-Store makes use of multi-core systems with abundant main memory to get significantly high transaction processing throughput than traditional DBMSs. For perfectly partitioned workloads, H-Store should be able to improve its throughput linearly as more CPU cores provided to it. As we show in the paper, however, this is not the case for H-Store running on system with more than eight cores.

We first begin in Section 2 by introducing H-Store's design architecture. In Section 3, we present an experiment that demonstrates H-Store's failure to scale up on processors with many cores. Then, we design experiments to investigate performance issue of H-Store. Following that, in Section 4, we describe the three main performance bottlenecks and our work to resolve them. At last, we conclude with a discussion of future work to improve H-Store.

2. H-STORE OVERVIEW

This section presents the overview of the H-Store MMDBMS. It is a parallel main memory relational DBMS that is designed to scale up on a cluster of shared-nothing nodes and optimized for OLTP applications. We introduce the design model of H-Store first and describe the system's architecture through its execution environment and internal threads. Then, we describe why H-Store uses the techniques including the Command Logging and group commit.

2.1 Design Model

A node in H-Store is a single physical computer system that hosts one or more execution sites. A site is the logical operational entity in the system; it is a single-threaded execution engine that manages some portion of the database. It consists of partitions and they are responsible for executing transactions. [13]

Every table for a database in H-Store is horizontally divided into disjoined partitions based on column keys. It is row-storage DBMS and the rows are separated across the partitions. All transactions are executed as stored procedures. A set of stored procedures consists of the application logic in H-Store. Each stored procedure has a unique name, several parameterized SQL commands and a run method that contains user-written Java code. Applications issue requests to an H-Store system to invoke the stored procedures as transactions. All of the stored procedures fetch data from partitions directly without accessing a disk.

Each transaction in H-Store is categorized as either a single-partition transaction or a multi-partition transaction (i.e., distributed transaction). Single-partition transactions only access data in one partition while distributed transactions access data on multiple partitions. Previous work [2,3] has shown how to choose the right partitioning scheme designed to allow most transactions to be single-partition transactions. And H-Store is optimized for single-partition transactions. A heavyweight concurrency control mechanism, however, is necessary for distributed transactions to ensure correctness.

2.2 Architecture

H-Store is written in two programming languages: C++ and Java. The Execution Engines (EEs) of H-Store are in charge of the low-level data manipulation operations in the database. Every query executing in H-Store has to invoke the codes in EE to access or update the data. Each Partition Execution (PE) thread is a single thread assigned to a separate core. H-Store guarantees that a PE thread is pinned to a single core and that no other thread is allowed to run on that core. The EE is written in C++ as this allows for fine-grained control of memory. The front end of the system, that contains the core functional features, is written in Java. In order to call the C++ code functions of the EE from Java code level, it uses Java Native Interface (JNI).

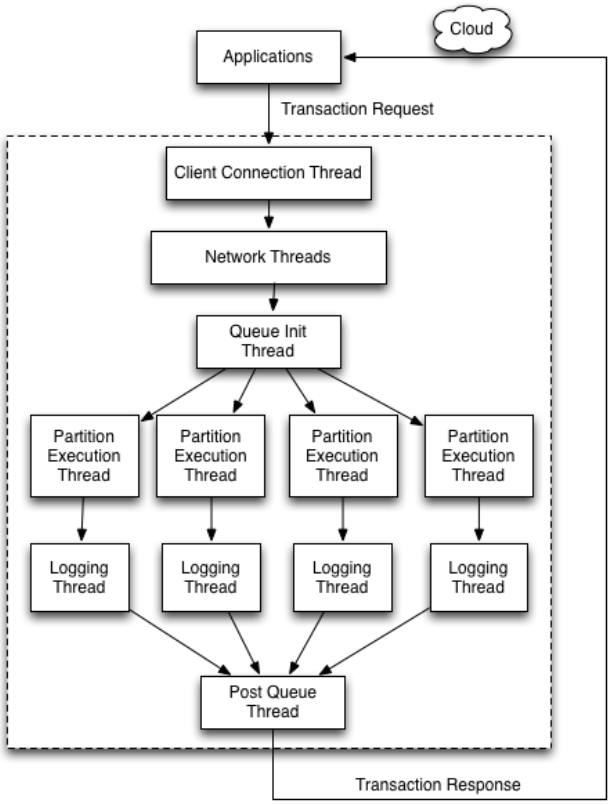

Figure 1: Overview of system threads

Threads in H-Store are divided into two groups: (a) system-networking threads, (b) transactions management threads. These threads are illustrated in Figure 1.

When the application issues a transaction request to H-Store, the incoming request is processed by a Client Acceptor thread. This thread accepts connection from applications and forward requests to internal network threads. The transaction Queue Initiator thread will process the request passed from networks threads and determine which partition to invoke the stored procedure on and which partitions the transaction will need to access. Then, it initiates incoming transactions and passes them to different PE threads. Each PE thread takes transaction requests from its work queue, executes them, and sends results to a logging thread that is assigned to it. After that, each logging thread sends execution results to the Post Queue thread. Later this Post Queue thread will send results back to client.

2.3 Command Logging & Group Commit

To ensure transactions are durable, H-Store uses the Command Logging [12] technique as its log manager. Command logging keeps the invocations of transactions as they are executed (i.e., the name of the stored procedure and the transaction input parameters), not the consequences of them. By recording only the invocations, the amount of data that the system writes to disk per transaction is small, limiting the impact the disk I/O will have on performance

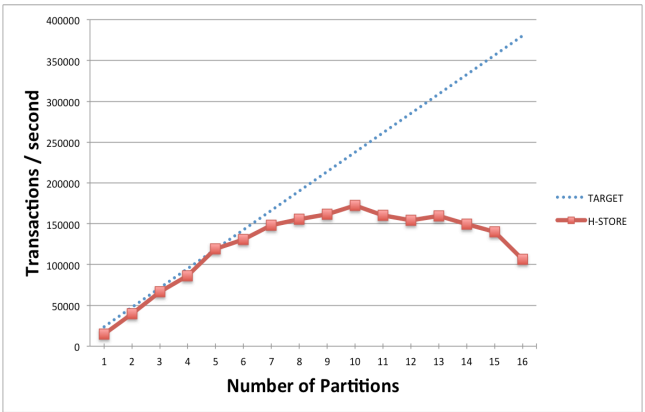

Figure 2: Throughput of H-Store

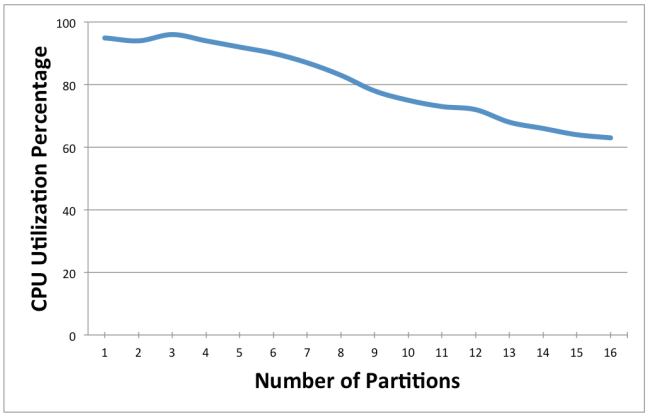

Figure 3: Average CPU utilization for execution cores

of the system. Although Command log will require addition processing to replay to the most recent state of the database, it is fast and lightweight compared to Write-Ahead log, which contains all the changed made to the records. This is the reason why H-Store chooses to use the Command Logging technique, rather than traditional Write-Ahead Logging (WAL). [21]

Instead of flushing the buffered log entries to disk per transaction, H-Store commits incoming transactions in a batch whose invocations are buffered in the system. Group commit technique removes the disk writing I/O bottleneck. H-Store does a group commit when the buffer is full of transaction invocations and flushes the buffer to disk permanently. [20]

3. INVESTIGATION EXPERIMENTS

Since most transactions in H-Store are single partitioned, the system should be able to scale-up easily as more cores are added. This is because the PE threads will not block each other. It is designed to execute single-partitioned transactions efficiently. But as we now demonstrate, this is not the case for H-Store running on a large number of cores.

In this section, we describe how we investigate the bottleneck shows the performance problem. We first demonstrate its scalability issues by running experiments. Then, we present the CPU utilization rate when H-Store is running with different partition cores. Following that, we make a new Client Imitator to validate our hypothesis, introduce a profiling tool, and other debug methods to diagnose the system. At last, we show the observation results and point-out the performance problems.

3.1 Scalability Issues

The Voter benchmark models a talent-show voting application. [22] It is based on the software system used for the Japanese version of the American Idol television show. Because the benchmark consists of short-lived, single-partition transactions, it is the ideal application to use on a MMDBMS.

We run the Voter benchmark on a single node with an increasing number of partitions ranging from 1 to 16. For each trial, we let the system "warm-up" for 90 seconds and then the throughput is measured for 60 seconds. We execute the benchmark three times in total per trial and report the average throughput of these trials. The final throughput measurement is the average number of transactions completed in a trial run divided by the total time (excluding the warm-up period).

Transaction requests are submitted from up to 16 simulated client terminals running on the same nodes. Each client submits transactions at a rate of 20,000 transactions/second without blocking until the system has 500,000 transactions. Using multiple clients at such a high transaction submission rate ensures that the execution engines' workload queues are never empty.

The results in Figure 2 show that the system does not scale linearly with more than 8 cores. In fact, the performance actually gets worse. There are problems that prevent the system throughput from linearly increasing with more partition execution cores.

3.2 Utilization of multi-core CPU

When running the Voter benchmark on H-Store, we notice that the CPU utilization rates for the PE cores decreases with using more cores. H-Store should be able to fully use the PE cores provided by the CPU and the utilization rate of each execution core is expected to be up to 100%. This is because no single-partitioned transaction will be blocked because of anther single-partitioned transaction. All transactions are single-partitioned and each PE core is single-threaded. But, this is not the case when the Voter benchmark is running on H-Store with more cores.

We collect the CPU utilization rate for each execution core every five seconds during executing the Voter benchmark. The result in Figure 3 shows that the average CPU utilization rate of execution cores decreases with more cores. H-Store ensures that a PE thread is running on a single core without any others threads competing for it, so each PE thread should fully use the core that it is pinned to. Limited by the reduced utilization of the multi-core CPU,

H-Store does not take the advantage of all the computational power of the multi-core processors, and thus, the overall throughput is below the expected target line in Figure 2.

Given these results, we need to investigate the causes of the reduced CPU utilization. At first we suspect that the overhead of the transmitting transaction requests over the network is causing the execution engines to be idle because of workload shortages. To explore this assumption, we add more clients to issue transaction requests to the system and improve the clients' transactions issuing rate. The PE cores are still not fully utilized, which leads us to another hypothesis that the system is network I/O bound. This bound can limit enough incoming transaction requests from clients.

3.3 Embedded Clients

In order to inspect and verify our hypothesis, we create a new special thread directly inside of the DBMS to generate transaction requests continuously without involving the network. This eliminates any possible overhead due to the network. So, if our hypothesis is true, the Partition Execution cores should be fully used with this new Client Imitator.

We run the system with the Client Imitator again and the trends of the new average CPU utilization for execution cores are the same as in Figure 3. The PE cores fail to be fully used no matter how many transactions are generated to the system. Thus, networking I/O is not the bottleneck, and the issue must be in the internal transaction management portion of the system.

3.4 JVM Profiling & Debug Tools

An in-depth study of the performance bottleneck of a large DBMS requires information about how the system is using the hardware advantages. We use a variety of tools to collection execution information of H-Store, including JVM profiling tool, JVM debug options, and H-Store debug options.

Since H-Store uses Java for its transaction management and coordinal subsystem, we choose a JVM profiling tool to analyze the execution details of H-Store. The Java profiling tool we use is JProfiler [11], which provides the ability to view the information about CPU utilization, memory usage, threads status, and locking graph. This tool also records the number of objects the system has allocated, the garbage collection (GC) activity, and the hot spots of methods and classes.

The JVM itself also provides debug options to help application developers to understand its activity. In JVM, there are daemon threads called Garbage Collectors that attempt to arrange the heap to ensure that the subsequent space requests are successful. Unreferenced objects in heap are deleted in the GC's work cycle and these cycles can lead to unexpected pauses in the execution of application code. The pause time of GC indicates how long it pauses the system from running other applications on it. The GC

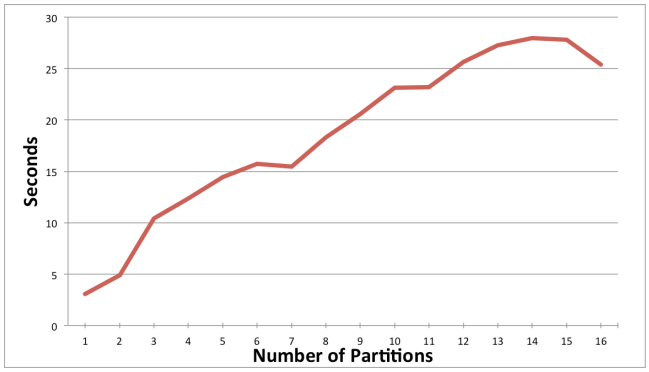

Figure 4: Summation of GC pause time

liberates Java programmers from the memory management work and helps them focus more on useful logic instead. But for a system with high performance goals, programmers have to take garbage collection into consideration to achieve the desired performance.

H-Store also provides additional debug options to output status details for Partition Executors. The system collects information about how much time it spends on execution transactions, sitting idle waiting for new work or processing utility work. Moreover, it reports the size of each transaction work queue, the average execution time of the transactions, and how the incoming transactions are distributed to different Partition Executors. Ideally, we want transactions to be evenly distributed among partitions.

3.5 Observation Results

Using JProfiler, we see that the total memory size of the system grows quickly when we are running the Voter benchmark. In order to check whether they are expected or not, we create a stored procedure that does not execute any query or perform any comprehensive work. Without any updates to the tables in H-Store, its memory usage should not increase. Now we run the Voter benchmark again with the same settings. But in fact, the amount of memory that the system uses arises and drops greatly as much as before. The above observations lead us to focus on memory related issues.

This time we invoke the new stored procedure at a low rate to the system and record the allocated objects in the Java Heap in H-Store using JProfiler. With fewer incoming transactions that do not update the database, the quantity of allocated objects in the system should be stable. We see however that many objects inside of the single-partitioned transaction object are increasing quickly when a transaction comes into the system.

Meanwhile, we also find the other CPU cores, rather than the PE cores, are doing work that H-Store does not specify them to do. CPU utilization of non-PE cores is unexpected high. As the JVM uses a garbage collector to perform most of its work concurrently (i.e., while H-Store is running) to reduce the pause time of GC, we realize that the activities on the other cores are related to garbage collection. By enabling the GC debug output, we are able to record the

pause time in the activity of the GC and present the summation of pause times in Figure 4. The GC pause time are recorded while executing the Voter benchmark for 150 seconds on H-Store. This figure shows that the GC pause time increases as more cores are used.

Because each PE thread is a single thread that do not share their allocated objects, H-Store will allocates more objects with increasing the number of cores. This means that more objects are created as more partitions are added to the system.

In order to ensure correctness, the GC has to pause all the threads in the system, even though these threads are all single threads that do not need to be blocked, when it reclaims memory. The overhead of GC is not significant for a small number of objects in the heap, but it is serious with more objects when the heap becomes corresponding larger. This is the reason why the garbage collection pause time increases with more partitions.

Moreover, if a thread is in the middle of a JNI call, it takes longer for the GC to pause the whole application running on the JVM. H-Store requires using JNI to call C++ functions to read, insert, or update the data in the database. This is a key part of the system, and we cannot make changes of this main architecture.

4. IMPROVEMENTS

In this section, we discuss our work to resolve the performance scalability issues described above. We first start from making changes related to reducing the number of allocated objects, followed by the fixes of removing lock contentions in the H-Store system. Then we examine the recovery logging component and optimize it. For each kind of these three fixes, we demonstrate the performance of H-Store compared with the original H-Store by executing the Voter benchmark.

4.1 Reducing Object Allocations

As shown in Section 3.5, we know that the GC has a significant impact on H-Store's overall performance. In order to prevent the GC operation from invoked frequently, we can limit the number of objects allocated in H-Store.

H-Store uses the object pools that maintain a set of reusable objects to avoid needing to allocate and destroy them at run time. The profiling tool, however, shows that there are still a large number of objects continuously created and destroyed while H-Store is running. In order to find the reason, we use our Embedded Client thread to issue transaction requests at a low rate without inserting tuples to the database and observe the memory usage status of the system. Based on this evaluation, we see that several kinds of objects are unnecessarily allocated and they are generally categorized into three parts:

(1) Distributed transaction-related data structures.
(2) Unnecessary object arrays.
(3) Internal per-partition transaction state objects.

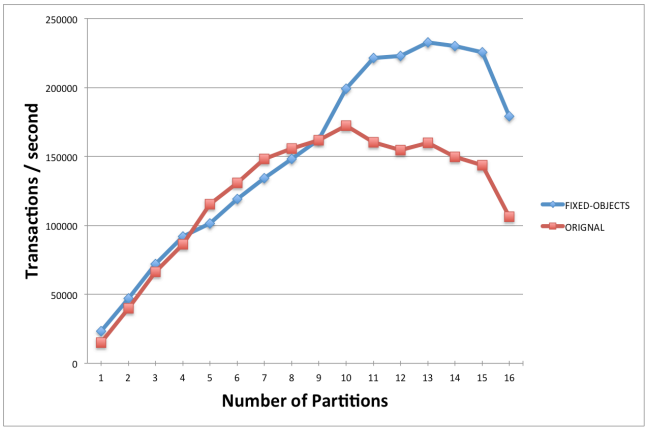

Figure 5: Performance with reduced objects

For the first group, we find a lot of allocated objects that are only needed by distributed transactions, like read and write sets for tables. Because most transactions in OLTP applications are single-partitioned, these objects allocated for distributed transactions are not helpful. We moved those objects into a special container that is allocated for creating distributed transactions.

For the second group, we find some allocated array of objects that can be replaced by a single variable. For example, the cached query batch planner in H-Store is guaranteed not shared across partitions, so there is no need to create a new array for every SQL statement batch. This change eliminates the number of total objects allocated in the system.

For the last group, we fix it by allocating objects as late as possible until the Partition Executing thread has to use them. This optimization shortens the lifetime of allocated objects in the system, and thus, it reduces the frequency in which the GC is invoked.

After minimizing the number of allocated objects in H-Store, we run the Voter benchmark again and show the new performance compared with the original H-Store. In Figure 5, although the trends are same, the throughput of the fixed version of H-Store is higher than the original version. The performance of both versions of H-Store drops off at 13 partitions.

4.2 Reducing Lock Contentions

If the system is executing only single-partition transactions, the PE threads should not conflict with each other. But from the profiling tool, we see several PE threads that are blocked on each other waiting for network resources. This is because that different PE threads try to put client response message to Post Queue thread at the same time. Then, we use the logging threads to resolve the lock contention between PE threads with the Post Queue thread. Each PE thread passes client responses to its logging thread directly, instead of competing with other execution threads for the shared network I/O resource. This allows the PE threads to go back and process more incoming transactions

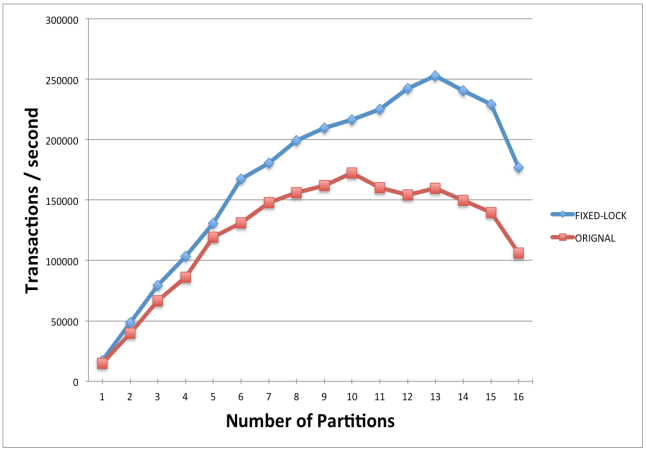

Figure 6: Performance with reduced lock contentions

without blocking. Then the logging thread puts those client responses to the Post Queue thread.

In H-Store, there are some locks that are needed (e.g., locks that are useful to add transaction requests to queues in the system). Thus, we cannot remove them entirely but we can reduce the amount of time that the locks are held. By using JProfiler, we gather information about how long each lock lasts and what threads are waiting to acquire that lock. We find several places where we can reduce the critical sections between the PE threads and the transaction Queue Initiator thread. We shorten the time that the locks are held and remove the acquiring nested locks inside of the PE threads. There are also some places that the transaction queue in the PE threads try to acquire the lock it holds already and we optimize this case.

With all these fixes of lock contention and objects pooling, we run the Voter benchmark on the fixed H-Store, compare it with the original H-Store, and demonstrate the results in Figure 6. This time our fixed H-Store has a higher throughput than the original version of H-Store, no matter how many partitions are used. The critical sections appear more times when there are a large number of PE threads. This is why our fixed H-Store has a bigger throughput increase with a large number of cores.

4.3 Recovery Logging

Using JProfiler again, we find a large number of objects in memory that are related to Command Log entries. The number of allocated objects for the buffer of the Command Logger is too big. These objects introduce the JVM to invoke GC many times in a short time interval. Then, the GC pauses PE threads, reducing the overall system throughput. But we find a mistake where the system calculates the number of log entries to use per partition. H-Store commits early before all the allocated Command Log entries are used. We fix this problem by allocating fewer log entries for each partition and let the system commits them in a batch when the buffer of the Command Log entries is fully used.

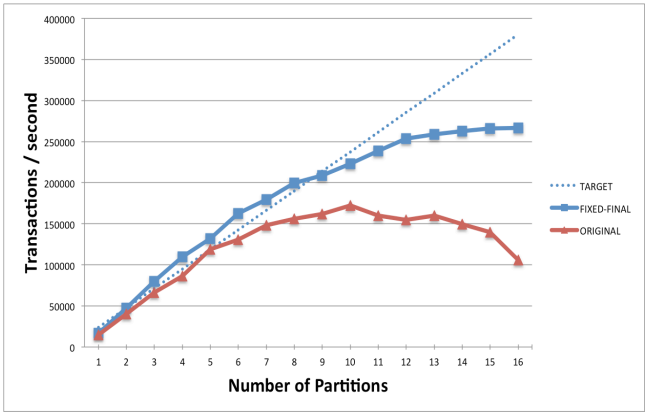

Figure 7: Final performance comparison

Instead of allocating and destroying the Command Log entries objects at run time, we pre-allocate a number of Command Log entries when starting the system and reuse them after H-Store commits them to log in disk. We switch the maximized number of log entries for each partition to a small number. This is because that we want to keep fewer objects in the heap. In order to get the best performance of the system, future work may focus on how to determine the number of log entries to be pre-allocated.

We run the Voter benchmark on H-Store again using these three major fixes from 4.1, 4.2, and 4.3. Then, we demonstrate the total performance improvement compared with the original version of H-Store in Figure 7.

Our fixed version of H-Store improves throughput by 70% over the original non-optimized version. Its performance linearly increases with up to 13 execution cores. The throughput of H-Store, however, is still not able to speed up any more and it stays stable.

5. FUTURE WORK

Figure 7 shows that the H-Store system still has issues preventing it from scaling up linearly when it is running on more than 14 cores. The main bottleneck is still the same. The GC is invoked more frequently with more cores. Future investigations will focus on the tuning of the garbage collector of the JVM. We will also explore using different garbage collectors provided by JVM and tune their parameters to achieve better performance. We will investigate the effects of calling C++ codes through JNI to garbage collection.

More optimization of the system will focus on its buffer pool manager. For the worst case, if we find the next bottleneck to be the garbage collector only, we will re-write the front part of the system in C++ instead.

6. RELATED WORK

There has been an extensive amount of work on diagnosing DBMSs' performance bottleneck. The Shore-MT project [6] studies the scalability of the database storage managers for the multi-core machine. It removes the internal bottlenecks of DBMS and has a higher absolute throughput

than its peers. DORA [17] is a system that decomposes a transaction to smaller actions and assigns actions to threads based on which data each action is about to access. But the scenario changes if the DBMSs put all data in main memory and many different issues come out.

7. CONCLUSIONS

Main memory database management systems can greatly improve performance over traditional disk-oriented DBMSs for OLTP applications. As multi-core processors become more prevalent, architectures need to be re-examined in order to fully utilize them. There are, however, various issues preventing them from scaling up linearly. H-Store is one such MMDBMS designed for multi-core processors, but we find its performance degrades when it is running on a large number of cores. After in-depth investigation, we remove major internal bottlenecks related to excessive allocated objects in memory, lock contentions, and recovery logging. Our fixed version of H-Store improves throughput by 70% from its original version. The approach we attempt to find these performance issues in H-Store can also be applied to future investigations.

8. REFERENCES

[1] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In VLDB, pages 1150–1160, 2007.

[2] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In SIGMOD '12: Proceedings of the 2012. international conference on Management of Data, pages 61–72, 2012.

[3] Carlo Curino, Evan Jones, Yang Zhang, Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. In VLDB '10.

[4] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP through the looking glass, and what we found there," in SIGMOD '08.

[5] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a High-Performance, Distributed Main Memory Transaction Processing System," Proc. VLDB Endow., vol. 1, iss. 2, pp. 1496-1499, 2008.

[6] R. Johhson, I. Pandis, N. Hardavellas, A. Ailamaki and B. Falsafi. In EDBT '09: Shore-MT: A Scalable Storage Manager for the Multicore Era.

[7] E. P. C. Jones, D. J. Abadi, and S. Madden, "Low Overhead Concurrency Control for Partitioned Main Memory Databases," in SIGMOD '10.

[8] J. D. Davis, J. Laudon and K. Olukotun. "Maximizing CMP Throughput with Mediocre Cores". In Proc. PACT, 2005.

[9] Virtual Machine Garbage Collection Tuning: http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#generations.performance

[10] E. Bugnion, S. Devine, and M. Rosenblum. "Disco: running commodity operating systems on scalable multiprocessors." In Proc. SOSP, 1997.

[11] JProfiler: http://www.ej-technologies.com/products/jprofiler/

[12] Malviya, Nirmesh and Weisberg, Ariel and Madden, Samuel and Stonebraker, Michael. "Recovery Algorithms for In-memory OLTP Databases" In Submission, 2013.

[13] H-Store: A Next Generation OLTP DBMS. http://hstore.cs.brown.edu

[14] J. Gray. "Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King." Gong Show Presentation at CIDR, 2007.

[15] VoltDB. http://www.voltdb.com.

[16] DeWitt, David J and Katz, Randy H and Olken, Frank and Shapiro, Leonard D and Stonebraker, Michael R and Wood, David. In SIGMOD 1984: "Implementation techniques for main memory database systems".

[17] Pandis, Ippokratis and Johnson, Ryan and Hardavellas, Nikos and Ailamaki, Anastasia. In VLDB 10:"Data-oriented transaction execution".

[18] Heytens, M. and Listgarten, S. and Neimat, M-A. and Wilkinson, K. In heytens95: "Smallbase: A Main-Memory DBMS for High-Performance Applications".

[19] Whitney, Arthur and Shasha, Dennis and Apter, Stevan. In HPTS 1997: "High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++".

[20] Helland, Pat and Sammer, Harald and Lyon, Jim and Carr, Richard and Garrett, Phil and Reuter, Andreas. In HPTS 1989: "Group Commit Timers and High Volume Transaction Systems".

[21] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, Peter Schwarz. In TODS 1992: "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging".

[22] Voter benchmark: https://github.com/VoltDB/voltdb/tree/master/examples/voter