AJITTS: Adaptive Just-In-Time Transaction Scheduling

Ana Nunes, Rui Oliveira, José Pereira

AJITTS: Adaptive Just-In-Time Transaction Scheduling

Ana Nunes, Rui Oliveira, and José Pereira

HASLab / INESC TEC and U. Minho
{ananunes,rco,jop}@di.uminho.pt

Abstract. Distributed transaction processing has benefited greatly from optimistic concurrency control protocols thus avoiding costly fine-grained synchronization. However, the performance of these protocols degrades significantly when the workload increases, namely, by leading to a substantial amount of aborted transactions due to concurrency conflicts.
Our approach stems from the observation that when the abort rate increases with the load as already executed transactions queue for longer periods of time waiting for their turn to be certified and committed. We thus propose an adaptive algorithm for judiciously scheduling transactions to minimize the time during which these are vulnerable to being aborted by concurrent transactions, thereby reducing the overall abort rate. We do so by throttling transaction execution using an adaptive mechanism based on the locally known state of globally executing transactions, that includes out-of-order execution.
Our evaluation using traces from the industry standard TPC-E workload shows that the amount of aborted transactions can be kept bounded as system load increases, while at the same time fully utilizing system resources and thus scaling transaction processing throughput.

Keywords: Optimistic concurrency control, adaptive scheduling

1 Introduction

Optimistic concurrency control in distributed data processing systems is increasingly popular. In replicated database systems [1–3], it allows concurrent transactions to execute at different sites regardless of possible conflicts. Conflict detection and resolution are performed at commit time, before the changes are applied to the database. In large scale, high throughput transactional systems such as Google Percolator [4] and OMID [5], implementations of optimistic concurrency control with different isolation levels and locking policies are key to achieving radical scalability.

While optimistic concurrency control allows more concurrency and thus better use of resources than its counterpart, transactions that are later found to conflict are aborted and must be re-executed. Notice that the more transactions are allowed to execute concurrently, the more likely it is for conflicts to arise. Also, any transaction is vulnerable to being aborted by other transactions from the moment it starts to execute until is is certified: the longer it takes to execute and certify a given transaction, the more vulnerable it is. This is the caveat of most optimistic concurrency control strategies: when loaded, latency increases and fairness is compromised, particularly for long-running transactions [1].

In contrast, conservative concurrency control is implemented by subjecting transactions to a priori conflict detection. In some of these protocols, conflict detection is done by associating queues to data partitions [3, 6–8]. The queues are used to serialize transactions that access the same data partitions. The performance penalty imposed by the conservative strategy depends on the grain considered for concurrency control: if the grain is too fine, conflict detection will result in a delay before transaction execution; on the other hand, if the grain is too coarse, transactions that would not otherwise conflict are unnecessarily prevented from executing concurrently [1].

It has been proposed that both approaches be combined by conservatively re-executing previously aborted transactions, which mitigates this issue. However, if appropriate conflict classes cannot be used to ensure no conflicts occur during re-execution, the system will quickly be prevented from exploting optimism [1]. Moreover, although most optimistic concurrency control protocols execute transactions as soon as these are submitted [2, 3], it has been pointed out that the worst scenarios for optimistic concurrency control can be mitigated by limiting the number of transactions executing concurrently [1]. Transaction scheduling on non-distributed settings using queue-theoretic models for automatically adjusting the maximum parallelism level has been studied [9]. However, selecting the correct level of parallelism is not straightforward and can result in a severe limitation to maximum throughput.

In this paper we solve this problem with AJITTS, an adaptive just-in-time transaction scheduler. This mechanism minimize aborts while maximizing transaction throughput by computing the appropriate start time for each transaction. The intuition behind this proposal is simple: If a transaction must wait to be certified in the correct order, to ensure consistency in a distributed system, it is better that it waits prior to execution, as it is not susceptible to being aborted by conflicts with concurrent transactions. The implementation of this simple intuition does however imply that the system is continually monitored and that an appropriate execution start time is computed for each transaction.

The rest of this paper is structured as follows. In Section 2 we show that there is an ideal configuration for each workload, that improves performance regarding the basic optimistic protocol, introducing then the mechanism used to dynamically compute such configuration. In Section 3 we use traces obtained from the TPC-E workload running on a MySQL database server to simulate different workload scenarios and evaluate our proposal. Finally, Section 4 concludes the paper.

2 Approach

The main insight leading to our proposal is as follows: Since transactions are vulnerable to being aborted from the time execution starts until certification, in order to minimize the number of aborts, execution should start as late as possible. On the other hand, if certification goes idle because transactions at the head of the queue have not been completely executed, the throughput decreases. Our approach is thus based on reaching and maintaining the optimal level of queuing in the system: As low as possible to minimize aborts but as high as needed to ensure that certification doesn't go idle, to maximize throughput.

2.1 System model

To test this hypothesis, we assume an abstract model that captures key aspects of a distributed transaction processing system. First, we assume that transactions submitted to the system are totally ordered and placed at the tail of a queue in the *not_executed* state. This models either a centralized queue at the transaction manager server [5] or local replicas of a queue built by a group communication system [2]. Because transactions must be certified in a conflict-equivalent order to the total order on which replicas agreed, the system can be modelled as a single queue in which transactions go through several states.

We then introduce a line in the queue that determines which transactions should start execution: all transactions before the line are not eligible to start executing, while all transactions between the line and the head of the queue that are in the *not_executed* state are to be executed. Simply put, transactions are evaluated for execution whenever a transaction arrives to (i.e. is submitted) or leaves the queue (i.e. committed or aborted). Transactions can only be certified upon reaching the head of the queue and having completed execution, entering the *certification* state.

Conflict detection ensues: if the transaction was in the *executed* state, conflicting transactions in either *executing* or *executed* states are aborted and the transaction is immediately certified; if it was in the *aborted* state, the outcome is an abort. For certification, we assume *snapshot isolation*, which differs from serializability by considering only write/write conflicts [10]. This is used in the overwhelming majority of current RDBMSs and has also been favored in distributed transaction processing systems.

2.2 Impact of scheduling

Ideally, the line would be placed at such a position that each transaction completes execution just as it arrives at the head of the queue, minimizing the time spent in the *executed* state before reaching *certification*, thus minimizing its vulnerability to being aborted by others. However if the transaction reaches the head of the queue in either *not_executed* or *executing* states, it cannot be certified until it finishes. Certification must occur in a conflict-equivalent order to the previously established total order, which is key to guaranteeing determinism in a distributed setting. As such, transactions running late cannot be overtaken by others, leaving certification idle.

Transactions can have widely varying execution times (i.e. duration), which should be considered when scheduling them. Let d_t be the estimate of the duration of transaction t, which can be easily obtained from a database planner.

Assuming that the head of the queue corresponds to position 1, let

$$pos_t = input \cdot d_t \tag{1}$$

be the position in the queue after which transaction t will be executed. This means that transaction t will be executed when there are $pos_t - 1$ or less transactions ahead of it in the queue, which is the same as placing a line in the queue. Notice that transaction t is not scheduled for a particular instant in absolute time: it is relative to the current inter-arrival rate of transactions at the head of the queue. This enables out-of-order execution: a small transaction will begin execution near the head of the queue, while a very large transaction will begin execution as soon as it is submitted.

The *input* parameter provides a simple way to adjust how early transactions should be executed: for the same estimated duration, a higher value of *input* means that the transaction will be executed earlier than with a lower value.

We then built an event-driven simulation of the abstract system model with the adjustable executuon start line. This simulation allows the position of the line that triggers execution to be adjusted, as well as to use different workloads, further described in Section 3. From this simulation we collect a number of statistics, namely: usable throughput, considering transactions that can be committed; share of transactions aborted due to concurrency conflicts; and latency at each state, from which we derive also end-to-end latency.

Figure 1 shows the latency breakdown for a particular workload (400 clients) while varying the input parameter. On the right hand, transactions are scheduled early, thus reducing the amount of time in the *not_executed* state, shown in blue. In fact, an extreme setting of the parameter is equivalent to the baseline optimistic protocol, meaning that transactions are immediatly scheduled for exection and the entire impact of synchronization happens in the *executed* state. On the left, transactions are scheduled later, thus waiting an increased amount of time before execution, but waiting very little as *executed* (in brown). As expected, varying this parameter does not have an impact in execution duration (in red). As expected, we observe that overly delaying transaction execution has an impact in total latency.

Figure 2 shows a complete set of statistics for three different workloads. These workloads differ only in the number of concurrent clients submitting transactions. First, we observe that besides impacting end-to-end latency, the input parameter that determines when execution is started also impacts throughput and the abort rate leading to the following trade off:

– On the left, with a larger delay before execution, transactions arrive at the head of the queue but are not yet fully executed, thus stalling certification

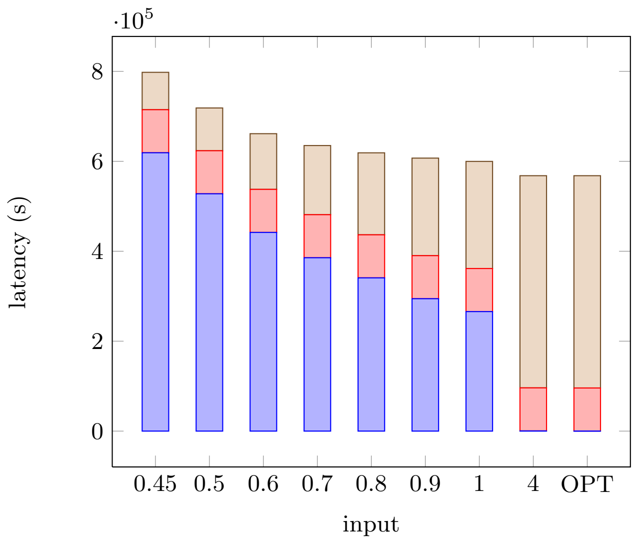

Fig. 1. Average latency breakdown with a varying scheduler parameter: Pre-execution delay (blue), execution latency (red), and queueing for certification (brown).

and leading to reduced throughput. However, the small delay to certification leads to a reduced number of concurrency aborts.

– On the right, transactions are executed fairly ahead of time, thus avoiding stalling the queue. On the other hand, by having started early they become concurrent with a larger number of transactions and thus lead to an increased amount of rollbacks due to conflicts.

Notice that, for example, if *input* is between $0.4 \cdot 10^{-3}$ and $0.9 \cdot 10^{-3}$ for 800 clients, throughput is sub-optimal because transactions are being executed too late. This can be confirmed by analyzing the transaction latency in the same interval. Also, for example for 200 clients, the abort rate steadily rises as *input* increases, but when *input* becomes larger than 1, the abort rate stabilizes at around 5%. This happens because after this point roughly all transactions are being executed as soon as they are submitted, equivalent to using DBSM [2]. This effect occurs for any number of clients, it is just a question of using a large enough *input*.

Figure 3 shows similar results when, instead of varying the workload, we vary the resources available for execution. This leads to the time spent in the *executing* state growing. However, the same trade off holds. In short, we observe that there is an intermediate configuration that provides the best usable throughput with moderate latency. This is true regardless of the number of concurrent clients or the resources available to execute transactions. This optimal configuration is however different for different settings, which makes its configuration by the system developer unfeasible. As it varies with the workload, it is also impractical as a configuration parameter.

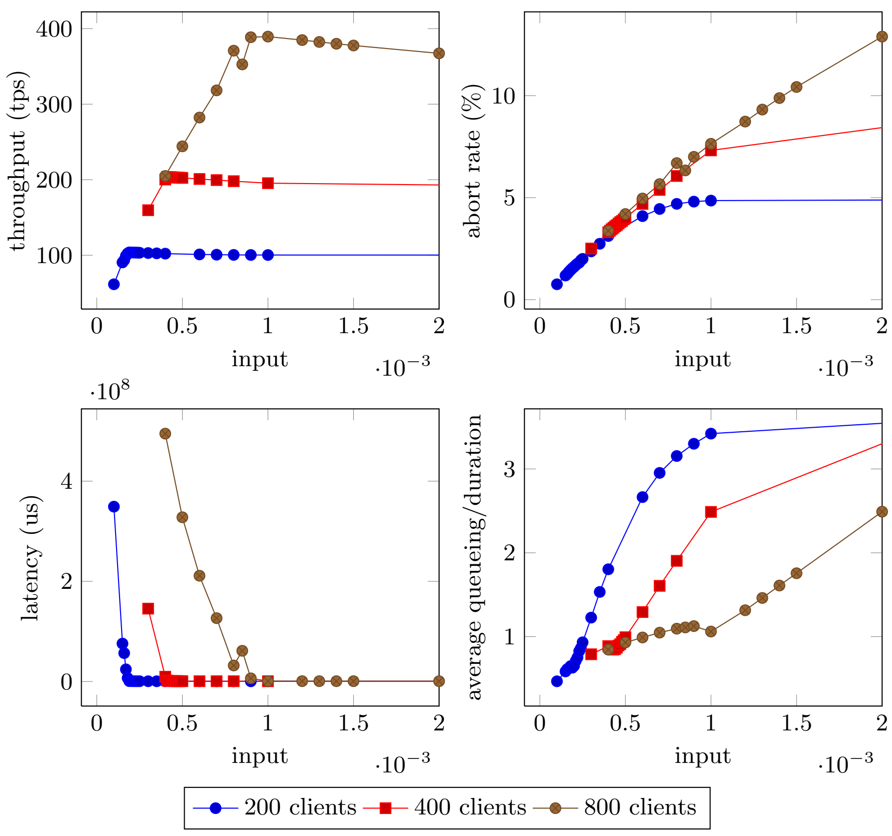

Fig. 2. Effect of the input value on throughput, the abort rate, transaction latency and on the ratio between average transaction queueing and average duration for different numbers of clients.

2.3 Adapting the workload

The question becomes how to determine an appropriate *input* value that provides optimal throughput without resorting to trial and error.

A simple adaptation mechanism would be to simply start execution one position sooner whenever a transaction reaches the head of the queue in the *not_executed* or *executing* states or one position later whenever it has to wait in the *executed* state or has been *aborted*. This approach was tested, but such an adaptation mechanism, while simple, causes oscillation, since such changes are too abrupt [11].

Let t_{state} be the instant in which transaction t reaches state *state*. For any transaction t,

$$s_t = t_{not\_executed} - t_{executing}$$

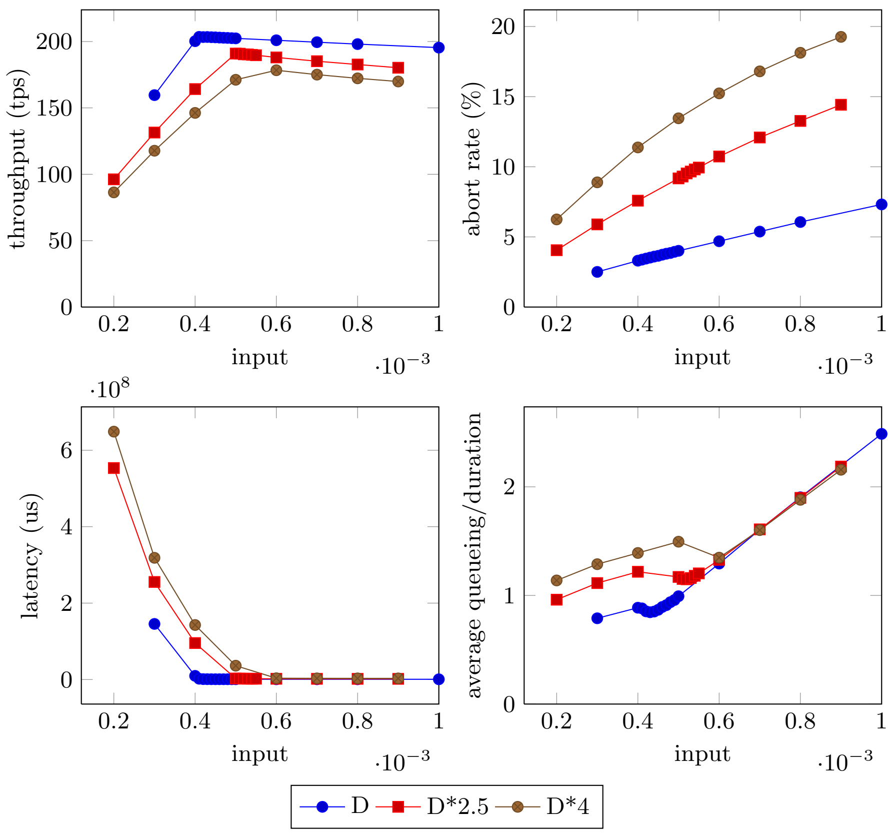

Fig. 3. Effect of the input value on throughput, the abort rate, transaction latency and on the ratio between average transaction queueing and average duration for different distributions of transaction duration.

measures how long transaction t had to wait to begin to execute since it was submitted, and

$$q_t = t_{certification} - t_{executed}$$

measures how long transaction t had to wait after its execution was complete before reaching the head of the queue to be certified, henceforth simply referred to as pre-execution delay and queuing, respectively. Queuing is directly affected by whether transactions are executed sooner or later: on average, the former increases queuing while the latter decreases it. Let Q be a weighted cumulative rolling average of q and Q_{opt} the optimal level of queuing for a system. An adaptive mechanism that reacts to the state of the queue can be defined using a proportional-integral-derivative controller[11] with Q as the sensor, Q_{opt} as the set point and *input* as the system input [11] such that:

$$
\begin{aligned}
&error = setpoint - sensor \\
&P_{value} = Kp * error \\
&input{+} = P_{value}
\end{aligned}
$$

Simply put, the error of the measured value (*sensor*) relatively to the desired value (*setpoint*) is used to update an input to the system (*input*) which will in turn impact the measured value, constituting the control feedback loop.

Kp is referred to as proportional gain, a tuning parameter that adjusts how the sensitivity of the controller, *i.e.*, the magnitude of the adaptation relatively to the magnitude of the error. Several methods exist for selecting an appropriate value for Kp, from manual tuning to methods based on heuristics[12].

Because computing the position of the line relies on an estimated value for transaction duration, for which the expected value is the mean, depending on the variance of the population, selecting a set point of 0 would mean that several transactions would not finish its execution in time, leaving certification idle.

The key to finding Q_{opt} is in comparing the bottom-right chart of Figure 2 which shows the ratio between the average queuing and the average duration of all transactions with the top-left chart showing throughput. Notice that the *input* values that achieve optimal throughput in the top-left chart match those for which the ratio in the bottom-right chart is roughly 1. Intuitively, selecting the set point to target average duration would mean that certification does not go idle and, consequently, that the rate at which transactions are certified is the same as the rate at which transactions arrive at the head of the queue which corresponds to optimal throughput but minimizing the size of the queue meaning a minimal abort rate. If deemed necessary, due to high variance in transaction duration, from the cumulative distribution function of transaction duration one can choose a value for the set point corresponding to a desired percentile: the higher the percentile of the chosen value, the higher the number of transactions that will have completed execution as expected.

3 Evaluation

3.1 Workload

TPC-E [13] is a benchmark that simulates the activities of a brokerage firm which handles customer account management, trade order execution on behalf of customers and the interaction with financial markets. This analysis was based on *tpce-mysql*,[1] an open-source implementation of the TPC-E benchmark. This benchmark defines 33 tables across four domains: customer, broker, market and dimension and 10 main transaction types that operate across the domains. TPC-

[1] `https://code.launchpad.net/perconadev/perconatools/tpcemysql`

E's read/write transactions are: Market Feed (MF), Trade Order (TO), Trade Result (TR), Trade Update (TU) and Data Maintenance (DM).[2]

AJITTS was evaluated using a simple event-driven simulator that enables a profound analysis of each aspect of scheduling and concurrency control of replication protocols. The simulator processes execution traces obtained by running TPC-E like benchmark on a centralized MySQL [3] database and then parsing the resulting binlog to generate the workload to test replication protocols. In essence, the simulator uses the following information from the binlog: the timestamps at which each transaction started, how long it took to execute each transaction and transaction write sets.

Another relevant feature of the simulator is that it allows the parallelization of the load generated by serial runs of TPC-E over the same database. In short, it does so by creating unique identifiers for each transaction and by manipulating timestamps making these relative to a reference instant. As a result, the load applied to the protocol under test can be easily scaled. Also, the applied load is not limited by resource constraints on the original MySQL database: there is no limit on the number of load units that can be applied in parallel.

The values of transaction duration extracted from the binlog reflect the penalty introduced by synchronization and locking in the MySQL engine when the benchmark is executed. A correction factor (β) can be calibrated by running the traces through the simulator with optimistic scheduling, without admission restriction and without re-execution, chosen such that the abort rate is close to 1%. The reason for this is that the sequence of transactions in the binlog has implicitly been proved to be conflict-free with the original values for transaction duration.

Let dur'_t be the duration extracted from the binlog for transaction t. The respective value to be used in the simulation is $dur_t = \beta * dur'_t$.

The value of the correction factor depends on the benchmark load induced on MySQL. Therefore, the β used in the simulation is independent of the number of parallel traces used to fuel the simulator, as long as the load induced by each benchmark run was about the same. If using another set of traces, the beta must be recalculated. β is 0.2 for the traces used to evaluate AJITTS.

As discussed in Section 2, the set point should be chosen taking into consideration the distribution of transaction duration. In the results presented here, the set point used in AJITTS is the same as the average transaction duration of the given workload.

In order to simplify the implementation of AJITTS, instead of estimating the duration of each transaction, a line is placed on the queue for each type of transaction. The position of the line for a transaction of type MF, for example, is calculated as

$$line_{MF} = input * d_{MF}$$

[2] The Data Maintenance transaction type operates exclusively on a separate group of tables. As such, it is not relevant for this analysis and is essentially omitted from the discussion that follows.

[3] `http://www.mysql.com`

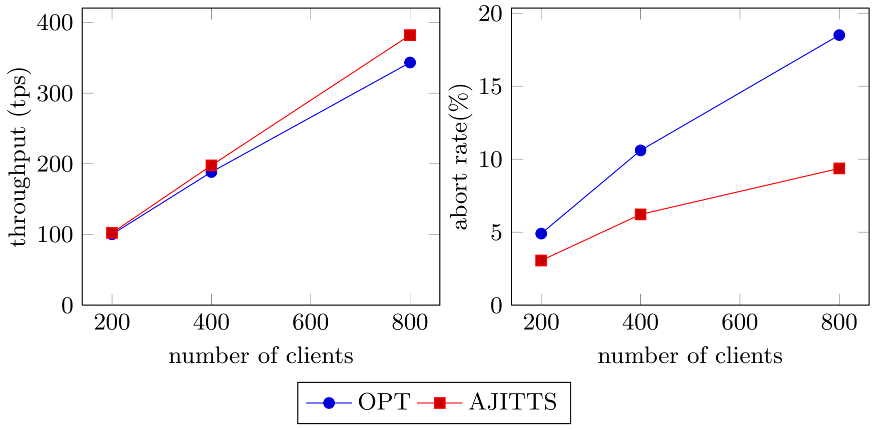

Fig. 4. Throughput and abort rates using OPT and AJITTS for different numbers of clients.

where d_{MF} is an online estimate of the duration of MF transactions using a cumulative rolling average.

While TPC-E already provides strictly defined transaction types, one could classify transactions in generic workloads by either considering the similarity in execution plans or, for example, the conflict classes accessed by transactions. Still, AJITTS can be implemented without this simplification, computing a line for each individual transaction.

3.2 Results

We compare AJITTS with OPT, a protocol with a standard optimistic scheduler. Simply put, OPT schedules each execution as soon as it is submitted. Using the TPC-E based workload described in Section 3.1, this is simulated by admitting at most one transaction per client, since clients are single-threaded. Notice that without this restriction, the number of concurrent transactions would be higher than allowed in the original benchmark.

Figure 4 compares OPT and AJITTS in terms of throughput and aborts for three workloads that differ only on the number of concurrent clients submitting transactions. Notice that even though AJITTS introduces delays on transaction executions, throughput is not only not adversely affected, but actually improved. Also, AJITTS clearly succeeds in significantly reducing the abort rate. In fact, a clear trend of further improvement can be observed in both charts as the load increases.

Figure 5 shows how the line positions per transaction type evolve during a run with a particular workload. Line positions are updated whenever the estimates for execution duration change or whenever the adaptation input parameter changes. The position of the line for each transaction type converges quickly: the

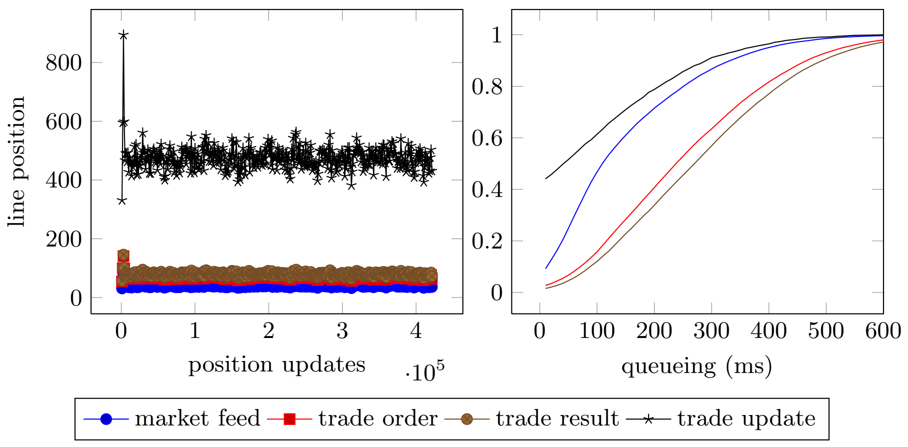

Fig. 5. Evolution of the position of the lines during a particular run.

amplitude of the variation stabilizes after considerably few updates. In particular, TU transactions actually consist of three different types of subtransactions: the variability of the duration of trade update transactions is mirrored in the variation of the position of the line for this type of transaction. Notice that TU transactions are scheduled much earlier than other types of transactions. Figure 5 also shows the cumulative distribution function of the measured queueing (q) aggregated by transaction type, which is a result of the position of the lines.

Figure 6 shows how the different average durations (in red) influence the pre-execution delay (in blue) when using AJITTS: again, TU transactions (TUa) are scheduled much earlier than others, while MF transactions (MFa), for instance, are only executed nearer the head of the queue. When comparing the results regarding, for example, MF transactions, the average time during which these are vulnerable to being aborted much smaller using AJITTS (110 ms) than using OPT (562 ms). This is also the case for TR and TO transactions. However, for TU transactions queueing actually increases using AJITTS. This is a consequence of ensuring that certification does not go idle. As expected, the net effect is still a reduced abort rate.

Considering different values of β shapes the workload: higher βs simulate less available resources and vice-versa. Figure 7 shows how AJITTS leverages available resources significantly better than OPT. In particular, the less available resources, the more OPT's throughput decreases relatively to AJITTS.

4 Conclusion

Although increasingly popular and often used, optimistic concurrency control may lead, with more demanding workloads, to a large number of conflicts and aborted transactions. This endangers fairness and reduces usable throughput.

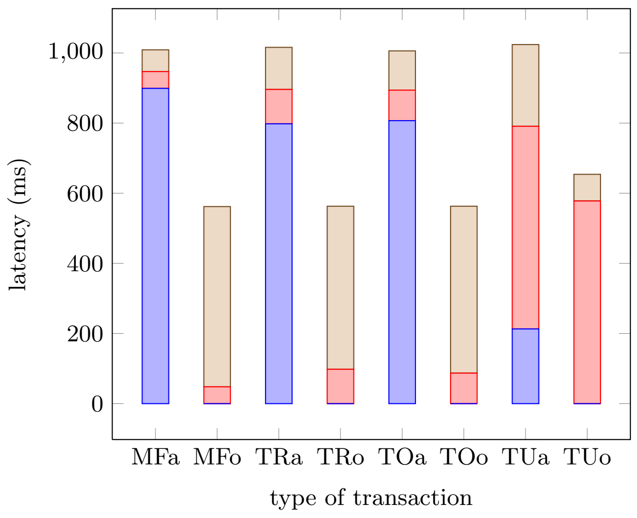

Fig. 6. Average latency breakdown using AJITTS and OPT: Pre-execution delay (blue), execution latency (red), and queueing for certification (brown). Columns MFa, TRa, TOa and TUa refer to an execution of the AJITTS protocol, while the others refer to an execution of the OPT protocol.

Previous attempts at tackling this problem required workload-specific configuration and would still impact peak throughput [1].

With AJITTS, the adaptive just-in-time transaction scheduler, we provide a solution that does not require workload specific configuration and adapts in run-time to current workload and resource availability conditions. This is achieved by delaying transaction execution, for each transaction individually based on the estimated time to complete and current queueing within the system.

AJITTS was then evaluated using a simulation model driven by traces from TPC-E running on MySQL, demonstrating that it clearly outperforms the baseline protocol. In fact, in addition to reduced aborts, it actually improves peak throughput even if it throttles transaction execution. This is the consequence of using available resources better.

Acknowledgements

This work is part-funded by: ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project Stratus/FCOMP-01-0124-FEDER-015020.

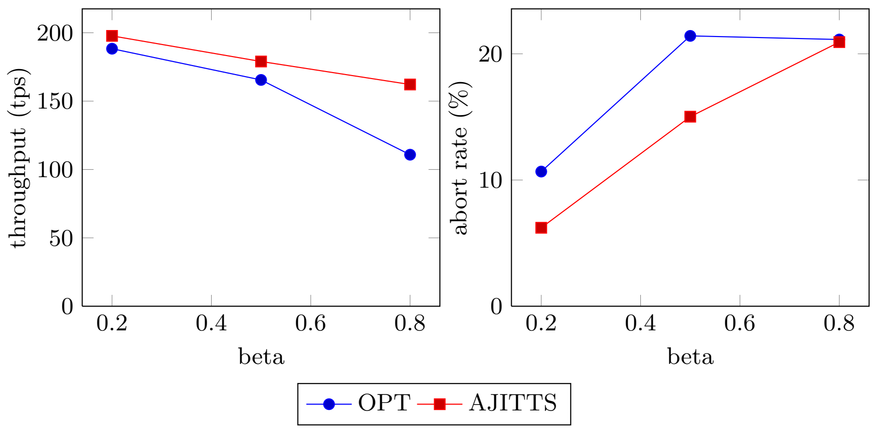

Fig. 7. Throughput and abort rates for OPT and AJITTS for different βs.

References

1. Correia, A., Pereira, J., Oliveira, R.: Akara: A flexible clustering protocol for demanding transactional workloads. On the Move to Meaningful Internet Systems: OTM 2008 (2008) 691–708
2. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distributed and Parallel Databases **14** (2003) 71–98 10.1023/A:1022887812188.
3. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In: Proceedings of the 26th International Conference on Very Large Data Bases. VLDB '00, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2000) 134–143
4. Peng, D., Dabek, F., Inc, G.: Large-scale incremental processing using distributed transactions and notifications. In: 9th USENIX Symposium on Operating Systems Design and Implementation. (2010) 4–6
5. Yabandeh, M., Gómez Ferro, D.: A critique of snapshot isolation. In: Proceedings of the 7th ACM european conference on Computer Systems. EuroSys '12, New York, NY, USA, ACM (2012) 155–168
6. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: Scalable replication in database clusters. Distributed Computing (2000) 147–160
7. Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B., Alonso, G.: Improving the scalability of fault-tolerant database clusters. In: Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on, IEEE (2002) 477–484
8. Kemme, B., Pedone, F., Alonso, G., Schiper, A.: Processing transactions over optimistic atomic broadcast protocols. In: Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on, IEEE (1999) 424–431
9. Schroeder, B., Harchol-Balter, M., Iyengar, A., Nahum, E., Wierman, A.: How to determine a good multi-programming level for external scheduling. In: Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on. (april 2006) 60

10. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Middleware based data replication providing snapshot isolation. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, ACM (2005) 419–430
11. Aström, K.J., Murray, R.M.: Feedback systems: An introduction for scientists and engineers. Technical report, Princeton University Press (2007)
12. Aström, K., Hägglund, T.: Automatic tuning of simple regulators with specifications on phase and amplitude margins. Automatica **20**(5) (1984) 645 – 651
13. Transaction Processing Performance Council (TPC): TPC Benchmark E - Standard Specification. Revision 1.12.0 edn. (June 2010)