# Framework for Replaying Transactions Using Dependency Graph

Virinchi Srinivas

## 1 Introduction

There has been a lot of work recently on the topic of "Deterministic Transaction Processing" that assumes that the read and write sets of transactions are known in advance before the execution of the transaction. The knowledge of the read and write sets of transactions enables scheduling the transactions in some global order efficiently while being able to guarantee serializability while achieving high concurrency. The advance knowledge of read and write sets of transactions allows to perform at least the following in an efficient manner:

- Concurrency Control: We can design a concurrency control by scheduling a batch of transactions by using the knowledge of the read and write sets of the transactions for efficient execution of transactions. This deterministic scheduling and execution of transactions potentially introduces a possibility of obtaining higher concurrency when compared to executing the transactions under existing concurrency protocols or its variations like Two-Phase locking (2PL), OCC and MVCC that could suffer from overheads due to non-deterministic scheduling.
- Efficient Replay: We can design a framework for replaying transactions based on the transaction that made the unauthorized change; we need to undo the change by replaying the transactions. We can perform this task by constructing a dependency graph that uses the knowledge of the read and write sets of the transactions. This is efficient when compared to the trivial way to replay transactions that executes all the transactions irrespective of the transaction that performed the unauthorized update.

In this work, we address the second problem which is designing and implementing an efficient framework to replay transactions. In the next section, we explain the problem in detail.

## 2 Problem

Let us consider that $n$ transactions $T_1$,$T_2$,...,$T_n$ have executed on $d$ items $x_0, x_1$, $x_2$,...,$x_{d-1}$. For 2 transactions $T_a$ and $T_b$, $T_a$ commits before $T_b$ iff $a < b$. Further, each transaction reads $r$ and writes $w$ items ($(r + w) << d$). Let us denote the read and write sets of a transaction $T$ by $R_T$ and $W_T$ respectively. At a given point in time, let us say that we realize that $T_k$ ($k \leq n$) makes unauthorized update(s) and we want to undo the change. We can perform this in two ways:

- Case 1: We can thrash the entire database and replay tranasactions $T_k, T_{k+1}, ..., T_n$ to undo the unauthorized update performed by $T_k$.
- Case 2: A smarter and more efficient option to undo this change would be to replay a subset of transactions that read or wrote the items written by $T_k$ i.e replay those transactions among $T_k, T_{k+1}, ..., T_n$ that directly or indirectly depend on the write set of $T_k$.

We illustrate the difference between the above two cases by considering the following scenarios.

1. **Unauthorized Update on 1 item by a transaction:** Let us consider that 6 transactions $T_1, T_2, ..T_6$ executed, where each transaction performs reads and writes to items. For simplicity, let us assume that each transaction writes every item that it reads (RWM). We can construct a dependency graph using the knowledge of read and write sets of these transactions. For now, let us assume that we are given the dependency graph for 6 transactions as

shown in Figure 1; we explain how to construct the dependency graph later. For example, we can see that $T_1$ reads and updates $x_1$ and $x_{10}$ (Figure 1).

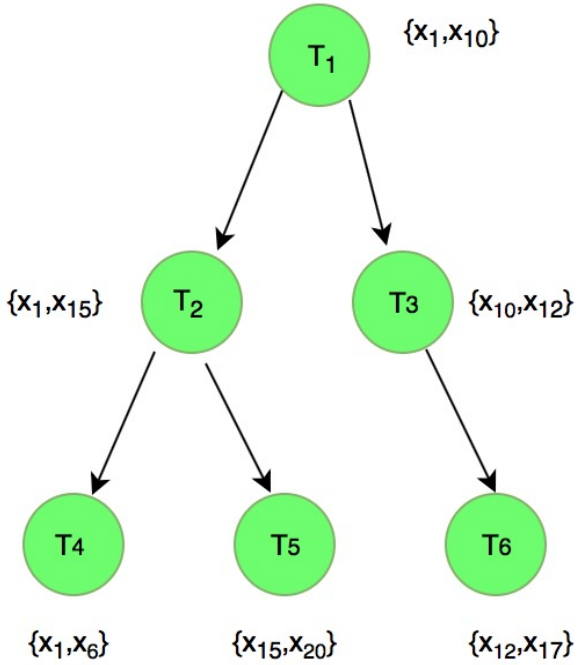


Figure 1: $T_1$ makes unauthorized update on $x_1$

Let us consider that an unauthorized transaction $T_1$ wrongly updated $x_1$ that might correspond to the number of products of item $x_1$ in an inventory. We need to undo this wrong update of $x_1$. An trivial way to undo this change is to replay all transactions starting from $T_1$ i.e. $T_1, T_2, T_3, T_4, T_5, T_6$ because all these transactions were executed after $T_1$.

However, by specifically considering the dependency graph and the fact that $T_1$ wrongly updated $x_1$, we can replay more efficiently. From Figure 1, note that only transactions $T_2, T_4, T_5$ are affected by the unauthorized update made by $T_1$ on $x_1$. In other words, transactions $T_3$ and $T_6$ are unaffected by this wrong update made by $T_1$. Hence, we could replay efficiently by replaying the transactions $T_1, T_2, T_4, T_5$ in the same order as they were executed, thereby pruning the unaffected branch $T_3$ and $T_6$.

2. **Unauthorized Update on Complete Write Set (W) by a transaction:**

Let us consider that 6 transactions $T_1, T_2, ..T_6$ executed each one reading 3 items and writing 2 items. The read and write sets are denoted by R and W respectively in Figure 2. We construct a dependency graph using the knowledge of read and write sets of these transactions as shown in Figure 2. We can see that $T_1$ reads $x_3$,$x_6$,$x_{17}$ and makes unauthorized updates on $x_{13}$ and $x_{15}$.

In order to undo this unauthorized update made by $T_1$ on W, like we explained before, one way to undo this change is by replaying all transactions that executed after $T_1$ i.e. $T_1$ to $T_6$ in the same order. However, by using the write sets of $T_1$ and the dependency graph, we can replay transactions efficiently.

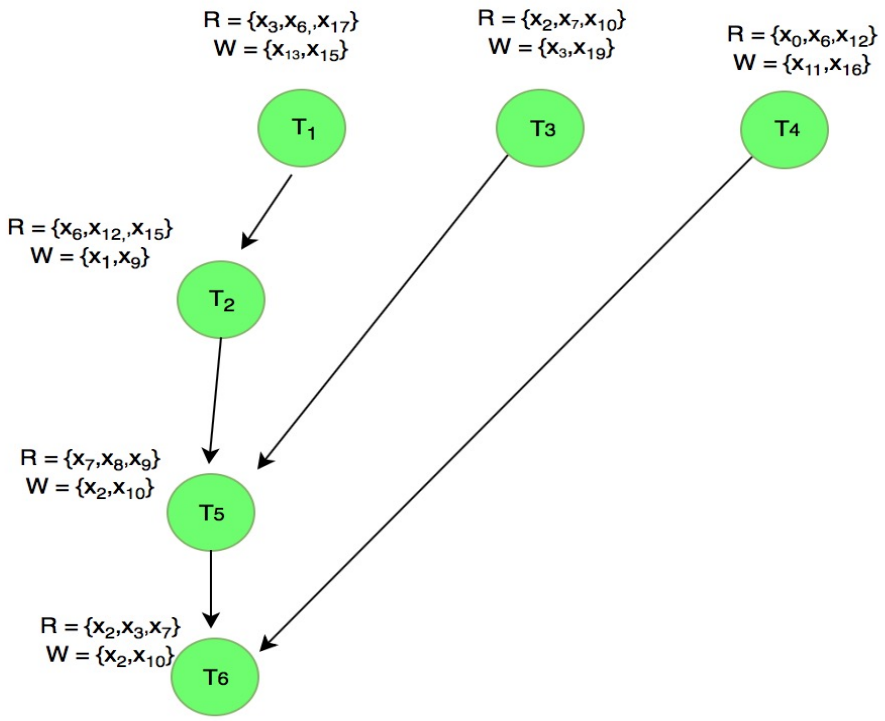


Figure 2: $T_1$ makes unauthorized update on W = $\{x_{13}$,$x_{15}\}$

From the dependency graph shown in Figure 2, we can observe that only transactions $T_2$, $T_5$ and $T_6$ are affected by the unauthorized update made by transaction $T_1$ on its write set $x_{13}$ and $x_{15}$. Further, $T_3$ and $T_4$ are unaffected by the unauthorized update made by $T_1$. Hence, we efficiently replay transactions $T_1$, $T_2$, $T_5$ and $T_6$ in the same order to undo this unauthorized update and avoid replaying $T_3$ and $T_4$.

From the above two scenarios, we observe that using the knowledge of the read and write sets of all the transactions, we could construct the dependency graph which can be used to efficiently replay only the relevant transactions that are affected by the unauthorized update. We explain the high-level overview of transaction execution and replay pipeline next followed by dependency graph construction in Section 4.

# 3 Transaction Execution and Replay Pipeline

We present the high-level overview pipeline involving transaction execution and replay in Algorithm 1.

**Algorithm 1** Transactions Execution and Replay Pipeline

```
1: procedure ReplayPipeline(n, unauth, cmode, rmode)   ▷ n :no. of txns, unauth:id of unauthorized txn, cmode: Concurrency Control Mode, rmode: Replay Mode
2:     for i in 1 to n-1 do
3:         ExecuteTxn(T_i,cmode)
4:     if rmode == COMPLETE then                        ▷ Complete Replay
5:         CompleteReplay(T,unauth,n)
6:     else                                             ▷ Smart Replay
7:         G = Construct-Dependency-Graph(T,unauth,n)   ▷ G:Dependency Graph
8:         SmartReplay(G,T,unauth,n)
```

The first phase in the pipeline consists of executing the $n$ transactions in the database using the concurrency control protocol cmode. cmode could be either SERIAL, 2PL, OCC, MVCC or any

variant. However, in this work, we do not focus on designing a concurrency protocol; our primary focus is on the second phase (line 4 onwards in Algorithm 1) that involves replaying transactions.

After the transactions have executed, and we realize that $T_{unauth}$ ($unauth < n$) has made an unauthorized update, we need to undo this unauthorized update. The replay mode, given by, *rmode* could be either COMPLETE or SMART as illustrated earlier. In the case of COMPLETE replay mode, we replay all the transactions starting from $T_{unauth}$ till $T_{n-1}$. However, if *rmode* is SMART, we construct the dependency graph $G$ of transactions using the Construct-Dependency-Graph module which we explain next. We use $G$ to efficiently replay relevant (affected) transactions and prune irrelevant (unaffected) transactions. Note that the technique followeed to replay transactions is independent of the concurrency control protocol followed to execute the transactions. However, the concurrency control protocol decides the commit order and the timestamps assigned to the transactions.

We should take care to ensure that during transaction replay, each transaction reads the correct values before performing the writes to undo an unauthorized update. We accomplish this by storing the read and write buffers for each transaction as it executes (before replay). Further, for each item in the in-memory key-value, we also maintain the id of the transaction that made the latest update to that item.

Let us consider a scenario where we have 100 transactions $T_1$, $T_2$,...,$T_{100}$. Further, let $T_1$ reads and makes a wrong update on $x_1$, $T_2$ reads and updates $x_1$ and $T_{100}$ reads and updates $x_1$. There arises two cases during replay while each transaction either uses the read value from its local read buffer or performs a read from the in-memory key-value store as explained.

1. Using the value from local read buffer: $x_1$ is last updated by $T_{100}$ during execution phase. While replay, $T_1$ consists of the value of $x_1$ in its local read buffer (when $T_1$ was executed). $T_1$ has two choices, use the value of $x_1$ from its read buffer or perform a read from the in-memory key-value store for $x_1$. We observe that $x_1$ was last updated by $T_{100}$ and timestamp($T_{100}$) > timestamp($T_1$). Hence, $T_1$ has the correct read value for $x_1$ in its local buffer, which it uses while replaying.

2. Reading the value from in-memory key-value store and overwriting local read buffer: Consider the case of $T_2$ assuming that $T_1$ has already replayed. $T_2$ has two choices, use the value of $x_1$ from its read buffer or perform a read from the in-memory key-value store for $x_1$. After $T_1$ has replayed, we observe that $x_1$ was last updated by $T_1$. Further, timestamp($T_1$) < timestamp($T_2$) according to the commit order. Hence, $T_2$ reads $x_1$ from the in-memory key-value store and overwrites its local read buffer after which it performs its replay.

By maintaining the local read and write buffers along with the transaction id that made the latest update to each transaction, we could ensure that each transaction performs the correct read while replay. However, we could use a more general form of versioning that maintains versions of each item to accomplish the same. We discuss how to construct the dependency graph in the next section.

# 4 Dependency Graph

Dependency graph $G$ is a graph of transactions where each node corresponds to a transaction and an edge corresponds to dependency between the transactions. We present some properties of the dependency graph that are invariant during graph construction for the sake of correctness:

- An edge can exist between two nodes (transactions) $T_i$ and $T_j$ iff $i < j$. We want to ensure that transactions are replayed in the same order they were earlier executed.
- For $i < j$, if $T_j$ connects to $T_i$ ($T_i \implies T_j$), $T_j$ is not connected to any ancestor of $T_i$. Ancestors of $T_i$ are the transactions that $T_i$ indirectly depends on. We follow this to ensure efficiency and correctness of the SmartReplay algorithm which we explain later.
- For $i < j$, then $T_j$ connects to $T_i$ if either:
  - $R_{T_j}$ overlaps with $W_{T_i}$ (WR conflict) where, $R_T$ and $W_T$ corresponds to read and write sets of T respectively or
  - $W_{T_j}$ overlaps with $W_{T_i}$ (WW conflict).

We present the algorithm to construct the dependency graph that satisfy the above invariants in Algorithm 2.

**Algorithm 2** Dependency Graph Construction

```
1: procedure CONSTRUCT-DEPENDENCY-GRAPH(T, unauth, n)   ▷ n :no. of txns, unauth:id of unauthorized txn
2:     for j in unauth to (n-1) do
3:         add node T_j to G
4:         for i in (n-1) to unauth do
5:             if T_j not connected to any descendant of T_i then
6:                 if (R_{T_j} ∩ W_{T_i} ≠ ∅ or W_{T_j} ∩ W_{T_i} ≠ ∅) then
7:                     add edge from T_i to T_j to G
```

As we need to replay the transactions starting from $T_{unauth}$, we construct the dependency graph $G$ having nodes from $T_{unauth}$ to $T_{n-1}$. We add every transaction node starting from $T_{unauth}$ to $G$. For optimality, while adding a transaction $T_j$ to the graph, we consider all potential transactions $T_i$'s for adding an edge in the reverse order. The reason for this is when $T_j$ connects to $T_i$ with an edge, we can skip all other ancestors of $T_i$ for potential edge consideration.

When we observe such a $T_i$ whose descendant is not linked to $T_j$, we compute the overlap between the $R_{T_j}$ and $W_{T_i}$. This corresponds to Write-Read (WR) conflict between $T_i$ and $T_j$. Similarly, an edge could be established from $T_i$ to $T_j$ when there arises a Write-Write (WW) conflict i.e. overlap between $W_{T_j}$ and $W_{T_i}$. In the next section, we explain how the transactions are replayed using both COMPLETE and SMART replay technique. The SmartReplay technique uses the dependency graph for smart replay.

# 5 Transactions Replay

In the section, we discuss both the replay techniques.

## 5.1 Complete Replay

We describe the algorithm we follow to completely replay all transactions in Algorithm 3.

**Algorithm 3** Complete Replay

```
1: procedure COMPLETEREPLAY(T, unauth, n) ▷ n :no. of txns, unauth:id of unauthorized txn
2:     for i in unauth to (n-1) do
3:         ExecuteTxn(T_i,SERIAL)
```

From the algorithm, we observe that in order to undo any change made by $T_{unauth}$, we replay all transactions starting from $T_{unauth}$ to $T_{n-1}$.

## 5.2 Smart Replay

We describe the algorithm we follow to completely smartly replay transactions in Algorithm 4.

**Algorithm 4** Smart Replay

```
1: procedure SmartReplay(T, unauth, n)        ▷ n :no. of txns, unauth:id of unauthorized txn
2:     visited = ∅
3:     batch = ∅
4:     visited[T_unauth]=true
5:     batch.push_back(T_unauth)
6:     while batch not empty do
7:         T = batch.pop_front()
8:         ExecuteTxn(T)
9:         for S in children[T] do
10:            if (!visited[S]) then
11:                visited[S] = true
12:                batch.push_back(S)
```

From the algorithm, we observe that in order to undo any change made by $T_{unauth}$, we replay only the transactions starting from $T_{unauth}$ that are relevant (reachable) from $T_{unauth}$ using BFS algorithm. This ensures that transactions that are irrelavant (unreachable) by the unauthorized update are not replayed. Connecting a node to the least descedant ancestor during graph construction ensures that we do not execute (explore) a child transaction before we execute its ancestors. We conduct experiments that compare the performance of CompleteReplay and SmartReplay algorithm.

# 6 Experimental Evaluation

For performance comparisons we compare our prototype implementation of CompleteReplay and SmartReplay on the YCSB Benchmark. For the experiments, we use a single table. We run 1000 transactions. The read and writesets of the transactions are generated as uniform distribution. All the experiments were performed on junkfood (junkfood.cs.umd.edu) machine. The code can be found at Project Github Repository.We use two kind of workloads:

1. 10 Read-Modify-Write (10RMW): In this workload, each transaction reads 10 items and writes the same 10 items that it previously read. We ensure the that the read and write sets of each transaction is unique.

2. 2RMW-8R: In this workload, each transaction performs 2RMWs and 8R (reads). Like earlier, we ensure that the read and write sets of each transaction is unique.

For both the workloads, we report the results by making the first transaction do the unauthorized update to its complete write set. We vary the running time of transactions as 1ms and 10ms that corresponds to short and long transactions respectively. Further, we run each experiment 10 times and report the average of the 10 runs when we report the results. As mentioned earlier, the performance results are independent of the Concurrency Control protocol that executed the transactions. For shorthand notations, from hereafter, let CR, GC, SR, GC+SR correspond to CompleteReplay, Graph Construction, SmartReplay, GraphConstruction+SmartReplay respectively.

## 6.1 10RMW Workload

As explained earlier, in 10RMW Workload, each transaction reads 10 items and updates the 10 items that it had read. We report the performance of CompleteReplay and SmartReplay under both low and high contention for this workload in Table 1.

| Txn Length | Parameter | Low Contention (20000 records) | High Contention (5000 records) |
|---|---|---|---|
| **Short Txn (1ms)** | *CR* | 1.0267 | **1.0276** |
| | *GC* | 0.072 | 0.845 |
| | *SR* | 0.056 | 0.783 |
| | *GC+SR* | **0.128** | 1.628 |
| | *Relevant txns for CR* | 1000 | 1000 |
| | *Relevant txns for SR* | 54 | 758 |
| **Long Txn (10ms)** | *CR* | 10.176 | 10.126 |
| | *GC* | 0.072 | 0.831 |
| | *SR* | 0.55 | 7.632 |
| | *GC+SR* | **0.622** | **8.463** |
| | *Relevant txns for CR* | 1000 | 1000 |
| | *Relevant txns for SR* | 51 | 775 |

Table 1: Running Time (sec) on 10RMW Workload

From Table 1, let us first discuss the case of low contention. We observe that for both shorter and longer transactions, GC+SR performs approximately 8 to 16 times faster when compared to CR for shorter and longer transactions respectively. Further, ignoring GC, SR performs 18 times faster when compared to CR for both shorter and longer transactions respectively. The main reason for this is CR needs to replay all the 1000 transactions, while SR replays around 50 relevant transactions thereby pruning upto 950 transactions.

For higher contention, in the case of shorter transactions, GC takes more time compared to the case of lower contention. Further, for short transactions, GC+SR takes more time when compared to CR; SR is much faster than CR because it avoids replaying around 250 transactions that CR runs in addition. We observe that during the case of high contention, we pay an additional overhead for GC.

## 6.2 2RMW-8R Workload

As explained earlier, in 2RMW-8R Workload, each transaction performs 2RMSs and reads 8 items. We report the performance of CR and SR under both low and high contention for this workload in Table 2.

| Txn Length | Parameter | Low Contention (10000 records) | High Contention (2500 records) |
|---|---|---|---|
| **Short Txn (1ms)** | *CR* | 1.016 | 1.015 |
| | *GC* | 0.032 | 0.16 |
| | *SR* | 0.008 | 0.189 |
| | *GC+SR* | **0.040** | **0.349** |
| | *Relevant txns for CR* | 1000 | 1000 |
| | *Relevant txns for SR* | 8 | 186 |
| **Long Txn (10ms)** | *CR* | 10.102 | 10.129 |
| | *GC* | 0.032 | 0.161 |
| | *SR* | 0.081 | 1.865 |
| | *GC+SR* | **0.113** | **2.026** |
| | *Relevant txns for CR* | 1000 | 1000 |
| | *Relevant txns for SR* | 8 | 181 |

Table 2: Running Time (sec) on 2RMW-8R Workload

From Table 2, let us first discuss the case of low contention. We observe that for both shorter and longer transactions, GC+SR performs approximately 3 to 25 times faster when compared to CR. The primary reason for superior performance of GC+SR is that CR needs to replay all the 1000 transactions, while SR replays around 10 relevant transactions there by avoiding to replay more

than 990 transactions. For higher contention, in the case of both shorter and longer transactions, we observe a pattern similar to the case of low contention.

## 6.3 Varying Graph Construction for 10RMW Workload

From the previous experiments on 10RMW and 2RMW-8R workloads, we observe some interesting points.

- For high contention 10RM Workload involving short transactions, we observe that SR+GC incurs 1.5 times more running time than CR.
- For high contention 10RMW workloads, we observe that much more relevant transactions using SR when compared to low contention workloads. We expect dependency graphs to have longer chains for high contention workloads as opposed to low contention workloads. We end up paying more for GC.

However, for the case of 2RWM-8R workload, the overhead of GC is much lesser. Hence, for the rest of section, we perform experiments that involve only 10RMW workload. Unless mentioned specifically, all the discussion in this section, hereafter pertains to 10RMW workload.

### 6.3.1 Motivation

For high contention 10RMW workload,

- We expect dependency graphs to form longer chains.
- For shorter transactions, GC might incur higher overhead due to the contention. GC+SR might be much more than CR.
- The number of relevant transactions for SR much higher. Approximately 70-80% of CR.

Keeping the above points in mind, we observe a *trade-off* between SR and GC. If we build the dependency graph using all transactions, we incur more overhead for GC and reduce running time of SR. If we build the dependency graph using a few transactions, we incur lesser overhead for GC and pay additional overhead for SR. However, in the case of high contention 10RMW workloads, we observe dependency graphs having longer chains. Hence, we anyway need to replay more transactions. This motivates us to think of a possibility where we could trade-off by creating the dependency graph by using $x$% of the transactions and replaying the rest $(1-x)$% transactions serially.

We conduct experiments in the case of 10RMW workload by varying the graph construction by creating the dependency graph using $x$% of the transactions and replaying the rest $(1-x)$% transactions serially where $0 \leq x \leq 100$. When $x$ is 0, we replay 100% of the transactions where in it is CR. For smaller values of $x$ ($<$10) we expect trends similar to CR. For larger values of $x$, we expect to pay higher price for GC in the case of high-contention short transactions. We present the results next under high contention and low contention workloads separately.

### 6.3.2 High Contention

The results pertaining to varying graph construction for high contention workloads is presented in Figure 3 and 4. Figure 3 presents the specific case involving short transactions. As we increase the number of transactions that are used for graph construction, we observe that time to construct the graph (GC) increases. GC cost is maximum at 100% when all the transactions are used for graph construction. However, at 100%, we observe that GC+SR is much higher than CR. This is what we expect for high contention workloads. However, when 20-40% of transactions are used to create the graph and rest 60-80% are completely replayed, we observe that GC+SR is more efficient when compared to CR. We observe that the difference between GC+SR and CR is maximum when 20-40% of transactions are used for GC.

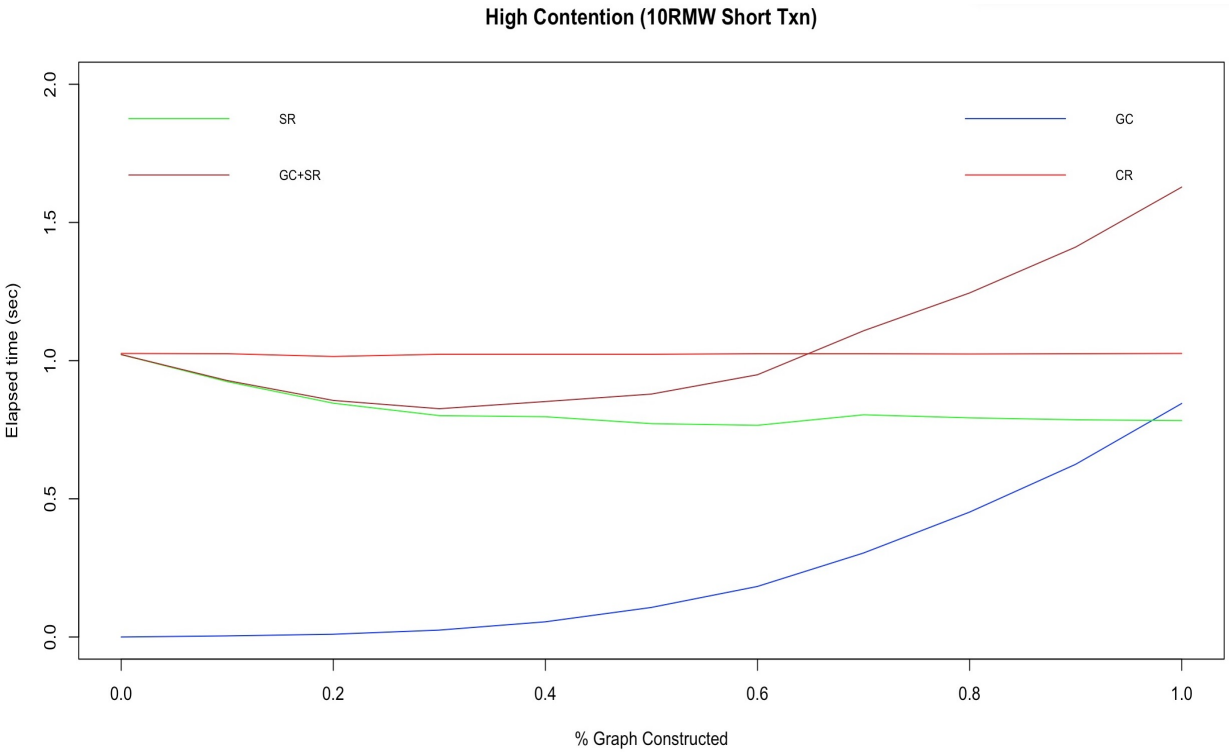


Figure 3: High Contention : Elapsed Time vs % Graph Constructed

Figure 4 presents the result having long transactions. We observe a similar trend although the severity is much lesser due to longer transactions. However, we can observe that when 40-60% are used for graph construction, it is the optimal point of trade-off between GC and SR. We observe that the difference between GC+SR and CR is maximum when 40-60% of the transactions are used in GC.

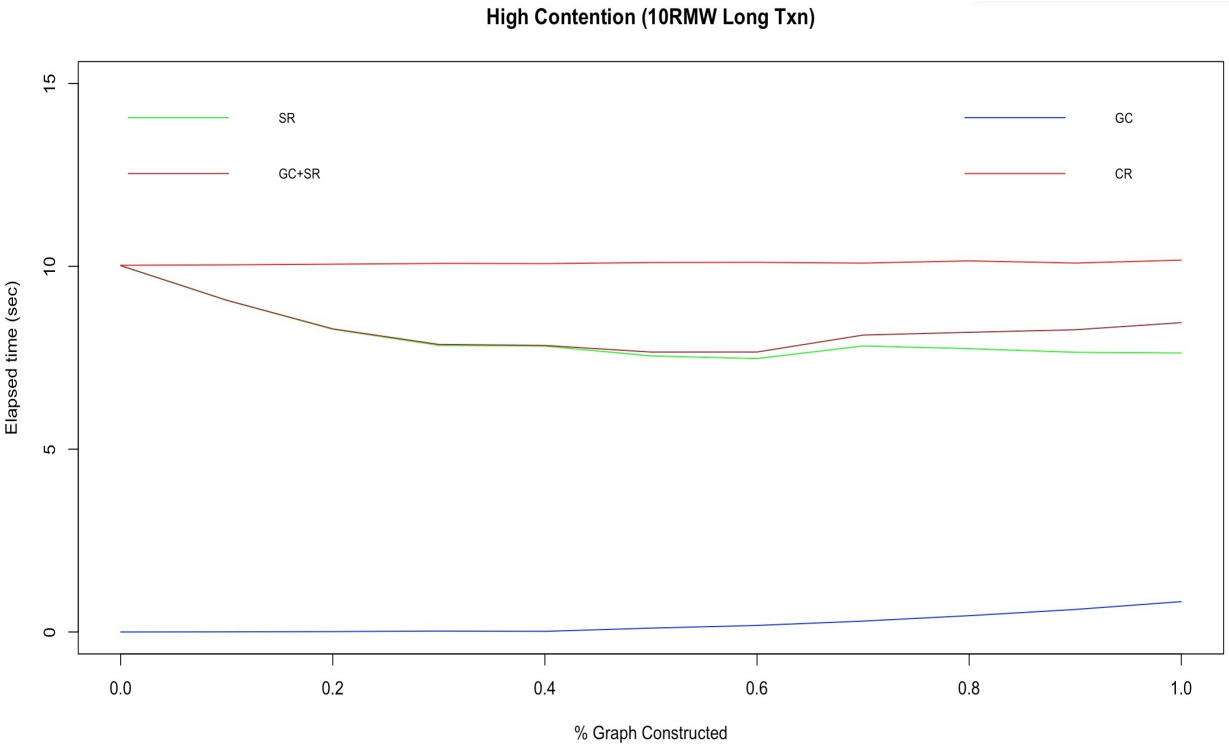


Figure 4: High Contention : Elapsed Time vs % Graph Constructed

### 6.3.3 Low Contention

The results pertaining to varying graph construction for low contention workloads is presented in Figure 5 and 6. From the results, we observe that in the case of low contention workload, for both short and long transactions, it is optimal to create the graph using all the transactions and replay efficiently using SR. We observe that the difference between GC+SR and CR is maximum at 100%.

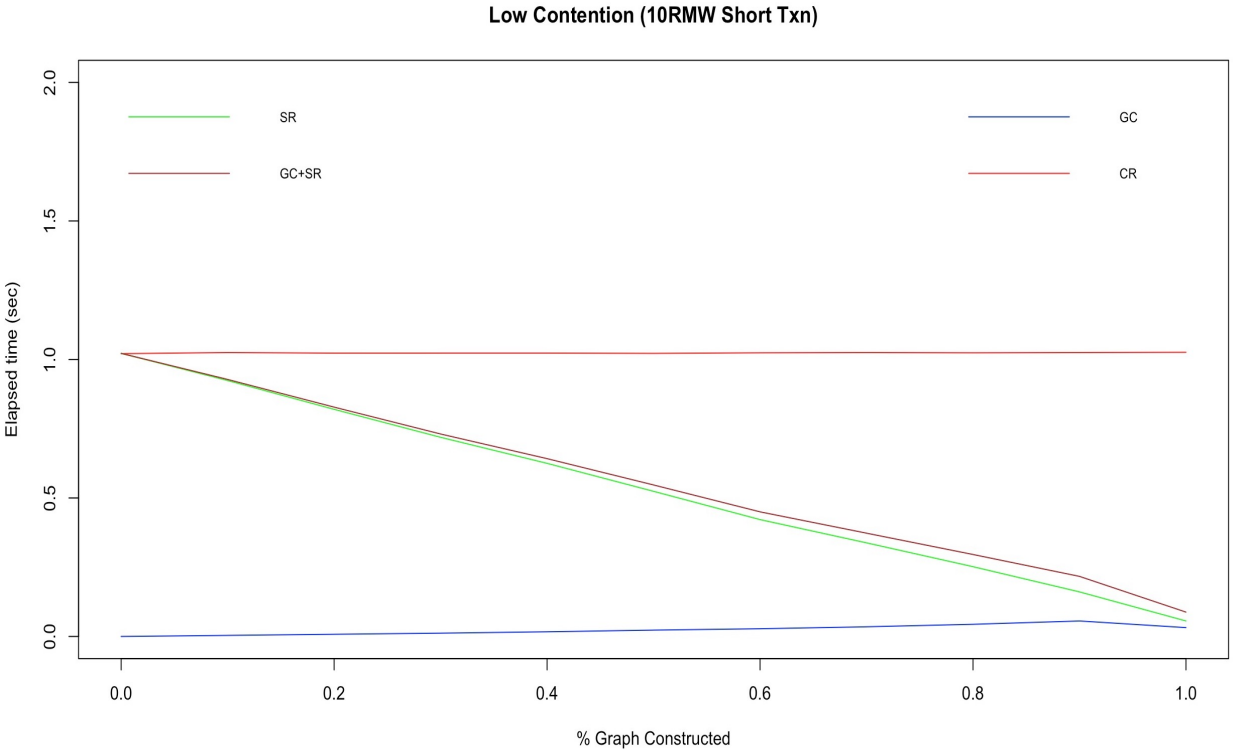


Figure 5: Low Contention : Elapsed Time vs % Graph Constructed

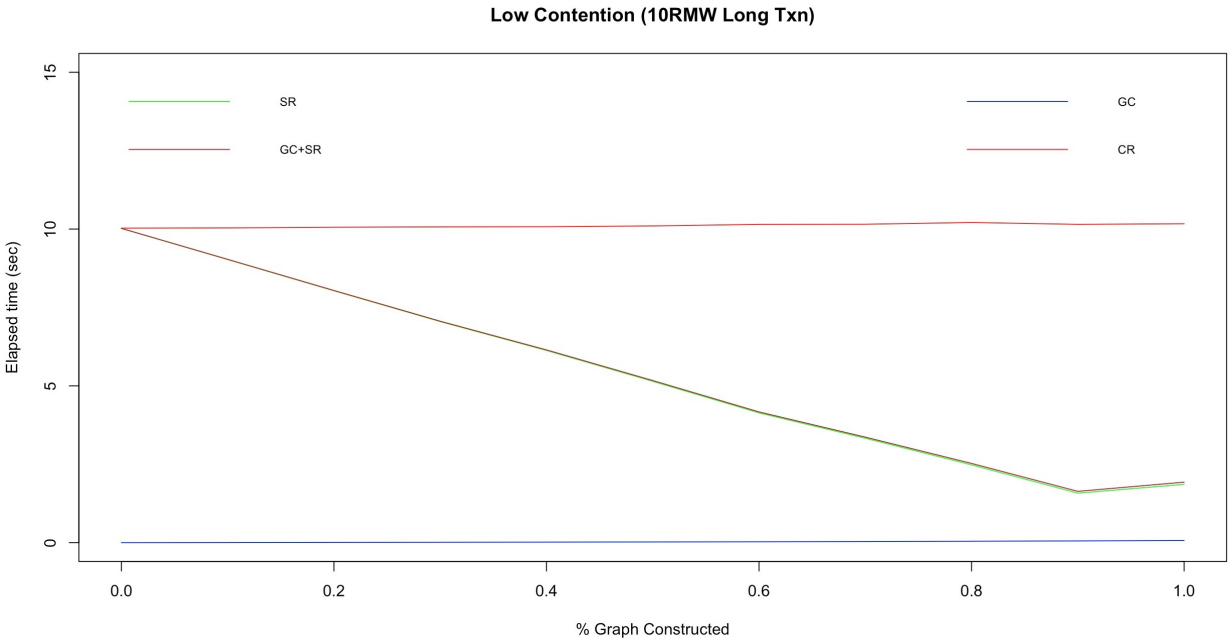


Figure 6: Low Contention : Elapsed Time vs % Graph Constructed

# 7 Summary

We present a framework that allows efficiently replaying transactions by constructing the dependency graph. At a very high level, the key points of the work are as follows:

- SmartReplay is much efficient when compared to CompleteReplay as it replays relevant transactions and avoids replaying irrelevant transactions.
- Graph construction overhead is minimum for low contention workloads and maximum for high contention workloads.
- In the case of low contention workloads, we might construct the dependency graph by using all the transactions. This saves a lot during SmartReplay. $GC + SR - CR$ is maximum when all the transactions are used for graph construction.
- In the case of high contention workloads, we might construct dependency graph using the first $20-40\%$ of transactions and replay the rest $60-80\%$ of transactions due to the expected longer chains in dependency graph for high contention workloads. $GC+SR-CR$ is maximum when the first $20-40\%$ of transactions are used for graph construction.

# 8 Future Work

We presented a simple framework that enables replaying transactions efficiently. However we could expand on the following points in the future.

- Perform Parallel Dependency Graph construction: We could create dependency graphs in batches and connect them. High overhead is incurred for creating a larger dependency graph having $n$ transactions rather than having 10 dependency graphs having $\frac{n}{10}$ transactions each. This gives a perfect setting for possible parallelism.
- Perform Parallel SmartReplay Algorithm: We could create a concurrency protocol that enables SmartReplay in parallel wherein each of the children could be replayed in parallel.