Exploiting Symbolic Execution to Accelerate Deterministic Databases

Shady Issa
U. Lisboa/INESC-ID
shadi.issa@tecnico.ulisboa.pt

Miguel Viegas
U. Lisboa/INESC-ID
miguel.viegas@tecnico.ulisboa.pt

Pedro Raminhas
OutSystems, Portugal
pedro.raminhas@tecnico.ulisboa.pt

Nuno Machado
Teradata, Spain
nuno.machado@teradata.com

Miguel Matos
U. Lisboa/INESC-ID
miguel.marques.matos@tecnico.ulisboa.pt

Paolo Romano
U. Lisboa/INESC-ID
paolo.romano@tecnico.ulisboa.pt

***Abstract*—Deterministic databases (DDs) are a promising approach for replicating data across different replicas. A fundamental component of DDs is a deterministic concurrency control algorithm that, given a set of transactions in a specific order, guarantees that their execution always results in the same serial order. State of the art approaches either rely on single threaded execution or on the knowledge of read- and write-sets of transactions to achieve this goal. The former yields poor performance in the multi-core era while the latter is achieved either via manual inputs from the user — a time-consuming and error prone task — or by relying on a *reconnaissance phase* that increases both the latency and abort rates of transactions.**

In this paper, we present Prognosticator, a novel deterministic database system. Rather than relying on manual transaction classification or a expert programmer to predict transaction accesses, Prognosticator automates the process through Symbolic Execution. This allows Prognosticator to build a fine-grained knowledge (at the key-level) of transactions accesses which is then used by our novel deterministic concurrency control algorithm to achieve a high degree of parallelism in transaction execution. Our experimental evaluation, based on both TPC-C and RUBiS benchmarks shows that Prognosticator can achieve up to 5× higher throughput, when compared to state of the art solutions.

I. Introduction

Deterministic databases (DDs) [1], [33] are an appealing approach to build replicated data management systems. DDs are typically organized in two logical layers: a consensus layer that delivers batches of transactions in the same order to the replicas, and a transaction processing layer responsible for executing the transaction logic. Modern consensus implementations leverage a series of techniques (e.g. batching schemes [32] and/or throughput-optimized/ring-based protocols [25])) to achieve high throughput at the expense of a small user latency and thus throughput is typically limited by the transaction processing layer.

Upon receiving a transaction batch, the transaction processing layer is responsible for executing the transactions in a deterministic fashion. Assuming the transaction logic is deterministic, a trivial approach to ensure correctness is to execute the batch of transactions sequentially in a single-thread. Naturally, this severely limits performance and fails to take advantage of modern multi-core processors. This limitation can be overcome by employing deterministic concurrency control algorithms [34], which ensure that, even though transactions are executed concurrently, they are serialized in an order *equivalent* to the order specifed by consensus. Unfortunately, building an efficient deterministic concurrency control algorithms is far from simple. A typical approach [26], [35] consists in *predicting* the set of data accesses that a transaction is going to read or write, i.e. the read- and write-sets (RWS), and uses this information to acquire the set of locks they will require prior to the their execution in a deterministic fashion using a single thread.

In a system such as NODO [26] the RWS are given by the set of *tables* each transaction accesses. While being very simple to implement, this approach leads to poor parallelism as it fails to exploit the fact that two transactions that access different records in the same table can, in fact, be executed in parallel. Calvin [35] takes a different approach and delegates determining the RWS to the programmer which must specify, for each transaction, the set of tuples that it is going to access. This has the potential to yield fine-grained RWS but it requires that programmers reason about *all* possible accesses of a transaction which is impractical and error prone for all but toy applications. As an example, which we will detail further in Section III-B, the *delivery* transaction of TPC-C [36] has 1024 RWS combinations depending on the input. Clearly, building this manually is very time consuming and error prone. Either the programmer is too optimistic and commits a mistake, compromising correctness, or she is too conservative and provides a coarse grained overestimation, hence reducing the potential parallelism. Nonetheless, even if we assume an expert programmer who is able to precisely specify the RWS, there are certain transactions where the RWS depends on the state of the database. As an example, a transaction could read an item from the database and depending on the value read, write in one of two different keys. As in Calvin, we call these transactions *dependent transactions* (DT), and will further discuss them in Section III-B. To overcome the complexity of manually determining the RWS, and to support DT, Calvin relies on a protocol called OLLP [34], which provides a *reconnaissance phase* mechanism where clients can pre-execute the transaction to obtain the set of data items that are accessed. Then, the client submits the transaction to the system, along with the *predicted* RWS. Note that this is only an estimation as the database state can change between the *reconnaissance phase* and the actual transaction execution. This not only results in wasted resources, as transactions are executed at least twice, but also leads to high-abort rate in write intensive workloads where the database state is more likely to change between the *reconnaissance phase* and transaction execution.

In this paper, we propose Prognosticator, a deterministic database system that addresses these limitations as follows. Rather than relying on programmer expertise to manually specify the RWS, we propose a fully automated transaction analysis, based on Symbolic Execution (SE) that determines the RWS for all but DT. Briefly, SE is a static program analysis technique to determine what inputs allow which parts of the program to be reached and has been successfully used for the verification of complex systems [2]. The key insight of Prognosticator is to use the exhaustive search capabilities of SE to determine for any given transaction: *(i)* all possible execution paths and the respective set of data items accessed in each path, *(ii)* build a tree — which we call the transaction profile — that, for a given transaction input, determines the concrete set of data items that are accessed. This eliminates programmer mistakes and allows for highly concurrent transaction execution. Regarding DTs, we use SE to obtain the fraction of the RWS that does not depend on the database state. For the RWS that depends on the database state, we still need to query the data store to *estimate* the RWS. However, our approach differs from Calvin, in two fundamental ways. First, because we have a partial transaction profile, the *reconaissance phase* needs to be done only for part of the transaction which reduces database load. Second, the *reconaissance phase* is done on the server rather than on the client. This has the effect of removing the consensus execution from the "vulnerability window" between the *reconaissance phase* and the actual transaction execution, hence minimizing the chance of an outdated prediction. Note that this requires the *reconaissance phase* to be deterministic such that all servers do the same prediction. Moreover, in case of an outdated prediction, rather than aborting the transaction and delegating the re-execution of the *reconnaissance phase* to the client, as in Calvin, we re-execute the transaction immediately on the server.

To ensure determinism in transaction execution, DDs such as Calvin rely on a single threaded scheduler and multiple worker threads for transaction execution. The single threaded scheduler ensures determinism in the lock acquisition across all replicas but bottlenecks system throughput. In Prognosticator we have several mechanisms to tackle this issue. First, worker threads are used in the *reconaissance phase* to predict the data accesses for DTs relieving the scheduler thread from this work. Second, worker threads operate in two phases: execution of read-only transactions and execution of update transactions. This allows to execute read-only transactions in a lock-less fashion, allowing Prognosticator to scale in read-dominated workloads, a limitation of state of the art DD highlighted in a recent study [12]. Further, it permits overlapping the work done by the *Queuer Thread* with the execution of read-only transactions. Finally, whenever a worker thread is throttled down (as the *Queuer Thread* has become a bottleneck for the system) and becomes idle, it can help the *Queuer Thread* by acquiring locks for the next batch of transactions, if any.

Algorithm 1 Sample code for SE analysis

```
1: function EXAMPLE(input)
2:     x ← input
3:     z ← x + 1
4:     if x $>$ 10 then
5:         z ← z + 9
6:     else
7:         z ← z + 19
8:     end if
```

In summary, this paper makes the following contributions:

- a novel SE-based technique that automatically analyzes an application and builds the transaction profiles,
- a novel deterministic concurrency control algorithm that leverages the transaction profiles, which scales across cores and reduces transaction abort rates, and
- the implementation and thorough evaluation of Prognosticator, showing up to 5× higher throughput compared to state of the art solutions across a wide range of TPC-C [36] and RUBiS [5] benchmarks[1].

The rest of this paper is organized as follows. In § II we provide background on Symbolic Execution. In § III we discuss the system model, and present the design of Prognosticator, which we evaluate in § IV. Related work is discussed in § V and finally, § VI concludes the paper and discusses future work.

II. Symbolic Execution

Symbolic execution (SE) is a static program-analysis technique pioneered by King [15] that explores multiple execution paths of a particular program based on its input. SE relies on a *symbolic execution engine* [29], [4], which is a special interpreter that allows program variables to have symbolic values rather than concrete ones. To this end, during program execution, the SE engine maintains a *symbolic state*, which is described as the conjunction of two formulae: a *symbolic store* and a *path constraint*.

The symbolic store can be defined as a function σ that maps a set of variables to symbolic expressions according to the instructions executed. For example, let us consider the simple program in Algorithm 1 that contains an integer variable x that is initialized with an input value passed by the user. When the program is executed symbolically, x will be assigned a symbol α regardless of the user input, hence $\sigma : x \mapsto \alpha$. After the program executes the statement `z = x + 1` in line 3, the symbolic store will be updated to $\sigma : x \mapsto \alpha, z \mapsto \alpha + 1$.

The path constraint keeps track of the program's control-flow by encoding the conjunction of the branch conditions

[1] The source code will be made available online when the paper is accepted.

taken up to a given point in the execution. Therefore, the path constraint is typically defined as a logical formula ϕ without quantifiers. Whenever the SE engine encounters a conditional statement containing symbolic values, it forks the symbolic state such that both branches can be explored. For example, upon reaching the conditional branch in line 4 `if(x > 10)`, the symbolic state above is forked into $\{\sigma : x \mapsto \alpha, z \mapsto \alpha + 1\ ; \phi = \alpha > 10\}$ and $\{\sigma : x \mapsto \alpha, z \mapsto \alpha + 1\ ; \phi = \alpha \leq 10\}$. Note that the evaluation of the conditional statement affects the path constraint, but not the symbolic store. After evaluating both paths of the branch, the symbolic state becomes $\{\sigma : x \mapsto \alpha, z \mapsto \alpha + 10\ ; \phi = \alpha > 10\}$ and $\{\sigma : x \mapsto \alpha, z \mapsto \alpha + 20\ ; \phi = \alpha \leq 10\}$

To ensure that only feasible paths are explored, the SE engine resorts to a constraint solver [8], [9] to assess the satisfiability of every path constraint. If a path constraint is unsatisfiable, then the corresponding symbolic state is discarded. Despite that, SE still suffers from *path explosion*, as the number of paths to explore grows exponentially with the number of branches in the code (which is particularly problematic in the presence of loops). Addressing path explosion generally falls into two approaches: restrict the input and/or code expressiveness (e.g. by bounding loops), or employ heuristics to guide state exploration towards a given criteria (e.g. maximize line coverage).

III. Prognosticator

In this section, we detail Prognosticator design and architecture. We start with an overview of the different modules, and then we discuss the design of each module in detail.

A. Overview

Prognosticator is a deterministic database execution layer that is designed with the following goal: given a batch of transactions with a specific execution order, Prognosticator maximizes transaction execution parallelism while ensuring the same serialization order across different replicas. Prognosticator assumes that data is not partitioned across replicas, i.e., each replica contains all the data state.and supports store procedures transactions that are written in Java, which query a data store through read and update interface. Prognosticator achieves this goal by determining the transaction profiles, i.e. the read- and write-sets (RWS) of each transaction, without any manual inputs from programmers. This is performed in two steps. In an offline step, Prognosticator leverages Symbolic Execution (SE) to automatically build a transaction profile that encodes, for every possible transaction input, the respective RWS. This is obtained by leveraging the SE symbolic state as discussed in Section II. The pre-built transaction profile is then used at runtime to schedule transaction execution in such a way that non-conflicting transactions — i.e. those whose RWS are disjoint — can be executed in parallel without requiring any coordination among replicas.

Figure 1 depicts Prognosticator's architecture, which is composed by two macro-modules: *(i)* the Client Application modules that receive and batch transactions and *(ii)* System Replicas that replicate the data state and consume requests provided by the client in order to read or update data. The number of Client Application modules and System Replicas do not have to be equivalent as they are independent.

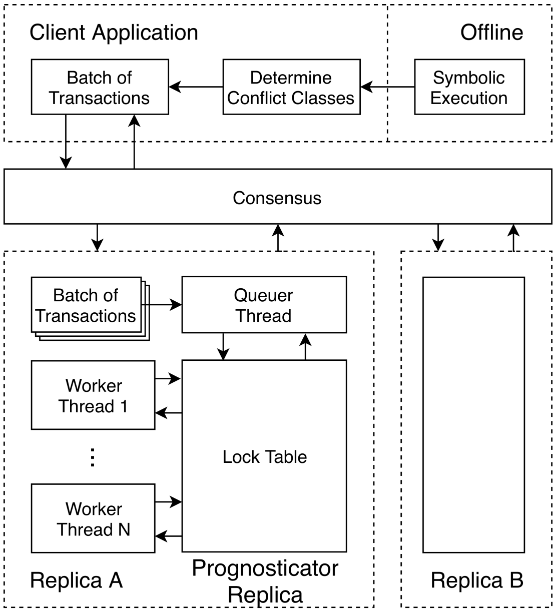


Fig. 1: Architecture of Prognosticator system

The Client application is composed by two sub-modules: *(i)* a run-time Client Request Dispatcher and *(ii)* an offline SE Engine. The Client Request Dispatcher receives transactions from external clients and is responsible for generating batches of transaction requests alongside the input parameters to which transaction requests are instantiated with and deliver those batches to the replicas to be executed. Before sending the transactions to the replicas, different clients must agree on the order of transactions within each batch. This is accomplished by relying on a consensus algorithm [17], [24].

The SE Engine is invoked by the Application Client, one time and offline, to analyze the application code and provide the transaction profiles. As described earlier, transaction profiles contain information on the execution paths and the RWS associated with each execution path. At runtime the Client Request Dispatcher sends the transaction requests enriched with this information to the System Replicas.

System Replicas are composed by a *Queuer Thread*, a set of *Worker Threads* and a *lock table* data structure. These three components work together in synergy to implement a multi-threaded deterministic concurrency control mechanism. They leverage the transactional profiles to execute non-conflicting transactions concurrently without violating the serialization order defined by the clients.

In the following sections, we will explain how Prognosticator uses SE to generate fine-grained transactions profiles then detail the algorithm employed by the *Queuer* and *Worker* threads that leverage these profiles.

B. Transaction Profiling

Overview. Prognosticator relies on JPF [29] with the Symbolic JPF extension [22] to symbolically execute the code associated with each transaction and compute its *profile*. Informally, the profile of a transaction defines the sets of data items read and written by a transaction for any of its possible execution paths.

More precisely and formally, a transaction profile is a set of N pairs $< PSC_i, RWS_i >$, with $i \in [1, N]$, where PSC_i (*Path-Set Condition*) identifies a partition of the execution paths, and RWS_i defines the set of items read and written by the transaction when its code executes along any of the paths identified by PSC_i. Since we want to be able to predict the data to be accessed by a transaction for all of its possible execution paths, the PSCs in a transaction profile define disjoint partitions of the whole set of execution paths, i.e., every execution path is included in exactly one PSC_i.

In the most general case, both the execution paths of a transaction (defined via PSC) and its accessed data items (defined via RWS) can be affected by two factors: (i) the input value with which the transaction code is invoked, noted $\mathcal{I}$, and/or (ii) the value of one or more data items which are read during transaction execution, which we call *pivot* items and note as $\mathcal{P}$. Through the use of SE, Prognosticator defines PSC and RWS via symbolic expressions that are functions of $\mathcal{I}$ and $\mathcal{P}$. In the following, if a symbolic expression is solely a function of the transaction's input ($\mathcal{I}$) we say that it is *direct*, else we say that it is *indirect*.

Prognosticator encodes the set of PSCs of a profile by means of a tree, where each node of the tree specifies a logical condition that is function of $\mathcal{I}$ and/or $\mathcal{P}$. This design decision allows to efficiently merge the logical conditions of execution paths that produce the same database accesses, thus allowing to produce a compact representation of the transaction profile that can be consulted efficiently at run-time.

Determining the RWS of an execution path. The SE engine of JPF provides a listener-based API, which allows for intercepting relevant events, such as the execution of conditional statements, method invocation and assignments of symbolic variables. In order to infer the RWS of a given execution path we implemented a custom listener which is notified whenever a method for reading from or writing to a data item is invoked. Our current implementation assumes a key/value data model with a classic GET/PUT interface[2] that allows reading/writing individual keys. Whenever one of these methods is invoked, we extract the symbolic formula that identifies the key read/written and add it to the read-set/write-set, respectively, of the path being currently analyzed. In order to identify the pivots, at this stage we also check whether the symbolic formula that encodes the identity of the item to be accessed depends, either directly — e.g., GET(GET($inputVar$)) — or indirectly — e.g., $y \leftarrow$ GET($inputVar$); GET(y)) — on a value that has to be retrieved from the database — e.g., in the two previous examples, the value of the item identified via $inputVar$. Pivots can affect not only the composition of the read and write set of transactions, but also their execution paths. This happens whenever a conditional statement depends on the value of some item maintained in the data store. To track these pivots, whenever a conditional branch is encountered, we check whether the corresponding symbolic formula (which is to be added to the path constraint) is indirect, in which case we extract any pivots on which the formula depends.

Exploring and merging execution paths. The approach we use for controlling the SE engine is designed to achieve two main goals: *(i)* reducing the memory footprint of the SE analysis and *(ii)* generating the transaction profile in a format such that it can be efficiently parsed to define the conflict classes of a transaction. For the former, we rely on a depth first exploration of the set of execution paths, which allows us to discard redundant states as soon as possible. For the latter, we adopt a tree data structure, where its nodes describe path conditions and the symbolic store of the key-set collected from the execution path between the node's path condition and the subsequent conditional statement.

More in detail, the set of execution paths is explored by executing symbolically the code of a transaction and collecting the identities of any data item read and written until the next conditional statement is reached. This information is stored within a node, along with the logical condition of corresponding conditional statement expressed in symbolic form. Each node that has two sub-trees corresponding to the different outcomes of the conditional statement and, as already mentioned, we visit (i.e., execute symbolically) them in depth first order. When we reach the end of the program for the first time, we have established the path constraint and RWS of the first execution path. Next, we back-track the execution to the last conditional statement and execute symbolically the other branch. At this point, if we detect that the two executions produced the same RWS, the left and right branches are pruned and their RWSs are added to the ones of the parent node. In this case, in fact, the conditional statement that caused the branching ended up not affecting in the same way the set of accessed data items. This depth first approach allows us to reduce the memory footprint of the SE analysis, by eliminating this sort of "redundant" paths in a timely fashion. Note that, had we followed a breadth first approach, we would have had to explore all the states before pruning the redundant ones.

The resulting tree encodes the transaction profile in a way that is both compact and efficient to query at run-time. Each distinct path in this tree encodes a different tuple $< PSC_i, RWS_i >$ of the transaction profile and can be used to predict the RWS of a (possibly very large) number of distinct execution paths. Also, at query time, the identification of which PSC to use to predict the accesses of an incoming transaction can be executed in logarithmic time (with the number of PSCs in the transaction profile).

Algorithm 2 illustrates a simple example based on a simpli-

[2] Methods for GET/PUT are defined in a configuration file, which does not need to be compiled and therefore can be easily adopted to different data stores' APIs

Algorithm 2 Pseudocode for the *newOrder* transaction of the TPC-C benchmark

```
 1: transaction NEWORDER(districtID, olCnt, olIid, olIquantity[])
 2:     districtInfo=GET(districtID);
 3:     districtId=districtInfo.lastOrderId++;
 4:     PUT(districtID,districtInfo);  // update id of last order on db
 5:     order $\leftarrow$ newOrder()
 6:     PUT(orderId, order)
 7:     olNum $\leftarrow$ olCnt - 1
 8:     while olNumber $\geq$ 0 do
 9:         itemId $\leftarrow$ olIid[olNumber]
10:         item $\leftarrow$ GET(itemId)
11:         olQuantity $\leftarrow$ olIquantity[olNumber]
12:         if item.quantity - olQuantity $\geq$ 10 then
13:             item.quantity $\leftarrow$ item.quantity - olQuantity
14:         else
15:             item.quantity $\leftarrow$ item.quantity - olQuantity + 91
16:         end if
17:         PUT(itemId, item.quantity)
18:         olNumber $\leftarrow$ olNumber - 1
19:     end while
```

fied version of the *newOrder* transaction of the TPC-C benchmark, which contains a for loop with number of iterations equal to the input $olCnt$. The value of $olCnt$ ranges between 5 and 15 according to the benchmark's specification and this information is used during symbolic execution to bound the possible values for this input variable, which is marked as symbolic (as all inputs). The loop contains a single conditional statement at Line 12, which generates a total of 2^{olCnt} distinct execution paths. Both branches, though, generate the same RWS, since the conditional statement only affects the value being written by the PUT at Line 17. In this simple example, thus, although a total of 2^{olCnt} different execution paths are explored, all of them yield the same RWS. As such, the resulting tree collapses into a single node.

Avoiding irrelevant paths. An optimization that we have found to be critical for avoiding state explosion issues (at least with the code that we analyzed) is to identify via a preliminary static code analysis phase a set of, what we call, *irrelevant* variables. These are variables for which we can statically guarantee (using well-known static analysis toolkits like Soot [37]) that there is no explicit (i.e., via assignment) or implicit (i.e., via control flow) information flow [31] to any of the variables that affect the RWS, i.e., the variables that are used in some GET/PUT statement to identify which data item should be read/written. During symbolic execution, we mark the irrelevant variables as concrete, i.e., we assign them an actual value instead of a symbolic one. The resulting (concolic) execution has an important advantage: whenever a conditional statement is encountered, if it only depends on concrete variables, only one execution path (the one satisfied by the concrete variables) is pursued, thus reducing the number of paths to be analyzed during symbolic execution.

Returning to the example of the the *newOrder* transaction (Algorithm 2), we note that the variables item.quantity and olQuantity can be statically identified as irrelevant, since they affect only the value written into the key updated key at Line 17 and not the key's identity. Thanks to the above optimization, only one of the two branches of the if statement at Line 12 has to be executed symbolically. As a consequence, the number of execution paths that has to be executed symbolically decreases from 2^{olCnt} to just 1.

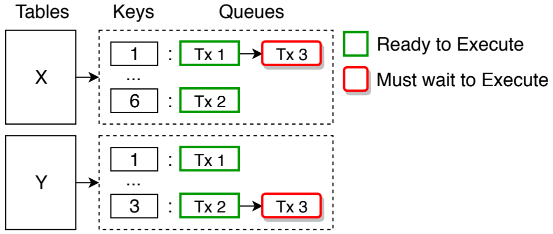


Fig. 2: Lock Table

It is worth noting that our pruning technique might not always yield a substantial reduction in the number of states to be explored. Therefore, to handle state explosion, we cap the SE analysis time and as a result, we may end up with an incomplete transaction profile. To ensure correctness, we rely on executing a *reconnaissance phase* to infer the key-sets of such transactions. Note though that the *reconnaissance phase* only happens for the part of the transaction that was not analyzed rather than the full transaction as it done in Calvin.

C. Transaction Execution

When a client receives a transaction and the associated set of inputs, it consults the offline generated transaction profile to define the key-set, or the set data items that will be accessed throughout the transaction. This step simply entails traversing the tree of path conditions generated by the transaction profiling phase (as explained in the previous section) to determine which data items will be accessed by the transaction. Prognosticator operates in batches where each client collects a set of transactions within a certain time window. After each client has collected transactions for a given batch, it synchronizes with the other clients in order to agree on a total order for executing the transactions within the batch. This agreement can be reached by running a consensus algorithm (such as Paxos [17] or Raft [24]) between the clients.

To ensure correct replication of the data across all the replicas, the state after each batch should be the same on all the non-faulty replicas. To satisfy this condition, Prognosticator leverages a deterministic multi-threaded concurrency control algorithm which achieves twofold goal: *(i)* guaranteeing that transactions are executed following the same serialization order across all replicas and *(ii)* executing transactions concurrently, utilizing the abundance of multicore architectures available in processors nowadays. The latter, specifically, exploits the fact that non-conflicting transactions, i.e., transactions that do not modify the same data [3] can be safely re-ordered while still maintaining the same serialization order.

At the core of Prognosticator's deterministic multi-threaded concurrency control algorithm, there is a *lock table* data structure. As shown in Figure 2, *lock table* is a set of queues,

[3] Prognosticator assumes a key granularity for conflict detection.

each identified by a table and a key within that table. The idea at the basis of *lock table* is to enqueue each transaction in the queues of all keys from its key-sets following the order agreed upon by the clients. Transactions that are at the head of all their queues (such as Tx_1 and Tx_2 in Figure 2) are guaranteed not to conflict with each other and therefore it is safe to execute them concurrently. Note that conflicting transactions must be serialized in the same queue. After the execution of a transaction, it can be removed from the *lock table* and new transactions can be executed once they are at the head of all their respective queues. For example, Tx_3 will be executed only after both Tx_1 and Tx_2 have executed.

Prognosticator differentiates between three types of transactions: *(i)* read-only transactions (ROT), i.e., transactions that do not modify the data, *(ii)* independent transactions (IT), which are transactions whose read- and write-sets depend only on the transaction's input and *(iii)* dependent transactions (DT), which are transactions whose key-sets depend on the state of the data, i.e., they include *indirect keys*.

We will first start by explaining *Queuer Thread*, which is responsible for populating the *lock table* with the transactions following the order given by the client. ROTs do not modify the state of the database, therefore they do not need to be enqueued into the *lock table*, as long as they are guaranteed to witness a consistent snapshot of the data whenever they are executed. This can be achieved in several ways, for example with a data store that support multi-version, ROTs can be set to read from the state generated by the previous batch. In our implementation of Prognosticator, however, we opt for a more generic solution to support a wider range of data stores: ROTs are enqueued — in a round robin fashion — into special, per worker thread, set of queues. We will describe shortly how do the worker threads consume ROTs in their local queues without requiring any lock acquisition. Such a design, alleviates the single *Queuer Thread* being a bottleneck for the scalability of ROTs, which is an issue that was reported in [12].

ITs and DTs are both enqueued into the *lock table*. However, there exists a subtle difference on how the *Queuer Thread* handles each of them. ITs are supplied by the client along with the set of keys that they are going to access (recall that the key-set relies only the input of the transaction in case of IT). In Prognosticator, we use a single *Queuer Thread*, therefore it is straightforward to ensure that ITs are enqueued into *lock table* following the order agreed upon by the clients. The key-set of DTs, however, depends on the state of the data, i.e., the *Queuer Thread* must consult the data store to get the value of the pivot keys to be able to determine the set of queues into which each DT will be enqueued. This raises the challenge of ensuring that this phase, which we call *preparing indirect keys*, results in the same key-set for each transaction across different replicas. By preparing the indirect keys at the beginning of the batch, after the previous batch had finished executing, and before executing any update transaction from the current batch, the *Queuer Thread* ensures a consistent snapshot across different replicas. This design decision, specifically, allows for relying on as fresh as possible snapshot to prepare the indirect keys, which in turn reduces the probability of the IT aborting due to the data being at a different state during the execution phase. Moreover, the *Queuer Thread* enqueues DTs ahead of ITs in the *lock table* so that they get executed earlier to further reduce the likelihood of abort.

Unlike a single *Queuer Thread*, Prognosticator employs multiple *Worker Threads*, which operate in phases to execute transactions concurrently. When a batch is ready, *Worker Threads* start consuming the available ROTs in their local respective queues. During this phase, *Worker Threads* access disjoint data structures and do not have to coordinate while executing ROTs as they are guaranteed to witness the same state and re-ordering of ROTs does not affect the final state of the replica after each batch. After a *Worker Thread* finishes executing its ROTs, it waits for both all other *Worker Threads* to finish executing their ROTs and for the *Queuer Thread* to signal that it has finished *preparing indirect keys*. Next, starts the phase of executing update transactions.

Instead of directly accessing the *lock table*, Prognosticator uses an auxiliary queue data structure called *ready queue*. After the *Queuer Thread* enqueues a transaction into the *lock table*, if that transaction is at the head of all of its queues, it would also be enqueued into the *ready queue*. To support this, for each transaction, there is a *total locks* variable which denotes the number of keys it is going to access. *total locks* is decremented by the *Queuer Thread* in case a transaction is at the head of its queue. A transaction with *total locks* set to 0, is enqueued into *ready queue*. When executing update transactions, *Worker Threads* start consuming transactions form the *ready queue*. Note that transactions in the *ready queue* do not conflict with each other, therefore it is safe for them to be executed concurrently. *Worker Threads* directly execute ITs that they fetch from the *ready queue* as they are guaranteed not to fail. Before executing a DT, however, *Worker Threads* need to first ensure that the keys that will be accessed throughout the transaction did not change since the *preparing indirect keys* phase. Accordingly, when executing an DT, *Worker Threads* first access the data store to check if all the values of the pivot keys have not changed. If it is the case, then it is safe to proceed with executing the DT. Otherwise, the DT must be aborted and is added to a list of failed transactions. In case of a successful execution of DT or IT, the respective *Worker Thread* accesses the *lock table*, removing the successful transaction from the head of its queues. Besides doing that, it also decrements the *total locks* of the next transaction in each queue, and in case it encounters a *total locks* with value of 0, it enqueues that transaction into *ready queue*. It is worth noting here that all access to the *lock table* are done in a lock-free manner as there exists no logical contention between *Worker Threads* and *Queuer Thread*.

Worker Threads wait for each other until they all finish executing the update transactions. Next, starts the phase of re-executing failed transactions. Failed transactions must be re-executed in the same order across different replicas. A simple solution is to re-execute the failed transactions sequentially using a single thread following their relative order from the order agreed upon by the clients. Although this solution would ensure that these transactions would not fail again, it sacrifices potential concurrency among non-conflicting transactions. Prognosticator resorts to re-enqueueing the failed transactions into the *lock table* following the same approach described above. Indirect keys are first re-prepared, then transactions are re-inserted into the *lock table* and *ready queue* to be consumed by *Worker Threads* accordingly. This cycle is repeated as long as their exists failed transactions after each iteration. In the evaluation section, we show that both strategies are beneficial in different workloads.

Optimizations. Data stores with high access latency can impose a bottleneck on the latency of executing a batch due to the time it takes to prepare indirect keys. To overcome this bottleneck, in Prognosticator *Worker Threads* can help the *Queuer Thread* in preparing indirect keys. Specifically, once *Worker Thread* finishes executing all its ROTs, it can start fetching DTs that are being prepared and access the data store to collect the values needed to generate their key-sets. As we shall discuss in our evaluation, this optimization can help to reduce the overall transactions latency.

An additional optimization that could be applied to enhance the efficiency of the *Queuer Thread* (but that has not been implemented in our prototype, yet) is to partially offload to the clients the prediction of the RWS. As described in the Section III-B, the SE engine produces a tree with path conditions and their respective RWS, including the set of indirect keys that are required in case of an DT. We can differentiate between two types of DTs: one where the indirect keys are not necessary to traverse the tree (i.e., traversing the tree depends only on the transaction's inputs, e.g., new order transaction in TPC-C) and one where the indirect keys are needed to traverse the tree (as they are used in the predicate of a conditional statement that affects the transaction's RWS). For the former, the tree that encodes the transaction profile does not need to be traversed by the *Queuer Thread*, as the client could directly predict the set of keys that the transaction will access based on its input values — thus alleviating the load on the *Queuer Thread*.

IV. Evaluation

In this section, we evaluate Prognosticator along several dimensions. We start by studying the cost of the SE analysis and the benefits of the techniques we propose (§IV-A), next we compare Prognosticator with Calvin [35] and NODO [26] under a variety of workloads (§IV-B), and finally study the impact of Prognosticator optimizations on transaction schedulling and execution (§IV-C). We selected NODO and Calvin to compare Prognosticator with, because they represent two interesting points in the design spectrum: NODO is very simple but produces coarse-grained transaction profiles, while Calvin is more complex but achieves fine-grained transaction profiles thus allowing us to infer the trade-offs between design complexity and potential parallelism.

For the benchmarks we used TPC-C [36], which emulates a OLTP workload for a wholesale suplier, and RUBiS [5] which emulates a bidding website. TPC-C consists of two ROT, two DT and one IT which perform operations across different warehouses. The number of warehouses determines the level of contention and therefore the maximum degree of parallelism. RUBiS consists of 28 transactions and in our evaluation we focus only on the 5 update transactions which are all DT. We implement Prognosticator on top of RocksDB [11], a persistent key-value store implemented in C++ with a Java API. Our experiments were conducted on an Intel Xeon Gold 6138 machine with 20 cores running Ubuntu 18.04 with Linux 4.18. For all experiments, we execute 10 runs, discard the first 3 as warm up and present the average of the remaining 7.

A. Symbolic Execution Analysis

Table I summarizes the results of the SE analysis of all the update transactions for both benchmarks. For each transaction, we report the number of states explored, the depth of an execution path (i.e. maximum number of conditional statements a transaction can observe), the number of distinct key-sets collected, the number of indirect keys, memory used and the execution time. For the states explored, we split it in explored states and total states. The latter corresponds to the total number of possible states that one would have to explore if not for the prunning and DFS techniques detailed in §III. This is particular relevant for transactions such as *newOrder* which suffer from a state explosion problem with the number of loop iterations, as detailed in §III-B. To illustrate this, in Table I we detal the results for 5, 10 and 15 iterations. The depth optimized/max, memory optimized/unoptimized and execution time optimized/unoptimized columns in Table I correspond to the SE analysis done with or without the techniques discussed above. This illustrates two important points of our fully-automated approach: first it is infeasible for a programmer to reason about the very large number of states of transactions such as TPC-C *delivery*, and secondly, without the proposed optimizations the running time of SE is impractical taking, for instance, ~ 35 days for *newOrder*.

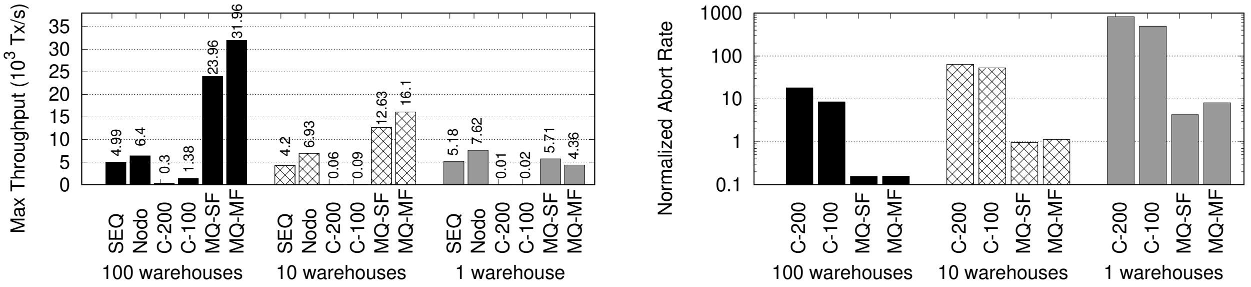
In conclusion, and for the selected benchmarks, the SE analysis finished in less than 2 seconds and 1211MB, which highlights the practicality of our approach.

B. Comparison with NODO and Calvin

To achieve a fair comparison between Prognosticator, NODO and Calvin, we implemented all approaches in the same code base which minimizes the technological and implementation aspects, and allow us to focus on the design trade-offs. In particular, NODO and Calvin benefit from our SE analysis to determine the transaction profiles hence allowing us to assume an ideal programmer that can precisely build such profiles, and also rely on the same *lock table* data structure for scheduling transactions. Therefore, the measured differences below correspond to the design decision of how to leverage the transaction profiles to schedule transactions. Next, we describe the configuration of each system.

Transaction Profile	States Explored / Total	Depth Optimized / Max	Unique Key-sets	Indirect Keys	Memory (MB) Optimized/Unoptimized	Execution (s) Optimized/Unoptimized
TPC-C: *new order (5 iters.)*	2 / 2048	2 / 12	1	1	960 / 1713	<1 / 2
TPC-C: *new order (10 iters.)*	2 / 2097152	2 / 22	1	1	960 / 4696	<1 / 2230
TPC-C: *new order (15 iters.)*	2 / 2147483648	2 / 32	1	1	960 / 21844	<1 / 3066800
TPC-C: *payment*	2	2	1	0	960	<1
TPC-C: *delivery*	2048	12	1024	20	1211	2
RUBiS: *store bid*	28	15	2	1	960	<1
RUBiS: *store buy now*	14	8	2	1	960	<1
RUBiS: *store comment*	16	9	3	1	960	<1
RUBiS: *register user*	12	7	1	1	960	<1
RUBiS: *register item*	14	8	1	1	960	<1

TABLE I: Profiling of the Symbolic Execution analysis of update transactions in TPC-C and RUBiS benchmarks.



(a) Max Throughput

(b) Abort Rate

Fig. 3: Maximum sustainable throughput and normalized abort rates for TPC-C benchmark at different levels of contention.

For Prognosticator we used the following variants: *(i)* single-threaded re-execution of failed transactions (MQ-SF) and *(ii)* re-enqueueing failed transactions (MQ-MF) into the *lock table*, which highlights the different strategies for handling failed transactions.

The differences between Calvin and Prognosticator are the following: *(i)* DT are prepared at the client side before being submitted to the system; *(ii)* the failed DT — i.e. DT which observed a state at execution time different than in the *reconnaissance phase* — are sent back to the client to be re-prepared and executed in a future batch. To emulate this, we run a dedicated client thread that prepares DT Nms ahead of their batch execution. Therefore, the larger the N, the higher the chance that DT fail during execution. Note that because failed DT are resubmited in a future batch, their commit time is affected also by the consensus latency required to produce a new batch. In our experiments we use different values of N (100 and 200ms) corresponding to the Calvin-N variants below, to stress the effect of this design decision.

NODO schedules transactions only according to the accessed tables, therefore it hardly needs to access the datastore to detect the transaction profile as the accessed tables seldom depend on the database state. For instance, both in TPC-C and RUBiS all transactions are deemed IT by NODO.

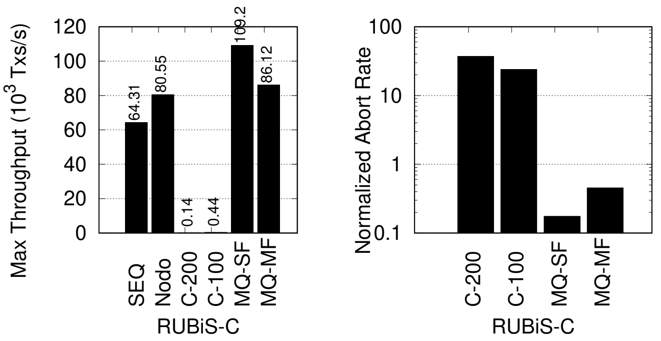
Finally, we also consider a SEQ baseline that executes all transactions sequentially using a single thread.

For all the runs, we used 20 threads and varied the transaction arrival rate by fixing the batch arrival rate at 10ms and varying the number of transactions per batch. Latency is measured from the time a transaction first arrives at a replica until it exits the system. We report the achievable throughput when then 99th percentile latency is less than 10ms.

TPC-C. Figure 3 shows the results for the standard TPC-C workload consisting of 44% new order (DT), 43% payment (IT), 4% delivery (DT), 4% stock level (ROT) and 4% order status (ROT) at different contention levels. We show the results for 100, 10 and 1 warehouse (first, second and third clusters in Figure 3, respectively) to demonstrate how Prognosticator performs in low, medium and high contention workloads. We report both the throughput (Figure 3a) and normalized abort rates (Figure 3b). The normalized abort rates is given by the percentage of failed transactions out of the total number of executed transactions over the batch size.

Overall, Prognosticator achieves the best performance across different contention levels. It either outperforms all other systems, achieving up to 5× higher throughput (Figure 3a, 100 warehouses) or, in the worst case, its throughput is on par with the best baseline (Figure 3a, 1 warehouse).

Under low contention (Figure 3a, 100 warehouses), Prognosticator with all optimizations (MQ-MF) achieves 5× higher throughput when compared with the second best system, NODO. The low contention workload allows Prognosticator to fully leverage the fine-grained transactions profiles to achieve higher degrees of parallelism and hence outperform NODO. Interestingly, because NODO does not have failed transactions (all transactions are IT) it outperforms Calvin despite the more complex machinery of the latter. In fact, Calvin suffers from a high abort rate (Figure 3b) which limits throughput, while Prognosticator achieves a low abort rate thanks to the design decision of re-executing failed transactions right away and preparing DT just before execution. Comparing the two variants of Prognosticator, we can notice that MQ-MF has 30% higher throughput than MQ-SF. This is because under low contention there is a smaller chance for transactions to fail more than once and hence re-executing failed transactions in parallel has benefits over a sequential execution. Still regarding



(a) Throughput (b) Normalized Abort Rate

Fig. 4: Maximum sustainable throughput and normalized abort rates for RUBiS benchmark.

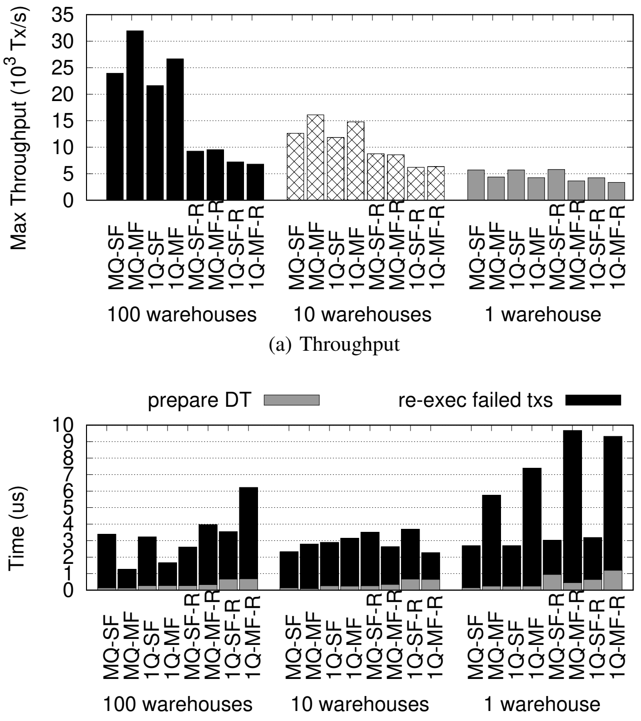
Calvin, and as expected, the value of N has a significant impact on throughput. This is because higher values of N are likely to result in observing different states between the *reconnaissance phase* and the actual execution thus leading to higher abort rates as confirmed by Figure 3b.

In a medium contention workload (Figure 3a, 10 warehouses), we observe the same trends as before albeit with with lower gains. This is mostly attributed to the lower degree of parallelism that this workload exhibits. Nevertheless, MQ-MF is capable of achieving 2.3× higher throughput than NODO. As before, the performance of Calvin (Figure 3b, 10 warehouses) is highly affected by the high abort rates.

At higher contention levels (Figure 3a, 1 warehouse), we can see that NODO outperforms the other approaches, achieving $\sim$30% higher throughput than MQ-SF. In this workload there is very little room for parallelism, therefore the simplicity of NODO outperforms the complexity of Prognosticator and Calvin that pay the overhead of exploiting fine-grained transactions profiles without being able to reap its benefits. It is worth noting that, MQ-SF achieves 30% higher throughput than MQ-MF because. In the presence of high contention, DT are expected to fail often and more than once, and hence it is better to execute them sequentially. This is confirmed by the abort rates, where MQ-SF incurs 2× lower abort rate than MQ-MF (Figure 3a, 1 warehouse).

RUBiS. RUBiS has five update transactions (Table I). All transactions insert a new entry in at least one table which requires generating a unique identifier. The identifier is generated by consulting the respective table which implies that all transactions are DT. We use the RUBiS-C workload which consists of 50% *store bid* transactions and 50% of the other transactions distributed equally [21].

Results are presented in Figure 4. RUBiS-C exhibits a high degree of contention, however, we can observe slightly different trends than TPC-C with 1 warehouse. Both Prognosticator variants are capable of outperforming all the other baselines with MQ-SF achieving 35% higher throughput than NODO. As for TPC-C, Calvin is highly affected by the abort rate (Figure 4b) which stems from the same reasons as discussed above. When comparing Prognosticator variants, we can notice that parallelizing the re-execution of failed transactions does not payoff. In RUBiS-C, transactions fail often which further emphasizes the relevance of executing failed transactions sequentially. In fact, MQ-SF achieves 3× lower abort rate than MQ-MF (Figure 4b).



(a) Throughput

(b) Time Breakdown

Fig. 5: Maximum sustainable throughput and per transaction execution times for various Prognosticator variants with TPC-C benchmark at different levels of contention.

C. Optimizations

To better understand the Prognosticator results, we now study in detail the impact of each component and optimization. To this end, we compare 8 Prognosticator variants across the following axis: *(i)* using reconnaissance vs symbolic execution (with vs without the -R suffix), *(ii)* using all threads to prepare and enqueue transactions vs a single thread (MQ vs 1Q) and *(iii)* using either a single thread or multiple threads to re-execute failed transactions (SF vs MF). We used the same TPC-C configuration as above and show the transaction throughput and transaction execution times in Figure 5a and Figure 5b, respectively. For the execution time, we split it, per transaction, into the time to prepare the DT and the time to successfully re-execute a failed transaction. Note that the time it takes to re-execute failed transactions in MF variants (where failed transactions are re-enqueued into the *lock table*) includes the time it takes to prepare them too.

When analyzing the throughput results (Figure 5a) it is clear that the SE variants outperform the reconnaissance ones (*-R) across all workloads. This is because the *-R variants always require the execution of the transaction logic to determine the key-sets which not only needs more time than non *-R variants but also imposes more load on the underlying datastore (Figure 5a). Thanks to SE, we do not need to execute all the transaction logic which significantly reduces the time needed to prepare a DT.

Similarly, by parallelizing the preparation of DT in the MQ variants as opposed to the single thread 1Q variants, we are able to reduce the transaction preparation time and hence improve throughput. Finally, the choice between SF and MF depends on the contention level. Under low contention, MF variants are capable of achieving higher throughput while in higher contention SF performs better (Figure 5a, 100 warehouses and 1 warehouse, respectively). The cause is once again the time to re-execute failed transactions. In low contention workloads, where transactions are expected to not fail frequently, the MF variants incur lower execution times than SF, whereas in the high contention workload, the opposite is true. This is because under high contention DT are likely to fail often and thus it is better to execute them sequentially.

V. Related Work

The initial approach for implementing deterministic databases relied on single-threaded sequential execution of transactions [14], [26]. If transactions are executed by a single thread following the same order on different replicas, then the state of the data on each replica is guaranteed not to diverge. However, in the multi-core era, such a design is clearly not going to scale. To circumvent this limitation, several works such as H-Store [13] and VoltDB [38] proposed partitioning the data into disjoint partitions, i.e., no transaction would access two partitions. Accordingly, different threads can execute transactions of different partitions without the need for inter-thread coordination. However, multi-partition transactions that constitute majority of real-life workloads [7], [27] render it impractical as it has to fallback to a more expensive coordination mechanism, e.g., serial execution.

Recently, Thomson et al. proposed Calvin [35], a deterministic database that allows concurrent execution of transactions without the need for disjoint partitions. With Calvin, transactions are batched and then sent to a sequencers to be ordered. After agreeing on an order, sequencers forward the transactions to replicas that execute transactions concurrently while guaranteeing to yield a serial order equivalent to the order agreed upon by the sequencers. To achieve this, the lock manager allows a transaction to acquire locks once all the preceding transactions have acquired all their locks. This requires the prior knowledge of which locks each transaction needs to acquire, i.e., transactions must expose their read- and write-sets. Calvin relies on either the manual input of the read- and write-sets by the developer or through the OLLP protocol [34]. The former is a time-consuming and complex task where errors may lead to violating the serializabilty of the transactions. The latter performs a *reconnaissance-phase* where transactions are executed without any isolation guarantees at the client side to collect the read- and write-sets. The *reconnaissance-phase* is particularly necessary for dependent transactions (i.e., transactions whose read- and write-sets depends on the database state). Prognosticator, instead, leverages symbolic execution to automatically define the read- and write-sets for each transaction via offline analysis. For dependent transactions, where SE defines only which keys rely on the database state (i.e., indirect keys), Prognosticator performs a *prepare indirect keys* phase to collect those keys. The *prepare indirect keys* is performed at the server side using a deterministic state just before executing dependent transactions. This subtle difference exposes dependent transactions within Prognosticator to a much shorter “vulnerability window” during which the data observed in the prepare phase can change, leading to much lower abort rates when compared to Calvin as we have shown in §IV. Moreover, OLLP increases the latency of transactions as it requires executing each transaction (at least) twice: once during a *reconnaissance-phase* to determine the read- and write-sets and once during the actual execution, unlike Prognosticator that needs to only read the *pivot keys* during the *prepare indirect keys* phase.

Prognosticator achieves determinism by exploiting application-specific knowledge, specifically the read- and write-sets. This allows Prognosticator to implement an efficient deterministic concurrency control. Several solutions were devised in the literature to support deterministic multi-threaded execution at either the OS or the threads library level [23], [18], [6], [19], [20]. Although these systems achieve determinism without requiring any knowledge about the application, they impose significant overheads that lead to severe slowdowns. Moreover, these solutions were not designed to support database replication, so utilizing them to achieve replication may not be straightforward.

Prior to the introduction of deterministic databases, a large body of literature has been developed investigating replication techniques that assumed non-deterministic concurrency controls at each replica, such as such as two-phase locking [10], timestamp ordering [3] and optimistic concurrency control [16]. Recent studies [30], [12] have performed extensive comparison between these systems and deterministic databases. Although the results have shown that there is no one size fits all and different workloads better fit different algorithms, deterministic databases have proven to be more robust against changes in workload and execution environments. Prognosticator inherits those advantages and manages to outperform state of the art deterministic databases, which further enhanced the state of the art of.

Prognosticator is also related to works aimed at predicting conflict classes among transactions. NODO [26] assumes that the transaction logic is expressed via a classic SQL-based interface, based on which it is typically relatively simple to identify which database table will be read/written within each transaction, independently of the transactions’ inputs. Such scheme provides a coarse grained conflict class detection, i.e. at the table level, which leads to poor parallelism degree since multiple transactions accessing different keys in the same table will be refrained from executing concurrently. Wang et al. [39] relaxed the constraint on SQL-based interface, but still suffered from the poor parallelism due to the coarse granularity in their static analysis-based conflicts classes. Pavlo et al. [28]

proposed using Markov-based modeling to predict the actions performed by transactions to decide upon which optimization to apply to them. Unlike SE, such approach provides only probabilistic information on the read- and write-sets which is not enough to ensure determinism.

VI. Conclusions

In this paper we present Prognosticator, a deterministic database system that leverages offline symbolic execution analysis to automate the process of exposing transactions' read- and write-sets. Prognosticator incorporates an algorithm that harnesses symbolic execution to achieve this goal while maintaining memory footprint of the offline analysis at amenable levels. Furthermore, it introduces efficient and scalable scheduling and execution mechanisms to consume the information produced by the offline symbolic analysis while ensuring that the state of data in different replicas does not diverge. Our thorough evaluation study across a wide range of workloads against state of the art solutions has shown that these techniques allows for achieving up to 5× higher throughput than state of the art solutions.

Acknowledgment

Support for this research was provided by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) via projects PTDC/EEISCR/1743/2014, LISBOA-01-0145-FEDER-031456 (Angainor) and UIDB/50021/2020.

References

[1] D. J. Abadi and J. M. Faleiro. An overview of deterministic database systems. *Commun. ACM*, 2018.

[2] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 2018.

[3] P. A. Bernstein, P. A. Bernstein, and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 1981.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'08.

[5] O. Consortium. Rice University Bidding System. https://rubis.ow2.org.

[6] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *ACM Symposium on Operating Systems Principles*, SOSP '13.

[7] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 2010.

[8] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[9] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, 2006.

[10] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 1976.

[11] FACEBOOK OPEN SOURCE. RocksDB: A persistent key-value store. https://rocksdb.org.

[12] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 2017.

[13] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 2008.

[14] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *International Conference on Very Large Data Bases*, VLDB '00.

[15] J. C. King. A new approach to program testing. In *International Conference on Reliable Software*, 1975.

[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 1981.

[17] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[18] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *ACM Symposium on Operating Systems Principles*, SOSP '11.

[19] T. Merrifield, J. Devietti, and J. Eriksson. High-performance determinism with total store order consistency. In *European Conference on Computer Systems*, EuroSys '15.

[20] T. Merrifield, S. Roghanchi, J. Devietti, and J. Eriksson. Lazy determinism for faster deterministic multithreading. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19.

[21] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[22] NASA. Symbolic PathFinder. https://github.com/SymbolicPathFinder/jpf-symbc, 2005.

[23] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV.

[24] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14.

[25] Parisa Jalili Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *IEEE/IFIP International Conference on Dependable Systems Networks*, DSN'10.

[26] M. Patiño Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *International Conference on Distributed Computing*, DISC '00.

[27] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *ACM International Conference on Management of Data*, SIGMOD '12.

[28] A. Pavlo, E. P. C. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proc. VLDB Endow.*, 2011.

[29] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *International Symposium on Software Testing and Analysis*, ISSTA '08.

[30] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow.*, 2014.

[31] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[32] N. Santos and A. Schiper. Optimizing paxos with batching and pipelining. *Theoretical Computer Science*, 2013. Distributed Computing and Networking (ICDCN 2012).

[33] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 1990.

[34] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 2010.

[35] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '12.

[36] TPC Council. Transaction Processing Performance Council, TPC BENCHMARK™ C. http://www.tpc.org/tpcc.

[37] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99.

[38] VoltDB. VoltDB. https://www.voltdb.com/.

[39] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *ACM International Conference on Management of Data*, SIGMOD '16.