Model-based Proactive Read-validation in Transaction Processing Systems

Simone Economo, Emiliano Silvestri,
Pierangelo Di Sanzo, Alessandro Pellegrini
Sapienza, University of Rome
Lockless S.r.l.

Francesco Quaglia
University of Rome Tor Vergata
Lockless S.r.l.

***Abstract*—Concurrency control protocols based on read-validation schemes allow transactions which are doomed to abort to still run until a subsequent validation check reveals them as invalid. These late aborts do not favor the reduction of wasted computation and can penalize performance. To counteract this problem, we present an analytical model that predicts the abort probability of transactions handled via read-validation schemes. Our goal is to determine what are the suited points—along a transaction lifetime—to carry out a validation check. This may lead to early aborting doomed transactions, thus saving CPU time. We show how to exploit the abort probability predictions returned by the model in combination with a threshold-based scheme to trigger read-validations. We also show how this approach can definitely improve performance—leading up to 14% better turnaround—as demonstrated by some experiments carried out with a port of the TPC-C benchmark to Software Transactional Memory.**

***Index Terms*—performance optimization, transactional runtime environments, early transaction aborts**

I. Introduction

Read-validation is a widely used technique in concurrency control protocols for transactional systems. It allows transactions to run without locking data being read, thus unleashing concurrency especially in read-intensive workloads. This approach is actually exploited in several families of concurrency control protocols, such as optimistic and multi-version ones [1]. In the context of distributed transactional systems, like in-memory data management platforms in the Cloud, read-validation based concurrency control allows for high scalability even under data replication [2]. On the downside, because of the intrinsic structure of read-validation schemes, the readset of a running transaction can become no longer valid because of conflicting updates performed by concurrent transactions. Therefore, read-validation schemes need to assess the validity of a transaction by means of a read-validation operation. Typically, a transaction is validated at its commit attempt. However, it is also possible to assess the validity of a transaction at earlier points along its lifetime. This can be done either for the sole purpose of performance, or to support correctness criteria stricter than serializability, such as opacity [3]. In any case, a concurrency control protocol doesn't necessarily reassess the validity of a transactional readset immediately after a conflict has arisen due to updates by a concurrent transaction. This prevents the possibility of early aborting a transaction that is already doomed to abort. In fact, the transaction will run until a commit is attempted, or until the concurrency control protocol decides to perform a read-validation. In both cases, this ultimately leads to waste of resources (e.g., CPU).

In this article we present a performance optimization for transaction processing systems. Our approach is based on a simple analytical model which predicts, at runtime, the probability that a transactional readset has become invalid. Such probability is then used by a threshold-based mechanism to proactively fire a transaction validation check whenever its value exceeds a given threshold. The threshold parameter can be tuned so as to optimize the trade-off between how much processing time can be saved (thanks to early aborts), and how often this gain may be achieved.

We frame our contributions within the field of Software Transactional Memory (STM), a paradigm based on the notion of transaction to support synchronization operations involving shared-data accesses by concurrent threads. Its relevance is witnessed by the fact that it is nowadays supported by the world leading C(++) compiler for Unix systems, and is integral part of the runtime environments of some well-known programming languages (such as Clojure and Haskell). The STM paradigm is also at the core of the construction of several NoSQL data stores used in modern systems, like the Infinispan data layer for JBoss applications [4]. In any case the model we present, and the transaction validation technique based on it, is of general conception, and could be exploited in the context of different transactional system technologies.

The experimental assessment of our model-based validation technique has been carried out by integrating it into the open source TinySTM [1] package[1]. Experimental tests have been performed on a port of the TPC-C benchmark to STM.

The remainder of this article is structured as follows. In Section II we discuss related work. The model-based read-validation technique is illustrated in Section III. An experimental assessment of our proposal is presented in Section IV.

II. Related Work

In the literature, several analytical models have been presented which cope with very disparate concurrency control

[1]Our implementation has been made open-source and is available at the HPDCS research group github repository `https://github.com/HPDCS`

schemes for transactional systems (see, e.g., [5]–[7]). Most of them have been exploited as off-line tools for performance analysis and prediction, while a few of them have been exploited as runtime decision supports. In the specific context of STM, analytical runtime decision models have been proposed in order to determine suited levels of parallelism for running applications [8]–[10]—as a way to avoid trashing due to excessive transaction aborts. However, to the best of our knowledge, none of the past literature works provides a model coping with the problem of determining proactively whether to (re)assess the validity of a transaction along its life time in concurrency control mechanisms based on read-validation schemes.

In [11] a model-based proactive approach is exploited in order to determine whether to scale up/down the number of nodes of an in-memory transactional data store depending on workload changes. However, this proposal does not attempt to optimize performance via proactive checks of transactions' validity along their lifetime. The proposed model predicts performance for the scenario where transactions are aborted either when reaching their commit point, or when the concurrency control mechanism detects some inconsistent access to data. Previously issued accesses are not accounted for to proactively predict transaction validity, as instead we do in our proposal. On the other hand, the approach in [11] can be seen as orthogonal to ours. The former is mostly oriented to drive decisions on the amount of resources to be used to sustain a given workload, while our proposal is oriented to optimize the usage of each individual CPU-core by reducing CPU-waste.

In [12] the transaction validity check is triggered periodically, via an ad-hoc operating system support integrated within Linux. Differently from this proposal, our model-based read-validation scheme does not need special support within the operating system, thus being of wider applicability. Also, the solution in [12] does not rely on any prediction scheme for choosing at which points to attempt transaction validation. The validation task is triggered independently of the probability that the transaction has become invalid. In our proposal, the validation task is triggered on the basis of predictions carried out by the analytical model.

Several works have targeted the reduction of the incidence of transaction aborts via heuristic based approaches [13]–[15]. These solutions either try to sequentialize conflicting transactions on the same thread or control the concurrency degree of the STM-based application by changing the number of threads/transactions that are allowed to run in parallel. A comprehensive survey of the proposed techniques can be found [16]. Other techniques have been oriented to the optimization of the strategy for managing contention across concurrent transactions [17], [18]. Some of these approaches also enable the runtime adaptation of the contention management strategy to the workload profile [18]. The orthogonal issue of mapping threads to CPU-cores for performance optimization has been addressed in [19]. An approach aimed at reducing the waste of CPU-time and energy caused by transaction aborts, which acts with per-transaction granularity, is the one in [20]. Here the authors propose a solution for enabling a no longer valid transaction to be rolled back partially (rather than totally), which may help saving work otherwise doomed to be unfruitful. Our work is orthogonal to (and ideally combinable with) the above solutions. Our target is to select points along the transaction lifetime where a read-validation operation is likely to produce an early abort of the transaction. On the other hand, similarly to the proposal in [20], we retain the ability to reduce CPU-time waste on per-transaction granularity. Indeed, our analytical model is exploited to trigger early transaction aborts that would have never be triggered by the underlying transactional layer otherwise.

III. Model-based Proactive Read-validation

Our probabilistic model is built on top of a transactional system with an optimistic concurrency control. Transactions are executed speculatively and can be aborted if something has doomed their execution. In particular, we assume that the readset of a running transaction can, at any time, be invalidated by a *conflicting transaction*—i.e., a concurrent transaction performing conflicting updates on the same data. In this scenario, the concurrency control protocol will not necessarily reassess the validity of the first transaction in a timely manner. Therefore, it will miss the opportunity to early abort the transaction.

Our focus in on those concurrent updates which conflict with prior reads by a different transaction. Stated differently, we only consider the abort probability deriving from *write-after-read* conflicts in the readset of a running transaction. Any abort or conflict occurring due to other events—write-after-write or read-after-write conflicts, and even conflicts on internal metadata—are not considered in the model and are not accounted for when computing the abort probability of a transaction. Write-after-write conflicts can be already handled efficiently by many read-validation schemes, since typically a transactional write operation takes a lock on the transaction object being written. The same holds true for read-after-write ones, as reading an object which is subject to a write lock is disabled in many protocols enforcing strict correctness. As for internal metadata conflicts, this is an orthogonal problem which depends on the specific software implementation of a given concurrency control protocol. For the sake of brevity, and in accordance with our focus on write-after-read conflicts, in the rest of this paper we will refer to read-validations as simply *validations*. Additionally, we will call *abort probability* the probability of observing any write-after-read conflict on the objects read by a running transaction. Therefore, the abort probability of a transaction is the probability that a transaction has become invalid (or doomed) due to conflicting updates to the objects in its readset.

A *commit validation* is the task which is performed at commit time to check that the readset of a transaction is still valid at that point. A validation is *proactive* (also termed as *early* validation) if it is performed prior to commit time, in an attempt to anticipate an abort that would only occur

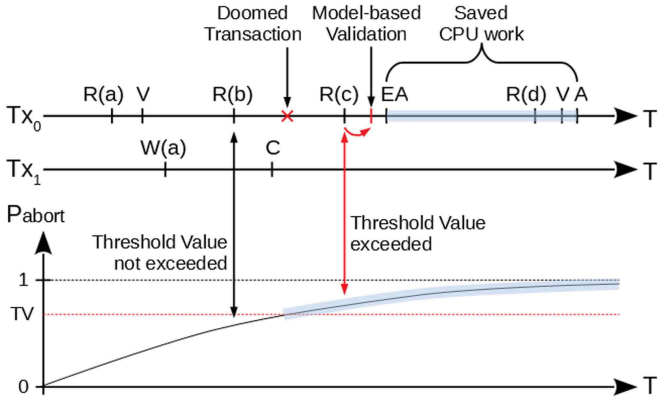


Fig. 1: A visual representation of our threshold-based mechanism for proactive read-validation.

at commit time. When a proactive validation produces an abort, we call it an *early abort*. Generally speaking there are many points along the lifetime of a transaction which can be good candidates for proactive validations (e.g., upon accessing a transactional object for the first time). Choosing whether and when to perform such an early validation is a choice which depends on performance and correctness aspects. Indeed, many concurrency control protocols already perform proactive validations that may lead to early aborts. Despite this fact, in our work we show how to install even further proactive validations within the execution of transactions, in an attempt to achieve a higher amount of early aborts and save even more computing resources. These additional early validations are *model-based*, i.e., they are driven by the analytical model explained in Section III-A, which takes a transactional readset as input and returns the probability of an abort resulting from observing conflicting accesses to the objects in this set. When we wish to distinguish model-based validations from other validations already performed by the concurrency control protocol, we call the latter as *spontaneous* validations. In order to trigger a model-based proactive validation, the computed abort probability must exceed a given threshold value, as we will discuss in detail in Section III-B.

Figure 1 shows two conflicting transactions performing read and write operations on four shared objects—a, b, c, and d. If our model-based mechanism is absent, the underlying concurrency control mechanism is anyhow able to perform some early validations (marked as V). The first of such early validations is forced on transaction Tx_0 right after its read access to object a, but at this point the readset of Tx_0 is still valid and so there is no need to abort. In between this access and the next access to object b, the readset of Tx_0 becomes invalid due to a conflicting write access to a performed by another transaction Tx_1, which occurred after the first validation of Tx_0 and becomes visible after Tx_1 commit (marked as C). Therefore, Tx_0 is doomed to abort. However, only after Tx_0 has performed a read access to d the underlying concurrency control mechanism decides to trigger another validation and the abort (marked as A) of the transaction Tx_0. Hence, the time between the access to object b and that to d is spent doing useless work. Let's see what happens we introduce a validation mechanism which is based on our analytical model and the abort probability computed over time. In the example we only report the abort probability for transaction Tx_0. Whenever such probability exceeds the specified threshold value (TV), the model-based mechanism is ready to trigger a model-based validation. In the example, this may happen as soon as a read to object c is performed, thus producing an early abort and reducing the time spent on useless computation by an amount which is equal to the highlighted area on the timeline of Tx_0.

The remainder of this section is devoted to providing the details on the analytical model being used and to discuss the threshold-based mechanism used to trigger model-based proactive validations.

A. The analytical model

We denote with $\mathcal{D}$ the repository of data objects available to transactions for read and write operations, with d_j being the j-th element of this set. The readset of a transaction x at time t is denoted with $\mathcal{R}_x(t)$. It is composed of a sequence r_i of reads of elements in $\mathcal{D}$. Each r_i is a tuple $\langle j_i, t_i \rangle$ containing the index of the transactional object being read and the time at which the read occurred. Both t and t_i represents points on a physical time axis[2]. We also maintain a *global commit clock* which advances with each distinct commit of a transaction. This clock is essentially a global counter of committed transactions and exists on a logical time axis. To ease the mapping between logical and physical time, as required in our analysis, we conveniently denote with $cc(t)$ the current global commit clock at physical time t.

With no loss of generality, we assume a system with one or more execution phases, such that each phase reaches stationary conditions and such conditions may vary across different phases. Therefore, we essentially model the system independently in each one of the aforementioned phases. As soon as each phase becomes stable, the rate at which updates to transactional objects are carried out by transactions has a characterizing average value. In such a case, t and t_i refer to the elapsed time since the beginning of the current stable execution phase.

The number of updates to a given element d_j in $\mathcal{D}$, which have occurred up to time t, are denoted as $n_j(t)$. The value $n_j(t)$ is by definition lower than or equal to $cc(t)$, since an object d_j can be updated at most once at each commit. The update rate λ_j of a transactional data object d_j at steady state is computed as the limit of the number of updates globally performed by transactions divided by the current global commit clock at time t when t tends to infinity:

$$\lambda_j = \lim_{t \to \infty} \frac{n_j(t)}{cc(t)} \tag{1}$$

[2]Note that such wall-clock time has nothing to do with the version clock (or object timestamp) of time-based and/or multi-version transactional systems.

In our model, the update rate is used to compute the probability that one or more updates affected any object d_j in between s and t, where s is a reference time and t is the current time. This scenario can be modelled using a Bernoulli trial where the success probability p_j is λ_j and the failure probability q_j is $1 - p_j$. In this context, we observe a success when the object d_j is updated at least once by any transaction, while a failure means that no update has occurred in between s and t. Since an update can only occur upon a global commit clock increment, the probability of failure in the $[s, t]$ time interval depends on the number of global commit clock increments in the same period. Therefore, the failure probability over $[s, t]$ is the probability of observing as many failures as the number of commits performed in the same physical time period, which can be expressed as $cc(t) - cc(s)$. The resulting failure probability corresponds to the probability of failure of independent Bernoulli trials, derived using a geometric distribution:

$$q_j(s;t) = q_j^{cc(t)-cc(s)} \tag{2}$$

The reference time for a transactional object read by a transaction coincides with its reading time t_i. Anything that happened before this moment is not of interest for the model, as the object didn't belong to the transactional readset. Therefore, we are only interested in those updates that occur after a transactional object in $\mathcal{D}$ is actually read by a transaction x and is put into its transactional readset $\mathcal{R}_x$. Given that s equals t_i for each r_i in $\mathcal{R}_x(t)$, the failure probability at time t can be rewritten as:

$$q_{r_i}(t) \equiv q_{j_i}(t_i;t) = q_{j_i}^{cc(t)-cc(t_i)} \tag{3}$$

The failure probability for the entire readset $\mathcal{R}_x(t)$ of transaction x is the probability of failure for any object in the readset. In this context, a readset $\mathcal{R}_x$ containing objects $r_i = \langle j_i, t_i \rangle$ is *valid* at time t if no object d_{j_i} read at t_i has been updated in $(t_j, t]$ by some concurrent committing transaction. The following equation expresses the probability of observing a valid readset at time t:

$$\mathcal{Q}_x(t) = \prod_i q_{r_i}(t) = \prod_i q_{j_i}(t_i;t) \tag{4}$$

To obtain the abort probability for transaction x at time t, we must have at least a success (therefore an update) inside $\mathcal{R}_x(t)$, meaning that the readset is invalid. In accordance with elementary probability theory, this means that the modelled abort probability of a transaction x having a readset at time t can be expressed as:

$$\mathcal{A}_x(t) = 1 - \mathcal{Q}_x(t) \tag{5}$$

An useful property of the geometric distribution that we exploit is the *memorylessness* property. It states that the probability of observing a failure at time t given that a reference time s already passed is actually equal to the probability of observing a failure in between s and t. Notice that we already exploited this property in Equation 2. In that case, the reference time s was the time at which we first read the transactional object. However, it may be the case that an object is re-read during the lifetime of a transaction. This happens for example after a validation operation marks the readset as valid at time t'. Upon a readset validation occurring at time t', all objects within it are checked to see if there has been any concurrent update by a conflicting transaction. If that is not the case, it means that the transactional object r_i was still valid at time t', using t_i as reference time. This is equivalent to saying that the object r_i was re-read at t', since we know that up to that moment there were no conflicts. If this is true for every object in the readset, the latter is valid and so is the transaction. When a new validation is performed, the reference time for each r_i is considered to be the last reading time, i.e., the time at which the last (re-)read occurred. By refreshing the reading time of all the objects in the readset, we abide by the memorylessness property and we make sure that future abort probability calculations performed by the model are correct.

B. Model-exploitation schemes

In this section we show how the proposed analytical model can be exploited to optimize the performance of transactional systems. A typical usage of the model is to put a threshold on the maximum abort probability that can be tolerated at any time in order to allow the transaction to run ahead without being (re-)validated. Once this threshold is exceeded, the transaction must undergo a validation of its readset, which might lead to an early abort.

Clearly, the higher the threshold, the higher the probability of actually experiencing an early abort when the predicted probability exceeds the threshold. In such a case the cost for performing the readset validation operation can actually pay off. At the same time, for very large threshold values, either the number of threshold violations decreases drastically or the gain in terms of saved computation cost decreases.

To motivate the above assertion, let us first consider a transactional system which only validates transactions' readsets at commit time. Many DBMS implementations based on the read-validation scheme actually adopt this approach. The probability of abort as computed by our model in such a scenario clearly increases as the transaction approaches its end, being it a function of the current time t and of the distance (in the past) of the performed read accesses. However, there is little gain in aborting a transaction when it is about to be aborted anyway by its commit-attempt procedure. Hence, it would be better to set a threshold value which allows to invoke an early validation when there is more computation time to save (hence a lower threshold value).

On the other hand, lets us consider the case of a transactional system which already performs spontaneous validations in the middle of a transaction upon a few read accesses to transactional objects, e.g., to provide some correctness criterion stricter than serializability, such as opacity [3]. In this other scenario, setting too high a threshold value for firing validations means performing a model-based proactive validation only in those rare cases when the aforementioned spontaneous validations have failed to catch any abort. Hence,

even if the (predicted and real) probability of abort of the transaction is high at some point in time, the probability of getting to that point itself—and exploiting the model for triggering validations—is actually very low.

Overall, finding an optimal value for the threshold parameter is a challenging task. In this article we report performance data without any manual or automatic determination of the optimal threshold value TV. So we simply explore how performance varies with different statically configured values of TV, as a way to show the potential of our proposal under different configurations. Generally speaking, there are at least two main approaches to finding an optimal threshold value. An off-line methodology is to run the application/workload of interest while collecting some high-level profiles and statistics. Then, the best threshold can be found after processing the collected information. For example, one can compute the actual abort probability distribution and set a threshold which is near to the observed mode parameter of that distribution. Another technique can be based on runtime feedback-oriented optimization. The application is run with an initial threshold value—possibly inferred using static off-line techniques such as the one suggested above—and its performance over time is observed. After some time, the threshold is changed and a new observation period starts. A biased exploration, like the hill-climbing approach, can be used to test new values and decide when to stop. In general, a run-time methodology eventually picks the threshold which provided the best performance among all observed values. This process can be repeated over time to optimize multi-phase applications. Additionally, when the workload is comprised of large and small transactions which exhibit quite different memory access patterns, the threshold can be set separately for each transactional profile. The extreme case is that of enabling the model and setting a threshold only for a subset of the profiles. These approaches will be explored in future work.

IV. Experimental evaluation

In this section we report an analysis of both the model accuracy and the impact of model-based read-validation on performance. This study exploits the TinySTM [1] package, which is a single-version word-based implementation of LSA (Lazy Snapshot Algorithm) for time-based STM systems. It manages transactions by relying on a *global version clock* (*gvc*), a shared counter which is atomically incremented whenever a thread commits a transaction that updates shared data. A data object is a memory word, and each word address is associated with its own meta-data consisting of a lock-bit and a timestamp, both kept in a single entry of a hash array that is manipulated atomically (also called *lock array*). When a transaction successfully commits, the updated *gvc* value is reflected as the new timestamp of the written word. Upon (re)starting a transaction, a thread stores the current value of the *gvc* into a local variable called *transaction start-timestamp* (*tst*). Upon a write operation, the target address and the value to be stored are both added to the transaction writeset. Read operations on shared objects previously updated by the same transaction are served by picking values from the transaction writeset. Instead, read operations performed on shared objects outside the writeset lead to sample the timestamp and the lock bit of the shared object in order to check if (A) the timestamp is less than or equal to the *tst* of the reading transaction, and (B) the object is not currently locked. If both checks succeed, it means that no concurrent transaction has modified the object in the interval between the start of the reading transaction and the actual read operation, hence the read value is *valid*. Otherwise, the transaction gets aborted.

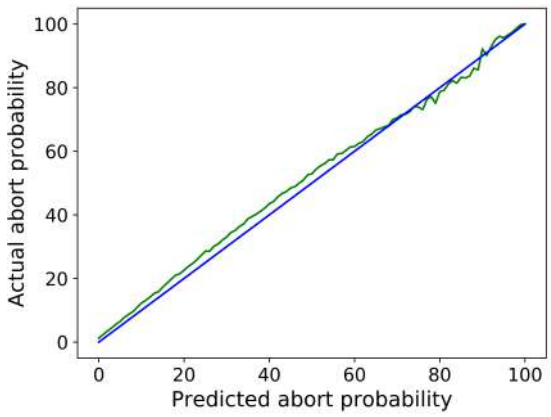
A mechanism that is used in combination with this scheme is called *snapshot extension*. When the thread reads an object whose timestamp is greater than *tst*, this mechanism checks if all the previously executed transactional read operations (if any) are still valid in an extended snapshot that includes the timestamp of the culprit read. If yes, the snapshot seen by the transaction is still consistent and the transaction is not aborted. Additionally, the *tst* is updated to the *gvc* value sampled immediately before performing the check. In such a case, since the abort is avoided, the transaction can continue its execution.

In this scenario, a validation (see points A and B above) is only attempted upon explicit read accesses, when such accesses may result into a violation of the *opacity* correctness criterion. If a thread runs a transaction that does not access shared data for a while (e.g., it manipulates local variables into the stack), or accesses data deemed valid (the object timestamp is within the transaction visibility snapshot), then the underlying STM layer does nothing to detect conflicting accesses. On the other hand, our analytical model can predict the probability that the transaction has become invalid upon performing those accesses, and be used to trigger a model-based proactive validation.

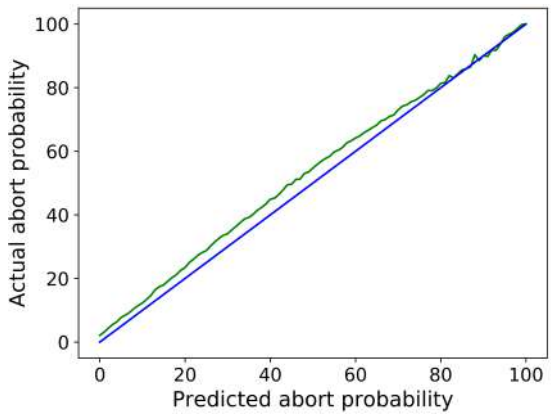
A. Model accuracy

To quantitatively assess the accuracy of our model we have carried out an experiment based on a synthetic benchmark for STM. A shared array in memory constitutes the whole transactional repository at disposal of transactions. Each transaction performs a given number of accesses to this repository, split between read and write accesses hitting the set of objects randomly. The size of the repository, the number of transactional operations and the probability of performing a read operation as the next transactional operation are all user-defined parameters of the synthetic benchmark.

We report the accuracy plots obtained when setting the overall dataset size to 1000 elements, and the number of per-transaction operations to 50. The experiments have been run relying on 24 threads hosted on top of a 32-core HP ProLiant server whose detailed technical description is provided while illustrating the performance study in the next sction. Figures 2a and 2b show how the predicted abort probability compares with the actual abort probability when the transaction read probability parameter is set to respectively 0.9 and 0.8—meaning that 10% or 20% of the data accesses by transactions are in write mode. These scenarios represent a



(a) Accuracy results for the case of 90% reads, 10% writes



(b) Accuracy results for the case of 80% reads, 20% writes

Fig. 2: Model validation plots using a synthetic benchmark. The dataset size is of 1000 elements, each transaction performs 50 operations on random dataset elements. The actual read/write ratio varies in the two experiments.

realistic situation where contention is fairly spread across all the dataset, with a probability of abort that can still be high due to the relatively high number of read operations compared to the overall dataset size. Additionally, it emulates a real-world application scenario where reads tend to be much more frequent than writes. On the x-axis is the abort probability as computed by the model, while on the y-axis is the actual abort probability, derived as the ratio of aborts over total validations for that specific point on the x-axis. To obtain a perfect match, the predicted probability must equal the actual probability, as suggested by the straight, blue line on the plot. The absolute error committed at any point in Figure 2a never exceeds 4%. Compared to the first plot, there is a slight loss in precision in Figure 2b when the number of write operations is increased, with a maximum absolute error that is roughly 5%. Such errors suggest that the accuracy of the proposed model is affected by small-order effects due to numerical stability and possibly by some slight statistical correlations between read and write operations from different transactions. Nevertheless, the data also show that our model is reasonably stable and capable of predicting the abort probability of transactions under the studied settings with negligible precision error.

B. Performance results

We tested the performance optimization based on model predictions, and the comparison of the predicted abort probability values to TV for triggering validations, with a port of the TPC-C [21] benchmark to TinySTM. So the performance study is carried out with a workload profile not fully matching the one for which we reported accuracy data, which makes the assessment of our proposal more meaningful.

TPC-C is representative of OLTP workloads and includes different transaction profiles that simulate a whole-sale supplying items from a set of warehouses to customers within sales districts. In our experiments we instantiated one district, and generated a workload made up by requests spanning four different transaction profiles specified by the benchmark, excluding the "delivery" profile since, according to the TPC-C specification, it is conceived to be run in deferred mode. In our porting to the target STM environment, CPU demands for the different transactional profiles of TPC-C range from tens of microseconds to milliseconds, as shown in Table I, where we also report the percentage mix of the different profiles. It must be noted that of all enabled profiles, "new order" is the only one having a high CPU demand, a relevant share of the whole workload (almost 50%), and a mixture of read and write operations. The "stock level" profile is too a long-running one, but it has a much smaller share of the workload and it is read-only[3]. To emulate a realistic deploy for modern applications, we have run experiments on a cluster of two 64-bit NUMA HP ProLiant servers. The STM application is deployed on one of these nodes—acting as a back-end data layer—while the other node is used for generating the workload of transactional requests. The server node is equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. The client node has two 2.2GHz AMD Opteron 6174 processors and 32GB of RAM. Both processor models have eight NUMA nodes. This type of machine is exactly the one on which we have run the accuracy-assessment experiments presented in the previous section. In all the experiments, threads remain pinned to their NUMA nodes so as to minimize the impact of (possible) thread migrations by the operating system. We don't move data across NUMA nodes and don't change the default allocation policy for dynamic memory.

We have run our experiments with continuous injection of transactional requests, using 24 threads for processing the requests at the back-end data management node and 6 threads for managing the socket pool from which the client-generated workload comes. This scenario led to use at most 94% of the CPU computational power at back-end data management node, thus avoiding hardware resources saturation that would

[3]Read-only transactions don't usually undergo any validation upon their commit. In fact, it is admissible that their readsets contain overwritten data, provided that the readset is consistent in the first place.

ID	Profile	Type	CPU demand	% mix
1	**new order**	RW	$\approx$ 350 μsec	0.49
2	**payment**	RW	$<$ 10 μsec	0.43
3	**order status**	RO	$\approx$ 10 μsec	0.04
4	**stock level**	RO	$\approx$ 650 μsec	0.04

TABLE I: Transaction profiles and associated CPU demand.

affect the reliability of the experimental analysis. In any case, we have run with the highest concurrency admitted by 24 threads since we configured TinySTM to rely on the Commit-Time Locking scheme for data-lock acquisition upon write operations. We set the backlog of pending transactional requests to be processed at the server side to 4096, and we experimented with a sustained workload leading the backlog to be close to saturation at any used thread count. Each experiment entails 3 million committed transactions.

In our experiments we varied the parameter TV using the following values: 100%, 85%, 70%, 55%, 40%, 25% and 10%. Upon a read access to a transactional object at time t performed by transaction x, the inequality $\mathcal{A}_x(t) > TV$ is checked to verify whether the predicted abort probability exceeds the threshold. If such inequality holds, a new validation task is triggered before completing the access. The configuration with $TV = 100\%$ leads our model-based optimization to never fire any model-based validation, since the estimated abort probability of transactions cannot be greater than one. However, this configuration is important in order to assess what is the actual overhead for managing the model at runtime—especially to access the global commit clock value and to keep information related to the update rate of each distinct transactional word in memory. Indeed, we observed that the overhead for handling the model at runtime is essentially caused by conflicting cache-line accesses by concurrent threads to the global commit clock value. Such value must be updated each time a transaction commits in any thread, and it must be accessed when the abort probability value for a running transaction needs to be estimated via the model. Our model-based validation scheme also lacks any estimation of the actual cost for performing a model-based proactive validation, compared to the cost of waiting until the next spontaneous proactive validation. The presence of a cost model would enable us to decide whether to perform an additional validation depending on these estimations, even when the threshold value has been exceeded. Techniques for reducing such architectural impact on performance, and further enhancing the gain provided by the model-based validation scheme will be the target of future research work.

In Figure 3 we report the variation of the transaction turnaround for profile #1 using the different values of TV, relative to the turnaround that can be observed when running the original configuration of TinySTM (not embedding our model-based validation scheme). We will refer to the latter configuration as 'baseline' in our discussion. By the data we see how, although the runtime management of the model introduces about 5% overhead (see the bar for $TV = 100\%$),

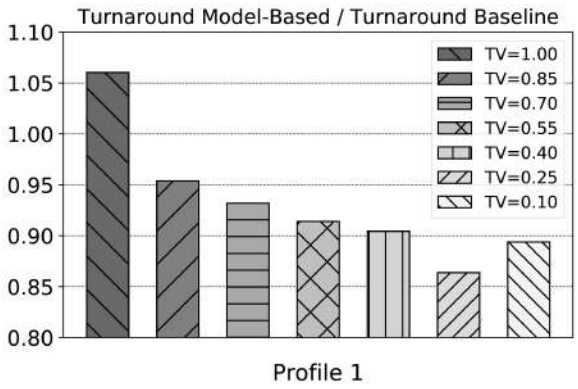


Fig. 3: Turnaround results for profile #1 compared to the baseline.

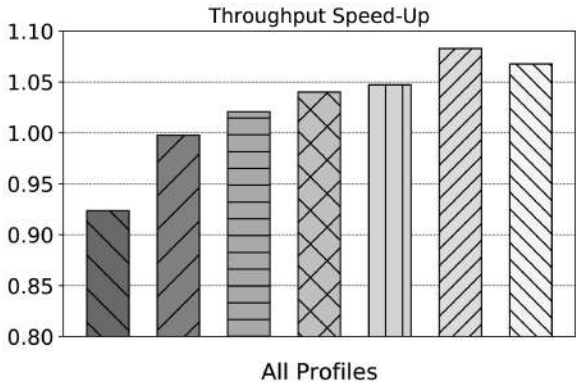


Fig. 4: Throughput results compared to the baseline.

as soon as we also exploit the model we do not only recover the loss of performance caused by the runtime model-management overhead; rather, we also achieve a performance gain over the baseline of about 14%. As already anticipated, we reported data for profile #1 only since profiles #2 and #3 are so short running that no early abort technique allows for actual improvements of their performance. On the other hand, profile #3 and #4 are read only, and TinySTM already applies to these profiles a set of runtime optimizations that stand aside of the model-based read-validation scheme we present.

Even more important, we observe that the performance gain provided by the model-based read-validation scheme is stable for large interval of values of TV. In fact, the throughput observed when the model is active, as shown in Figure 4, tends to degrade towards the one of the baseline only when TV exceeds the value 85%. This is in practice an indication of the effectiveness of the approach even in contexts where no (extremely) fine tuning of the value of TV is adopted. We also note that the gain in the turnaround of profile #1 of TPC-C is reflected into an overall throughput gain of about 8% over the baseline when considering all the transaction profiles. This is clearly linked to the relevance of profile #1 within the TPC-C transaction mix.

V. Summary and Future Work

In this article we have presented a model for estimating the abort probability of a transaction maintaining a readset of all the read-accessed transactional objects. Our proposal has applications to different concurrency control protocols which avoid read-locking data objects, such as optimistic and multi-version ones. Our model can be used to perform proactive validations of transactions, as a means to abort no longer valid transactions as fast as possible. We have shown the accuracy of our model and the performance benefits that can be achieved by exploiting it as a runtime decision model in the contest of Software Transactional Memory (STM) applications. In particular, we have adopted a simple threshold-based mechanism to trigger proactive transaction validations whenever the estimated abort probability of a transaction exceeds a certain value. Thanks to this mechanism, we have observed up to 14% performance gain—on a per transaction-profile analysis—for a port of the TPC-C benchmark to the STM environment. As future work we plan to investigate more sophisticated strategies for exploiting the abort probability predictions by the model in combination with other (predicted) costs—such as the expected residual transaction execution time and the real cost of the validation operation as function of the readset size. We also plan to investigate architectural solutions for reducing the overhead to manage the model at runtime, to further improve the performance of transactional applications.

References

[1] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[2] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. E. T. Rodrigues, "GMU: genuine multiversion update-serializable partial data replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2911–2925, 2016.

[3] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, 2008, pp. 175–184.

[4] "http://infinispan.org/."

[5] P. S. Yu, D. M. Dias, and S. S. Lavenberg, "On the analytical modeling of database concurrency control," *J. ACM*, vol. 40, no. 4, pp. 831–872, 1993.

[6] R. Osman, D. Coulden, and W. J. Knottenbelt, "Performance modelling of concurrency control schemes for relational databases," in *Proceedings of the 20th International Conference on Analytical and Stochastic Modelling Techniques and Applications, ASMTA 2013, Ghent, Belgium, July 8-10, 2013.*, 2013, pp. 337–351.

[7] P. di Sanzo, B. Ciciani, R. Palmieri, F. Quaglia, and P. Romano, "On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking," *Perform. Eval.*, vol. 69, no. 5, pp. 187–205, 2012.

[8] D. Rughetti, P. di Sanzo, B. Ciciani, and F. Quaglia, "Analytical/ml mixed approach for concurrency regulation in software transactional memory," in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, 2014, pp. 81–91. [Online]. Available: http://dx.doi.org/10.1109/CCGrid.2014.118

[9] A. Dragojević and R. Guerraoui, "Predicting the scalability of an stm: A pragmatic approach," in *Presented at the 5th ACM SIGPLAN Workshop on Transactional Computing*, 2010.

[10] P. di Sanzo, M. Sannicandro, B. Ciciani, and F. Quaglia, "Markov chain-based adaptive scheduling in software transactional memory," in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, 2016, pp. 373–382. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2016.104

[11] D. Didona, P. Romano, S. Peluso, and F. Quaglia, "Transactional auto scaler: Elastic scaling of replicated in-memory transactional data grids," *ACM Transactions on Autonomous an Adaptive Systems*, vol. 9, no. 2, pp. 11:1–11:32, 2014.

[12] S. Economo, E. Silvestri, P. di Sanzo, A. Pellegrini, and F. Quaglia, "Prompt application-transparent transaction revalidation in software transactional memory," in *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications, NCA 2017, Cambridge, MA, USA, October 30 - November 1, 2017*, 2017, pp. 157–162.

[13] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2008, pp. 169–178.

[14] S. Dolev, D. Hendler, and A. Suissa, "Car-stm: scheduling-based collision avoidance and resolution for software transactional memory," in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2008, pp. 125–134.

[15] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, "Identifying the optimal level of parallelism in transactional memory applications," *Computing*, vol. 97, no. 9, pp. 939–959, 2015. [Online]. Available: http://dx.doi.org/10.1007/s00607-013-0376-3

[16] P. D. Sanzo, "Analysis, classification and comparison of scheduling techniques for software transactional memories," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3356–3373, dec 2017.

[17] Q. Wang, S. Kulkarni, J. V. Cavazos, and M. Spear, "Towards applying machine learning to adaptive transactional memory," in *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

[18] D. Didona, N. Diegues, A. Kermarrec, R. Guerraoui, R. Neves, and P. Romano, "Proteustm: Abstraction meets performance in transactional memory," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, 2016, pp. 757–771. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872385

[19] M. Castro, L. F. W. Goes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Mehaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *Proceedings of the 18th International Conference on High Performance Computing*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–10.

[20] A. Porfirio, A. Pellegrini, P. di Sanzo, and F. Quaglia, "Transparent support for partial rollback in software transactional memories," in *Proceedings of the 19th International Conference on Parallel Processing, Euro-Par 2013, Aachen, Germany, August 26-30, 2013.* , 2013, pp. 583–594.

[21] TPC Council, "TPC-C Benchmark, Revision 5.11," Feb. 2010.