Project description - second semester

Deterministic highly-scalable concurrency control within database systems

1 ABSTRACT

Prior research on deterministic database systems has proposed pre-determining a serializable schedule of transactions prior to their execution. The pre-determined schedule is represented via an explicit dependency graph, which determines the order in which transactions must execute and effectively obviates the necessity for explicit locking.

Over the course of last semester’s CPSC 490, I have shown that building a dependency graph by considering batches of transactions rather than single transactions independently from one another does lead to much more efficient and parallelizable schedule. Although the theoretical results from simulations are promising, they do not measure the overheads present within a read database system. This semester I have built a real database system to investigate the interplay between the overhead of handling batches and the improvement that follows from more parallelizable schedules. In particular, I have focused on creating a system that minimizes the overhead of batch creation and allows for multiple batch schedules to be created at parallel.

The implementational part of the work has focused on decreasing the contention resulting from communication among threads and a large number of threads processing batches in parallel. While the system has not been fully profiled as of the time of writing of this report, it is very possible that the throughputs obtained are at least as good as those of the traditional concurrency control systems and potentially even better. Current work on the project focuses on memory management and investigation into efficient garbage collection for the system. The full discussion of overheads and the solutions employed to decrease them will follow in the appropriate sections. The system has been written in around 10000 lines of C++ and is under active development. This report will describe the general architecture of the system, the results I have currently obtained and the future work.
2 System’s architecture.

The general architecture of the system may be summarized using the following short overview.

![Figure 2.1: Simplified overview of the system’s architecture. The arrows indicate the communication between parts of the system.](image)

Please note that the only threads within the system are the worker threads within the thread managers and the singular thread that interacts with the supervisor. All of the other elements of the system represent shared data structures used for synchronization among respective systems. The overview omits additional threads used for measurements and experiments.

2.1 The scheduling system.

The scheduling system’s primary responsibility is the global schedule creation and the batch schedule creation. Moreover, it communicates with the execution system and does the assignment of transactions to execution threads for processing. It is important to mention that the scheduling system preserves the ordering of transactions on the batch level. That is, all transactions within batch $i$ will be scheduled before any transaction from batch $i+1$. The work-cycle of any one of the worker threads within the scheduling system may be described in pseudo-code as:

```java
void start_scheduling() {
    while (true) {
        receive_input();  // (1)
        create_batch_schedule();  // (2)
        merge_batch_schedule_into_global_schedule();  // (3)
    }
}
```
Notice that step (2) is the only one that is thread-local. Hence, the optimization work outside of the scheduling algorithm must focus on efficient synchronization of steps (1) and (3).

### 2.1.1 Summary of Synchronization Techniques

To assure that step (1) is efficient I reduce the interaction between threads and limit it to single-consumer single-producer queues. The general scheme may be described as:

```c
Input receive_input() {
    try_lock_global_input_queue();

    if (lock_granted) {
        // if lock has been granted, give all of the queues their respective input.
        for (Thread t : SchedulingThreads) {
            if (t has no input enqueued) {
                t->enqueue_input(global_input_queue.get_input());
            }
        }
    }

    // lock not granted. This means a different thread is enqueing
    // input for all of the threads that have no input enqueued.
    while (current_thread->input.empty()) {}

    return current_thread->input.get_head();
}
```

The basic idea within the code is to reduce contention on the lock shared among all of the scheduler threads. This is done by allowing each scheduling thread to attempt to gain access to the lock exactly once. In case of success, the thread assigns input to all of the scheduling threads. This is fast since there is an approximately constant number of threads and a dequeue from an input queue is a very fast operation. In case of failure to obtain the lock, a thread must simply wait for input to be given to it by a different thread. As before, this busy-wait phase is very short.

The above approach allows scheduling threads to spend up to 90% of the time within the batch creation logic for batch sizes of 100 and more. While merging of a batch schedule into the global schedule is more complex, the basic idea applies there as well: We allow a single thread to handle the requests for access to a shared resource. Notice that this solution works

---

1 The process has been discussed in more detail in the work from former semester. Please refer to the end-of-semester report for more details.
because a single thread may easily handle the workload imposed on it by other threads, which follows from the fact that the critical section is very short.

2.1.2 SUMMARY OF BATCH SCHEDULE CREATION.

The batch schedule creation has been discussed at length within the write-up of the former semester’s work. Here I simply review the algorithm and give a high-level description of it. The implementational details will not be discussed as the basic ideas follow former work.

Let us consider a set of transactions. To schedule said transactions I would like to reorder them so that I may execute as many transactions in parallel as possible. We have shown in the past semester’s work that one approach is to iteratively find maximal packings within the non-scheduled transactions and use them to create a batch schedule. A packing is defined as a set of transactions that have non-conflicting read and write sets. A maximal packing is one that contains as many transactions as possible.

It is relatively easy to see that finding a maximal packing within a set of transactions is equivalent to solving a max-packing problem on the set of records within read and write sets. Since this is an NP-complete problem in generality and the set of existing records may be large, I must use an approximation. The simple linear-time approximation used within our system may be summarized as:

```c
get_transaction_packing (unordered_set<Transaction> T) {
    packing P = {}
    while (there exist transactions not colliding with members of packing) {
        Find transaction t within T that
        a) minimizes f(t)
        b) does not collide with txns in the packing
        c) Is not already present in the packing
        Add t to packing
    }
    return packing
}
```

Having a set of packings that covers the initial set of transactions one may create a batch schedule by simply adding dependency-edges between any conflicting transactions across packings. This process has been discussed in detail in former work and one should refer to that work for details.

It is important to keep in mind that batch-schedule creation is by far the most time-consuming part of the scheduler thread’s work-cycle and that it is performed off-line. All memory used is thread-local.

---

2 Nowadays, many machines have tens or hundreds of physical CPU cores. This motivates the need for parallelizable workloads and is the reason why I choose to attempt to find ordering which are more parallelizable.
2.2 THE EXECUTING SYSTEM.

The executing system’s primary responsibility is to follow the global schedule created by the scheduling system and execute transactions without introducing inconsistencies or data corruption. The work-cycle of any execution worker thread may be simply summarized as:

```c
void start_executing() {
    while (true) {
        receive_input_batch(); (1)
        execute_input_batch(); (2)
    }
}
```

Step (1) within the work-cycle is simpler than step (1) within the work-cycle of a scheduling thread, since a scheduling thread enqueues transactions directly into the local input queues of executing threads. Hence, receiving input by an executing thread is as simple as dequeuing from a single-produce, single-consumer queue. Almost all of the complexity of an execution thread is hidden within step (2).

2.2.1 EXECUTING A BATCH OF TRANSACTIONS.

As mentioned before, scheduling threads divide transactions from every batch across executing threads. Each executing thread considers all such transactions together within a single iteration of `execute_input_batch`. A direct consequence of this fact is that an execution thread will not go on to execute transactions from batch \( i \) until all of the transactions from batch \( i - 1 \) that were "assigned" to it have executed. To assure that this system is efficient, I use the following algorithm:

```c
void execute_input_batch() {
    while (not all owned transactions in current input batch executed) {
        execute_transactions_in_pending_list();

        if (execute_next_transaction) {
            execute_blocking_transactions();

            // transaction is not yet ready
            push_to_pending_list();
        }
    }

    while (pending_list_not_empty()) {
        execute_transactions_in_pending_list();
    }
}
```
The execution cycle contains two important subtleties:

1. The cycle operates on two distinct lists of transactions. The list of transactions within a batch that are owned by the thread and the list of pending transactions. A transaction is considered to be pending if a thread attempted to execute it and failed.

2. When the execution thread fails to execute a transaction, it attempts to execute "blocking transactions." That is, the execution thread attempts to execute any transaction that is the oldest precursor of the current transaction in the schedule. We refer to this behavior as "transaction stealing."

Notice that because of transaction stealing an execution thread is rarely "blocked" within its batch. That is because a "block" occurs only if all transactions within a batch have precursors that are being executed or they themselves are being executed by other threads. In either scenario an executing thread is not blocked for a long time.

It is also important to note that execution threads do not explicitly communicate with one another. An execution thread communicates only with the transactions that it executes (or attempts to execute) through integer flags using atomic instructions (Fetch and increment, compare and swap etc.). Such a communication method incurs very little synchronizational overhead.

3 DISCREPANCIES BETWEEN THEORY AND IMPLEMENTATION.

While working on the implementation of the concurrency system I have found that some of the assumptions present in former work must be relaxed or revisited. Here we give a brief survey of the biggest difference between the algorithm as described before and as implemented in the system.

3.1 PARALLELIZING BATCH SCHEDULE MERGING.

Within the past work we have discussed the possibility of parallelizing the process of batch schedule merging into the system. Such process would be done by partitioning the batch schedule by requested records and merging each of the partitions independently. Such a process would require synchronization among the merging threads. The process of merging batch schedules into the global schedule has proved to be fast enough for the synchronization mechanism to be unnecessary.

3.2 COALEScing NEIGHBORING SHARED LOCK STAGES.

Let us recall that a lock stage is a data structure which holds handles to transactions which may access a resource at the same time. Hence, a "shared" lock stage holds multiple references while an "exclusive" lock stage holds only a single reference. Lock stages for a particular record form a linked list referred to as the lock queue.

3Because the transaction was not yet ready to execute
Logically, two adjacent lock stages within the same lock queue should be coalesced into a single lock stage. This is because there is no conflict between any two of the transactions within these lock stages. Notice, however, that the described situation (of two shared lock stages adjacent) is only created during the merging of a batch into the global schedule. Most importantly, the lock stage already existing within the global schedule might be accessed by execution threads during the merging process. This leads to the conclusion that any such coalescing would have to be synchronized among threads and the cost of such synchronization is too high.

Notice that dropping the assumption of coalescing shared lock stages reduces the theoretical parallelizability of the system. A workload composed solely out of read-only transactions will not be executed "all in parallel," but rather according to the constraints imposed by "batch boundaries" as discussed above. While this might seem like a huge constraint, considering transactions in large batches obviates the problem since the number of transactions within any lock stage will be high enough to provide adequate parallelizable work for any practical system.

4 HARDWARE.

All of the experiments shown have been conducted on an 80-core NUMA machine known at Yale as smorz.cs.yale.edu.

5 WORKLOAD SPECIFICATION.

The workload within all experiments may be simply described as:

GENERAL INFORMATION
  Transaction number: 1000000

DATABASE STORAGE
  Tables number: 1
  Records in table: 1000

READ LOCKS REQUESTED INFORMATION
  Locks requested from: 0
  Locks requested to: 999
  Average # of locks requested: 10
  Std Dev of # of locks requested: 0

WRITE LOCKS REQUESTED INFORMATION
  Locks requested from: 0
  Locks requested to: 999
  Average # of locks requested: 10
  Std Dev of # of locks requested: 0
Hence, I have a set of $10^6$ transactions with randomly distributed read and write sets over 1000 records in a single table.

6 THE THROUGHPUT OF THE SCHEDULING SYSTEM.

First, let us consider the throughput of the scheduling system assuming instant execution of scheduled transactions. This experiment serves to provide a baseline for comparison of the execution system's throughput.

![Graph showing average throughput of the scheduling system as a function of the number of scheduling threads for multiple batch sizes.](image)

Figure 6.1: Average throughput of the scheduling system as a function of the number of scheduling threads for multiple batch sizes.

The first thing to notice is that the scheduler's throughput initially increases for all batch sizes. While this behavior is fully expected, the consequent sharp decrease in throughput is surprising. Time profiles of the system show that this behavior is due to dynamic memory allocation and semi-automatic garbage collection that ensues from smart pointers used within the system. The rather low throughput of the system for batch size of 10 is due to the same effect. To ameliorate this effect I am currently implementing static memory pools.

It is worth noting that the throughput of the system for larger batch sizes does not scale quite as well as that for medium-batch sizes. While I still do not fully understand this behavior, I hypothesize that it is due to the increased memory traffic.

The general trend to keep in mind from these graphs is that the throughput for medium batches scales best within the system. The throughput for big and small batches is in general smaller and does not scale quite as well.
6.1 Investigating the Throughput of the Scheduling System in Time.

The time necessary to create a batch schedule depends heavily on the distribution of read and write sets within the batch. Since I am maintaining ordering among batches it is interesting to investigate the throughput of the scheduling system throughout time as we increase the number of scheduling threads.

As one would expect, an increase in the number of scheduling threads effectively increases the burstiness of the workload produced by the scheduling system. This is because any batch that is converted into a batch schedule may not be merged into the global schedule until all of the former batches have been merged. That means that the probability of producing a "blocked" batch schedule by any thread increases with the number of threads present in the system.

Another curious effect visible within the graphs is the initial extremely low throughput of the system for batch size 10 and 12 scheduling threads. As before, this behavior is still not well understood by us, but I hypothesize that garbage collection might be the culprit in this case as well.

Figure 6.2: Throughput in time for different batches. 1 scheduling thread.

Figure 6.3: Throughput in time for different batches. 4 scheduling threads.

Figure 6.4: Throughput in time for different batches. 8 scheduling threads.

Figure 6.5: Throughput in time for different batches. 12 scheduling threads.
7 Entire system’s throughput.

Within the experiments considering the throughput of the whole system I used 8 scheduling threads because of the high throughput recorded before.

Figure 7.1: Average throughput of the full system as a function of the number of executing threads for multiple batch sizes.

As I discuss the graphs one should remember that:

1. The throughput of the scheduler is highest for medium-sized batches.
2. Higher batch sizes correspond to more parallelizable schedules.

Hence, we expect the throughput of the system to increase with the batch size until a tipping point at which the scheduler’s throughput is too low to keep all of the executing threads busy. The graphs illustrate that behavior exactly. The small batch of 300 provides a base line. The throughput for batch sizes of 1000 and 2000 is always higher than the base line as we would expect. We have seen that the scheduler’s throughput for this configuration is high and we expect the batch schedules to be more parallelizable.

The batch size of 3000 is particularly interesting. The scheduler’s throughput begins to decrease in comparison to that of the system at batch size 2000. However, the schedule produces is more parallelizable allowing more scheduling threads to achieve higher throughput. Further increase in batch size, unsurprisingly, causes throughput decrease as the higher parallelizability of the workload may not make up for the low throughput of the scheduler.
8 Summary.

Within a semester-long effort I have implemented a concurrent database, which uses the formerly developed batched deterministic concurrency control system. While the resulting database has not yet been fully optimized and profiled, it may already be used to show that the batching approach results in increased throughput when compared to the non-batched approach.

9 Future work.

While the semester is nearing to an end, the work on this project will continue. As discussed throughout this document, there is an array of questions and optimizations that remain to be investigated. To be precise, the following set of issues must be addressed:

Static Memory Allocation. We have seen that dynamic memory allocation causes otherwise unexpected slowdowns and generally throttles the throughput of the system. Hence, per-thread static memory pools must be implemented.

Depth of Stealing Recursion. As we have remarked before, stealing transactions improves the throughput in most cases. However, sometimes execution threads are capture within a very deep recursive cycle in an attempt to execute blocking transactions of blocking transactions. The optimal "cut off" recursion depth should be investigated and an artificial "bottom-case" implemented to improve the general throughput of the system.

Changing the Function Used For Packing Creation. We must change the function used to create packings within the scheduling algorithm to find out whether prioritizing certain types of transactions is beneficial to the general throughput of the system.

Cache Locality. Within some of the early experiments we have noticed behavior that might indicate poor cache locality of the system for certain configurations. This behavior should be investigated closer to see whether cache locality of dynamic memory allocation is at fault. In the former case, the algorithms use must be rewritten to make them more cache-local.

Direct Throughput Comparison. We must compare the system with other known high-performance parallel concurrency control systems such as two phase locking, Bohm, MVCC and OCC. This comparison will truly show whether the idea of batched systems is viable and beneficial in the world of concurrency control.

10 Code.

All of the code used for the project may be found on my Github:

https://github.com/stswidwinski/multiversioning

The name of the repository is a result of usage of legacy code. The system is not multiversioned.