Achieving Starvation-Freedom in Multi-Version Transactional Memory Systems

Ved Prakash Chaudharynst1 Chirag Juyal[1] Sandeep Kulkarni[2]
Sweta Kumari[1] **Sathya Peri[1]**

[1]CSE Dept, IIT Hyderabad [2]CSE Dept, Michigan State University

The 7th Edition of The International Conference on Networked Systems (NETYS 2019)

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

Introduction to STMs

Software Transactional Memory Systems

What is a memory transaction?

- Sequence of instructions executing in memory
- Satisfying Atomicity

Introduction to STMs

Software Transactional Memory Systems

What is a memory transaction?

- Sequence of instructions executing in memory
- Satisfying Atomicity

What is Software Transactional Memory?

- A parallel programming paradigm
- Avoids concurrency overheads at programmers level
- Execute code optimistically

Introduction to STMs

Software Transactional Memory Systems

What is a memory transaction?

- Sequence of instructions executing in memory
- Satisfying Atomicity

What is Software Transactional Memory?

- A parallel programming paradigm
- Avoids concurrency overheads at programmers level
- Execute code optimistically

Methods in STMs :

- *stm-begin()*
- *stm-read()*
- *stm-write()*
- *stm-tryC()*

Algorithm Insert(LL, v): Invoked by a thread to insert a value v into a linked-list LL. This method is implemented using transactions.

1: **while** ($true$) **do**
2:     *id* = *stm-begin()*;
3:     ...
4:     v = *stm-read(id, x)*;
5:     ...
6:     *stm-write(id, x, v')*;
7:     ...
8:     *ret* = *stm-tryC(id)*;
9:     **if** ($ret == success$) **then** break;
10:     **end if**
11: **end while**

Introduction to STMs Cont'd

Definition of Starvation-Free STMs

- A well known blocking progress condition associated with concurrent programming is starvation-freedom.

[a] Bushkov, V., Guerraoui, R., Kapalka, M.: On the liveness of transactional memory. In: ACM Symposium on PODC 2012.

- A well known blocking progress condition associated with concurrent programming is starvation-freedom.
- An STM system is said to be *starvation-free* if a thread invoking a non-parasitic [a] transaction T_i gets opportunity to retry T_i on every abort, due to the presence of a fair scheduler, then T_i will eventually commit.

[a] Bushkov, V., Guerraoui, R., Kapalka, M.: On the liveness of transactional memory. In: ACM Symposium on PODC 2012.

- Wait-freedom is another interesting progress condition for STMs in which every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler [b].

[b] Herlihy, M., Shavit, N.: On the nature of progress. OPODIS 2011.

[c] Bushkov, V., Guerraoui, R., Kapalka, M.: On the liveness of transactional memory. In: ACM Symposium on PODC 2012.

Introduction to STMs Cont'd

Definition of Starvation-Free STMs

- Wait-freedom is another interesting progress condition for STMs in which every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler [b].
- But it was shown by Guerraoui and Kapalka [c] that it is not possible to achieve wait-freedom in dynamic STMs in which data sets of transactions are not known in advance.

[b] Herlihy, M., Shavit, N.: On the nature of progress. OPODIS 2011.

[c] Bushkov, V., Guerraoui, R., Kapalka, M.: On the liveness of transactional memory. In: ACM Symposium on PODC 2012.

- Wait-freedom is another interesting progress condition for STMs in which every transaction commits regardless of the nature of concurrent transactions and the underlying scheduler [b].
- But it was shown by Guerraoui and Kapalka [c] that it is not possible to achieve wait-freedom in dynamic STMs in which data sets of transactions are not known in advance.
- So in this paper, we explore the weaker progress condition of starvation-freedom for transactional memories while assuming that the data sets of the transactions are not known in advance.

[b] Herlihy, M., Shavit, N.: On the nature of progress. OPODIS 2011.

[c] Bushkov, V., Guerraoui, R., Kapalka, M.: On the liveness of transactional memory. In: ACM Symposium on PODC 2012.

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

Related Work on Starvation-Free STMs

- Starvation-freedom in STMs has been explored by a few researchers in literature such as Gramoli et al. [d], Waliullah and Stenstrom [e], Spear et al. [f].

[d] Gramoli, V., Guerraoui, R., Trigonakis, V.: TM2C: A Software Transactional Memory for Many-cores. EuroSys 2012.

[e] Waliullah, M.M., Stenström, P.: Schemes for Avoiding Starvation in Transactional Memory Systems. Concurrency and Computation: Practice and Experience (2009).

[f] Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory (2009).

Related Work on Starvation-Free STMs

- Starvation-freedom in STMs has been explored by a few researchers in literature such as Gramoli et al. [d], Waliullah and Stenstrom [e], Spear et al. [f].
- Most of these systems work by assigning priorities to transactions.

---

[d] Gramoli, V., Guerraoui, R., Trigonakis, V.: TM2C: A Software Transactional Memory for Many-cores. EuroSys 2012.

[e] Waliullah, M.M., Stenström, P.: Schemes for Avoiding Starvation in Transactional Memory Systems. Concurrency and Computation: Practice and Experience (2009).

[f] Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory (2009).

Related Work on Starvation-Free STMs

- Starvation-freedom in STMs has been explored by a few researchers in literature such as Gramoli et al. [d], Waliullah and Stenstrom [e], Spear et al. [f].
- Most of these systems work by assigning priorities to transactions.
- In case of a conflict between two transactions, the transaction with lower priority is aborted.

[d] Gramoli, V., Guerraoui, R., Trigonakis, V.: TM2C: A Software Transactional Memory for Many-cores. EuroSys 2012.

[e] Waliullah, M.M., Stenström, P.: Schemes for Avoiding Starvation in Transactional Memory Systems. Concurrency and Computation: Practice and Experience (2009).

[f] Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory (2009).

Related Work on Starvation-Free STMs

- Starvation-freedom in STMs has been explored by a few researchers in literature such as Gramoli et al. [d], Waliullah and Stenstrom [e], Spear et al. [f].
- Most of these systems work by assigning priorities to transactions.
- In case of a conflict between two transactions, the transaction with lower priority is aborted.
- They ensure that every aborted transaction, on being retried a sufficient number of times, will eventually have the highest priority and hence will commit.

[d] Gramoli, V., Guerraoui, R., Trigonakis, V.: TM2C: A Software Transactional Memory for Many-cores. EuroSys 2012.

[e] Waliullah, M.M., Stenström, P.: Schemes for Avoiding Starvation in Transactional Memory Systems. Concurrency and Computation: Practice and Experience (2009).

[f] Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory (2009).

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

Motivation towards Multi-Version STMs

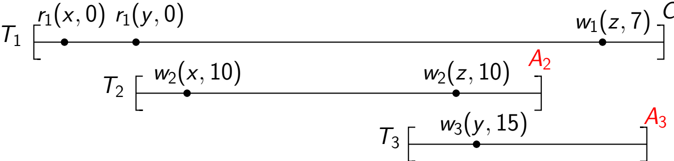

Figure: Limitation of Single-version Starvation Free Algorithm. T2 and T3 are aborted if T1 has the highest priority.

Motivation towards Multi-Version STMs

T_1 $r_1(x,0)$ $r_1(y,0)$ $w_1(z,7)$ C_1
T_2 $w_2(x,10)$ $w_2(z,10)$ A_2
T_3 $w_3(y,15)$ A_3

Figure: Limitation of Single-version Starvation Free Algorithm. T2 and T3 are aborted if T1 has the highest priority.

T_1 $r_1(x,0)$ $r_1(y,0)$ $w_1(z,7)$ C_1
T_2 $w_2(x,10)$ $w_2(z,10)$ C_2
T_3 $w_3(y,15)$ C_3

Figure: Advantage of Multi-version Starvation Free Algorithm.

Motivation towards Multi-Version STMs

T_1 $r_1(x,0)$ $r_1(y,0)$ $w_1(z,7)$ C_1
T_2 $w_2(x,10)$ $w_2(z,10)$ A_2
T_3 $w_3(y,15)$ A_3

Figure: Limitation of Single-version Starvation Free Algorithm. T2 and T3 are aborted if T1 has the highest priority.

T_1 $r_1(x,0)$ $r_1(y,0)$ $w_1(z,7)$ C_1
T_2 $w_2(x,10)$ $w_2(z,10)$ C_2
T_3 $w_3(y,15)$ C_3

Figure: Advantage of Multi-version Starvation Free Algorithm.

Takeaway

- Multi-version STMs reduce number of aborts and increases throughput.

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

Challenges for Multi-version STMs

- Although multi-version STMs provide greater concurrency, they suffer from the cost of garbage collection.

Challenges for Multi-version STMs

- Although multi-version STMs provide greater concurrency, they suffer from the cost of garbage collection.
- One way to avoid this is to use bounded-multi-version STMs, where the number of versions is bounded to be at most K.

Challenges for Multi-version STMs

- Although multi-version STMs provide greater concurrency, they suffer from the cost of garbage collection.
- One way to avoid this is to use bounded-multi-version STMs, where the number of versions is bounded to be at most K.
- Thus, when $(K+1)^{th}$ version is created, the oldest version is removed.

Challenges for Multi-version STMs

- Although multi-version STMs provide greater concurrency, they suffer from the cost of garbage collection.
- One way to avoid this is to use bounded-multi-version STMs, where the number of versions is bounded to be at most K.
- Thus, when $(K+1)^{th}$ version is created, the oldest version is removed.
- Furthermore, achieving starvation-freedom while using only bounded versions is especially challenging given that a transaction may rely on the oldest version that is removed.

Challenges for Multi-version STMs

- Although multi-version STMs provide greater concurrency, they suffer from the cost of garbage collection.
- One way to avoid this is to use bounded-multi-version STMs, where the number of versions is bounded to be at most K.
- Thus, when $(K+1)^{th}$ version is created, the oldest version is removed.
- Furthermore, achieving starvation-freedom while using only bounded versions is especially challenging given that a transaction may rely on the oldest version that is removed.
- In that case, it would be necessary to abort that transaction, making it harder to achieve starvation-freedom.

Challenges for Multi-version STMs

- Although multi-version STMs provide greater concurrency, they suffer from the cost of garbage collection.
- One way to avoid this is to use bounded-multi-version STMs, where the number of versions is bounded to be at most K.
- Thus, when $(K+1)^{th}$ version is created, the oldest version is removed.
- Furthermore, achieving starvation-freedom while using only bounded versions is especially challenging given that a transaction may rely on the oldest version that is removed.
- In that case, it would be necessary to abort that transaction, making it harder to achieve starvation-freedom.
- So, this paper addresses the gap by developing a starvation-free algorithm for bounded multi-version STMs as *KSFTM (K-Version Starvation-Free STM)*.

System Model and Preliminaries

- We assume a system of n asynchronous processes/threads, $p_1, \ldots, p_n$ that access a collection of *transactional objects* via atomic *transactions*.

System Model and Preliminaries

- We assume a system of n asynchronous processes/threads, $p_1, \ldots, p_n$ that access a collection of *transactional objects* via atomic *transactions*.
- Each transaction has a unique identifier.

System Model and Preliminaries

- We assume a system of n asynchronous processes/threads, $p_1, \ldots, p_n$ that access a collection of *transactional objects* via atomic *transactions*.
- Each transaction has a unique identifier.

Assumptions of Our Paper

Assumption (1)

We assume that in the absence of other concurrent conflicting transactions, every transaction will commit. In other words, (a) if a transaction T_i is executing in a system where other concurrent conflicting transactions are not present then T_i will not self-abort. (b) Transactions are not parasitic.

Assumptions of Our Paper

Assumption (1)

We assume that in the absence of other concurrent conflicting transactions, every transaction will commit. In other words, (a) if a transaction T_i is executing in a system where other concurrent conflicting transactions are not present then T_i will not self-abort. (b) Transactions are not parasitic.

Assumption (2)

Bounded-Termination: *For any transaction T_i, invoked by a thread Th_x, the fair system scheduler ensures, in the absence of deadlocks, Th_x is given sufficient time on a CPU (and memory etc.) such that T_i terminates (either commits or aborts) in bounded time, L.*

Our Contributions

[g] Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp Based Multi-version STM Algorithm. In: ICDCN 2014.

Our Contributions

We develop *KSFTM* algorithm in a step-wise manner starting from *MVTO* [g] *(Multi-Version Timestamp Order)* as follows:

[g] Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp Based Multi-version STM Algorithm. In: ICDCN 2014.

Our Contributions

We develop *KSFTM* algorithm in a step-wise manner starting from *MVTO* [g] *(Multi-Version Timestamp Order)* as follows:

Table: Comparison of the various Algorithms Developed

Algorithm	Starvation Freedom	Correctness
PKTO	No	Yes

[g] Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp Based Multi-version STM Algorithm. In: ICDCN 2014.

Our Contributions

We develop *KSFTM* algorithm in a step-wise manner starting from *MVTO* [g] *(Multi-Version Timestamp Order)* as follows:

Table: Comparison of the various Algorithms Developed

Algorithm	Starvation Freedom	Correctness
PKTO	No	Yes
SFKTO	Yes	No

[g] Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp Based Multi-version STM Algorithm. In: ICDCN 2014.

Our Contributions

We develop *KSFTM* algorithm in a step-wise manner starting from *MVTO* [g] *(Multi-Version Timestamp Order)* as follows:

Table: Comparison of the various Algorithms Developed

Algorithm	Starvation Freedom	Correctness
PKTO	No	Yes
SFKTO	Yes	No
KSFTM	Yes	Yes

[g]Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp Based Multi-version STM Algorithm. In: ICDCN 2014.

Our Contributions

We develop *KSFTM* algorithm in a step-wise manner starting from *MVTO* [g] *(Multi-Version Timestamp Order)* as follows:

Table: Comparison of the various Algorithms Developed

Algorithm	Starvation Freedom	Correctness
PKTO	No	Yes
SFKTO	Yes	No
KSFTM	Yes	Yes

- PKTO: Priority-based K-version Timestamp Order
- SFKTO: Starvation-free K-version Timestamp Order
- KSFTM: K-version starvation-free STM

[g] Kumar, P., Peri, S., Vidyasankar, K.: A TimeStamp Based Multi-version STM Algorithm. In: ICDCN 2014.

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

PKTO Algorithm

Main Idea

- Each transaction T_i has two timestamps:

PKTO Algorithm

Main Idea

- Each transaction T_i has two timestamps:
  1. *Current Timestamp (CTS):* This is a unique timestamp alloted to T_i when it begins.

PKTO Algorithm

Main Idea

- Each transaction T_i has two timestamps:
  1. *Current Timestamp (CTS):* This is a unique timestamp alloted to T_i when it begins.
  2. *Initial Timestamp (ITS):* This is same as CTS when a transaction T_i starts for the first time.

PKTO Algorithm

Main Idea

- Each transaction T_i has two timestamps:
  1. *Current Timestamp (CTS):* This is a unique timestamp alloted to T_i when it begins.
  2. *Initial Timestamp (ITS):* This is same as CTS when a transaction T_i starts for the first time.
- When T_i aborts and restarts later, it gets a new CTS.

PKTO Algorithm

Main Idea

- Each transaction T_i has two timestamps:
  1. *Current Timestamp (CTS):* This is a unique timestamp alloted to T_i when it begins.
  2. *Initial Timestamp (ITS):* This is same as CTS when a transaction T_i starts for the first time.
- When T_i aborts and restarts later, it gets a new CTS.
- But it retains its original CTS as ITS.

PKTO Algorithm

Main Idea

- Each transaction T_i has two timestamps:
  1. *Current Timestamp (CTS):* This is a unique timestamp alloted to T_i when it begins.
  2. *Initial Timestamp (ITS):* This is same as CTS when a transaction T_i starts for the first time.
- When T_i aborts and restarts later, it gets a new CTS.
- But it retains its original CTS as ITS.
- Timestamps are monotonically increasing.

PKTO Algorithm

Main Idea

- Each transaction T_i has two timestamps:
  1. *Current Timestamp (CTS):* This is a unique timestamp alloted to T_i when it begins.
  2. *Initial Timestamp (ITS):* This is same as CTS when a transaction T_i starts for the first time.
- When T_i aborts and restarts later, it gets a new CTS.
- But it retains its original CTS as ITS.
- Timestamps are monotonically increasing.
- The version created by a transaction T_i for a transaction object x is denoted x_i.

PKTO Algorithm Cont'd

Data Structure

Illustration of data structures maintained:

PKTO Algorithm Cont'd

Data Structure

Illustration of data structures maintained:

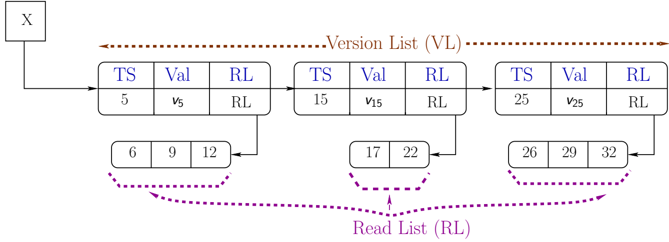

Figure: Data Structures for Maintaining Versions

PKTO Algorithm Cont'd

Illustration of Read Method

Read rule: T_i on invoking $r_i(x)$ reads the value v written by a transaction T_j that commits before $r_i(x)$ and j is the largest timestamp $\leq i$. After reading this version of x, T_i is stored in $x[j]$'s read list. If no such version exists then T_i returns *abort*.

PKTO Algorithm Cont'd

Illustration of Read Method

Read rule: T_i on invoking $r_i(x)$ reads the value v written by a transaction T_j that commits before $r_i(x)$ and j is the largest timestamp $\leq i$. After reading this version of x, T_i is stored in $x[j]$'s read list. If no such version exists then T_i returns *abort*.

Suppose transaction T_{14} wishes to read x:

Read rule: T_i on invoking $r_i(x)$ reads the value v written by a transaction T_j that commits before $r_i(x)$ and j is the largest timestamp $\leq i$. After reading this version of x, T_i is stored in $x[j]$'s read list. If no such version exists then T_i returns *abort*.

Suppose transaction T_{14} wishes to read x:

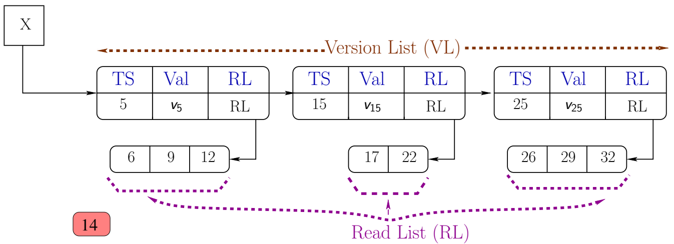

Figure: T_{14} wants to read x

PKTO Algorithm Cont'd

Illustration of Read Method

Read rule: T_i on invoking $r_i(x)$ reads the value v written by a transaction T_j that commits before $r_i(x)$ and j is the largest timestamp $\leq i$. After reading this version of x, T_i is stored in $x[j]$'s read list. If no such version exists then T_i returns *abort*.

Suppose transaction T_{14} wishes to read x:

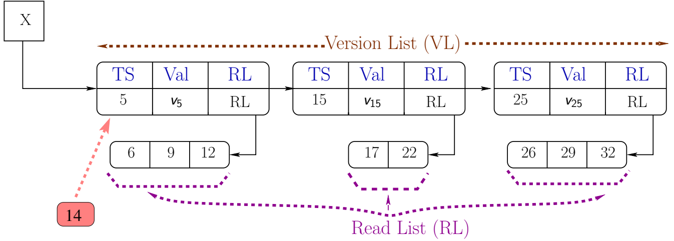

Figure: T_{14} identified the correct version with TS as 5

PKTO Algorithm Cont'd

Illustration of Read Method

Read rule: T_i on invoking $r_i(x)$ reads the value v written by a transaction T_j that commits before $r_i(x)$ and j is the largest timestamp $\leq i$. After reading this version of x, T_i is stored in $x[j]$'s read list. If no such version exists then T_i returns *abort*.

Suppose transaction T_{14} wishes to read x:

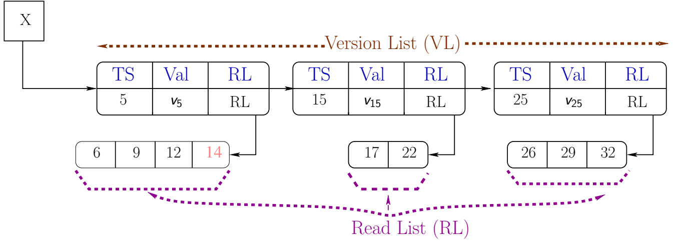

Figure: T_{14} added into read list (RL) of version with TS 5

Read rule: T_i on invoking $r_i(x)$ reads the value v written by a transaction T_j that commits before $r_i(x)$ and j is the largest timestamp $\leq i$. After reading this version of x, T_i is stored in $x[j]$'s read list. If no such version exists then T_i returns *abort*.

Suppose transaction T_4 wishes to read x:

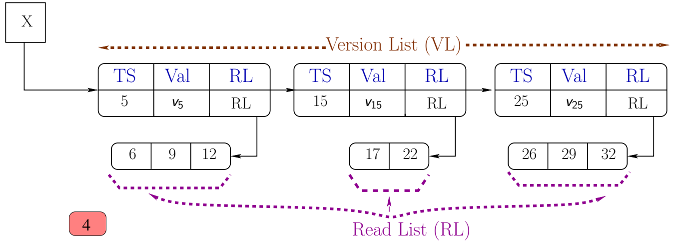

Figure: T_4 wants to read x

PKTO Algorithm Cont'd

Illustration of Read Method

Read rule: T_i on invoking $r_i(x)$ reads the value v written by a transaction T_j that commits before $r_i(x)$ and j is the largest timestamp $\leq i$. After reading this version of x, T_i is stored in $x[j]$'s read list. If no such version exists then T_i returns *abort*.

Suppose transaction T_4 wishes to read x:

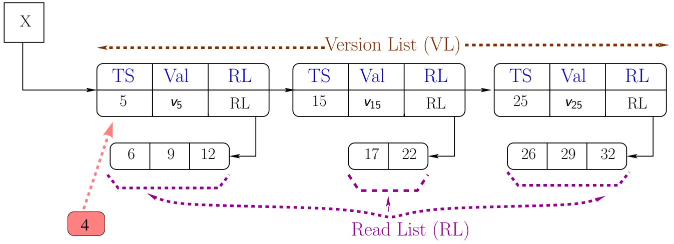

Figure: T_4 did not find the correct version to read

PKTO Algorithm Cont'd

Illustration of Read Method

Read rule: T_i on invoking $r_i(x)$ reads the value v written by a transaction T_j that commits before $r_i(x)$ and j is the largest timestamp $\leq i$. After reading this version of x, T_i is stored in $x[j]$'s read list. If no such version exists then T_i returns *abort*.

Suppose transaction T_4 wishes to read x:

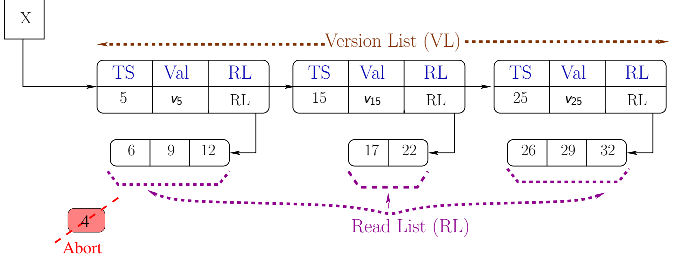

Figure: T_4 returns abort

PKTO Algorithm Cont'd

Illustration of Write/TryC Method

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

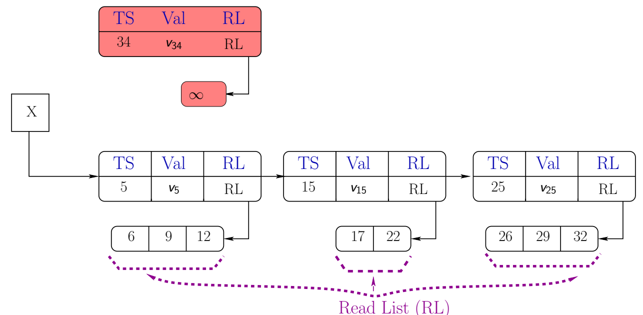

Figure: T_{34} wants to create a version of x (K=3)

PKTO Algorithm Cont'd

Illustration of Write/TryC Method

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

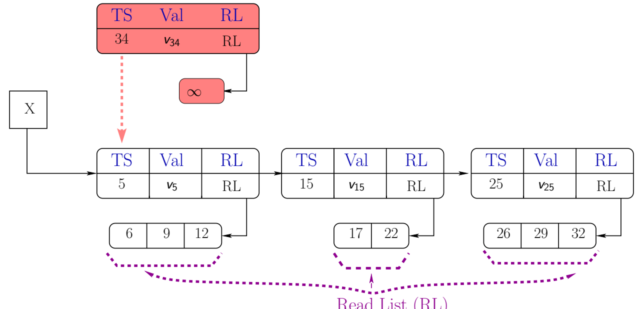

Figure: T_{34} identifying the correct version (K=3)

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

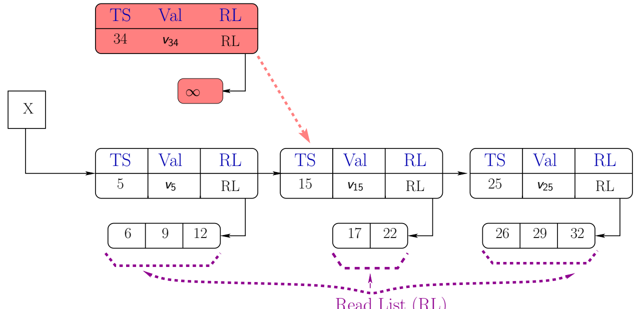

Figure: T_{34} identifying the correct version (K=3)

PKTO Algorithm Cont'd

Illustration of Write/TryC Method

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

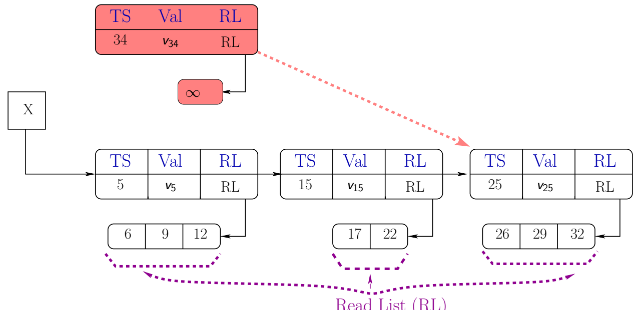

Figure: T_{34} identified the correct version as 25 (K=3)

PKTO Algorithm Cont'd

Illustration of Write/TryC Method

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

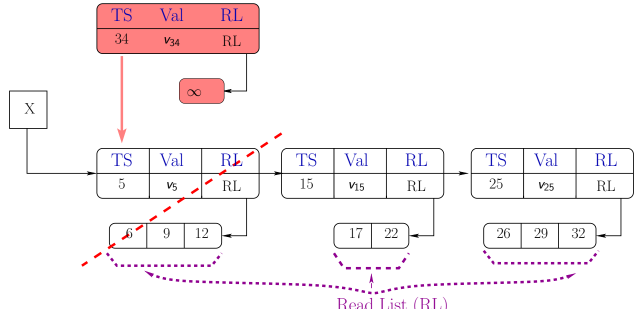

Figure: T_{34} replaced lowest version 5 with 34 (K=3)

PKTO Algorithm Cont'd

Illustration of Write/TryC Method

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

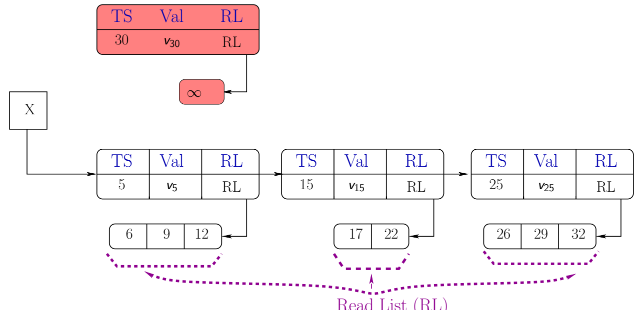

Figure: T_{30} wants to create a version of x (K=3)

PKTO Algorithm Cont'd

Illustration of Write/TryC Method

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

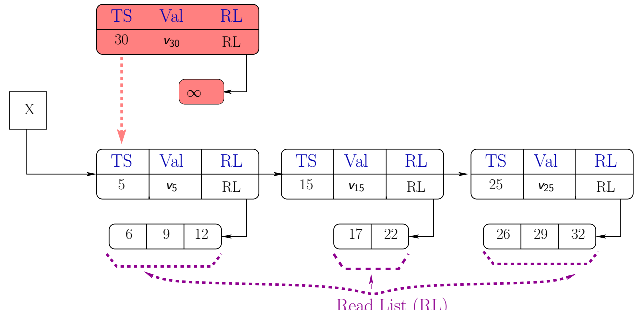

Figure: T_{30} identifying the correct version (K=3)

PKTO Algorithm Cont'd

Illustration of Write/TryC Method

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

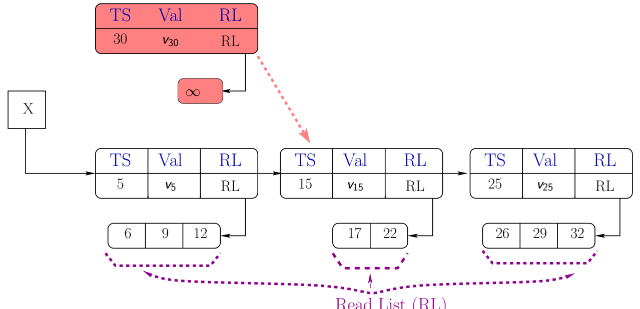

Figure: T_{30} identifying the correct version (K=3)

PKTO Algorithm Cont'd

Illustration of Write/TryC Method

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

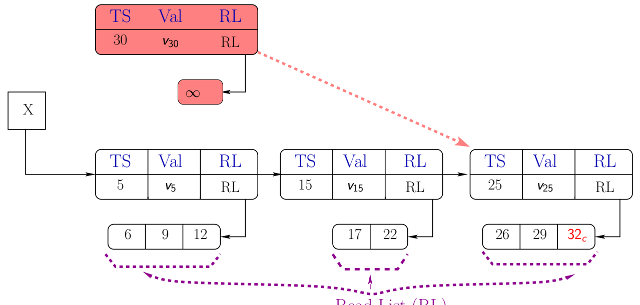

Figure: T_{30} identified the correct version 25 (K=3)

Commit rule: T_i on invoking tryC operation checks for each transaction object x, in its *Wset*:

1. If a transaction T_k has read x from T_j and committed, with $j < i < k$, then T_i returns abort. Otherwise abort T_k.
2. If such version T_j does not exist and reach the limit of K-version then T_i returns abort. Otherwise, the transaction is allowed to commit.

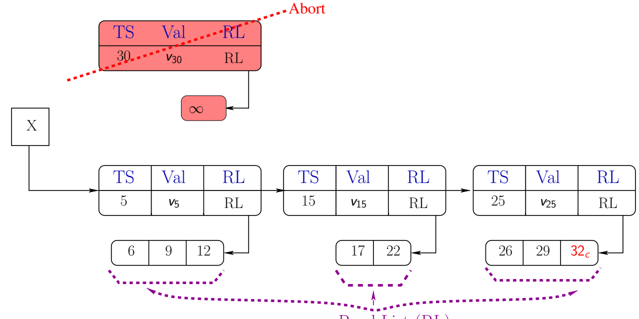

Figure: T_{30} returns abort because 32 has been committed in RL of 25 (K=3)

PKTO Algorithm Cont'd

Garbage Collection

The *PKTO* algorithm uses garbage collection method to deletes a version x created by transaction T_i if:

PKTO Algorithm Cont'd

Garbage Collection

The *PKTO* algorithm uses garbage collection method to deletes a version x created by transaction T_i if:

1. At least one another version of x has been created by T_k and $i < k$.

PKTO Algorithm Cont'd

Garbage Collection

The *PKTO* algorithm uses garbage collection method to deletes a version x created by transaction T_i if:

1. At least one another version of x has been created by T_k and $i < k$.
2. Let T_k be the transaction that has the smallest timestamp larger than i and has created a version of x. Then for every j such that $i < j < k$, T_j has terminated (either committed or aborted).

PKTO Algorithm Cont'd

Drawback

The history generated by *PKTO* algorithm satisfies *strict-serializability* [6] and local-opacity [3, 4]. But it does not ensures starvation-freedom.

The history generated by *PKTO* algorithm satisfies *strict-serializability* [6] and local-opacity [3, 4]. But it does not ensures starvation-freedom.

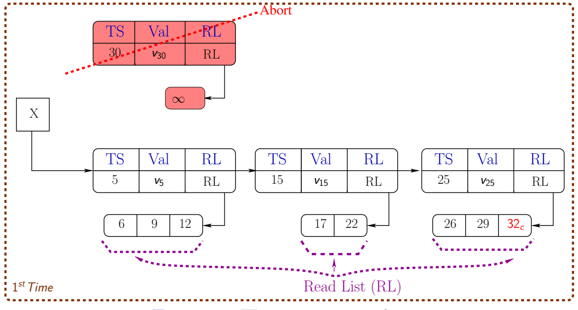

Figure: T_{30} returns abort

PKTO Algorithm Cont'd

Drawback

The history generated by *PKTO* algorithm satisfies *strict-serializability* [6] and local-opacity [3, 4]. But it does not ensures starvation-freedom.

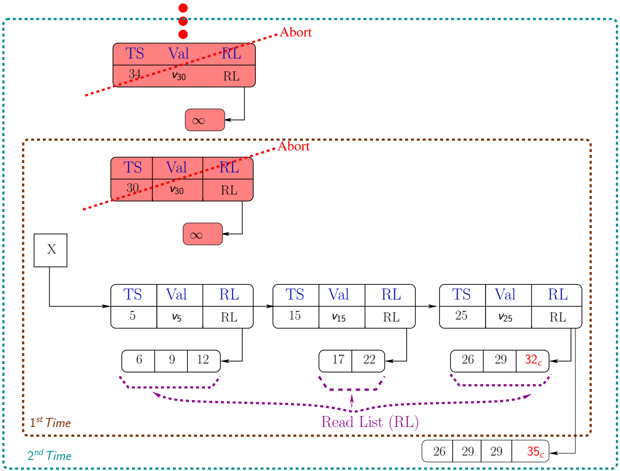

Figure: T_{30} is starving

Key Insights for Eliminating Starvation in PKTO

Permutations of operations

T_{50} $r_{50}(x, 40)$ $w_{50}(x, 50)$ C_{50}

T_{60} $r_{60}(x, 50)$ $w_{60}(x, 60)$ C_{60}

Figure: T_{60} reads the version written by T_{50}. No conflict.

Key Insights for Eliminating Starvation in PKTO

Permutations of operations

T_{50} $r_{50}(x, 40)$ $w_{50}(x, 50)$ C_{50}

T_{60} $r_{60}(x, 50)$ $w_{60}(x, 60)$ C_{60}

Figure: T_{60} reads the version written by T_{50}. No conflict.

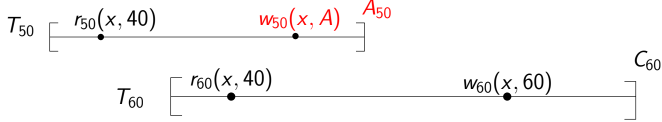

Figure: Conflict detected at w_{50}. Either abort T_{50} or T_{60}

Key Insights for Eliminating Starvation in PKTO

Permutations of operations

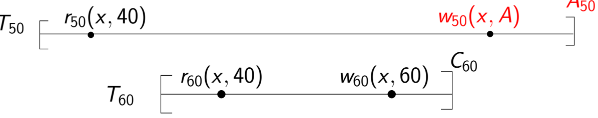

Figure: Conflict detected at w_{50}. Hence, abort T_{50}

Key Insights for Eliminating Starvation in PKTO

Permutations of operations

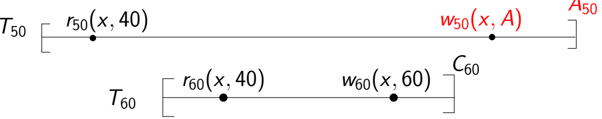

Figure: Conflict detected at w_{50}. Hence, abort T_{50}

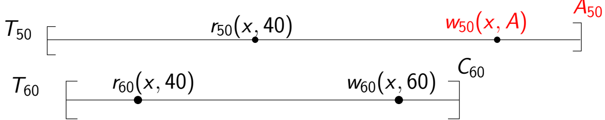

Figure: Conflict detected at w_{50}. Hence, abort T_{50}

Key Insights for Eliminating Starvation in PKTO

Permutations of operations

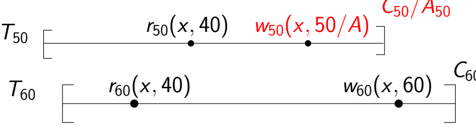

Figure: Conflict detected at w_{50}. Abort T_{50}

Key Insights for Eliminating Starvation in PKTO

Permutations of operations

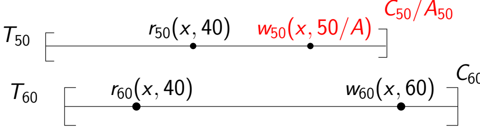

Figure: Conflict detected at w_{50}. Abort T_{50}

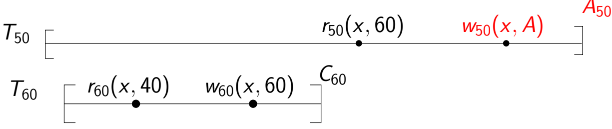

Figure: Conflict detected at w_{50}. Hence, abort T_{50}

Key Insights for Eliminating Starvation in PKTO

Takeaway

Key Observation

A transaction cannot be aborted if

- It has the lowest ITS, i.e., it is the earliest transaction and
- It has the highest CTS

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

- A transaction T_i instead of using the current time as CTS_i, uses a potentially a unique higher timestamp, *Working Timestamp - WTS* or WTS_i.

- A transaction T_i instead of using the current time as CTS_i, uses a potentially a unique higher timestamp, *Working Timestamp - WTS* or WTS_i.
- Specifically, it adds $C * (CTS_i - ITS_i)$ to CTS_i, i.e.,

$$WTS_i = CTS_i + C * (CTS_i - ITS_i);$$

where, C is any constant greater than 0.

SFKTO Algorithm

Modifying *PKTO* to Achieve Starvation-Freedom

- A transaction T_i instead of using the current time as CTS_i, uses a potentially a unique higher timestamp, *Working Timestamp - WTS* or WTS_i.
- Specifically, it adds $C * (CTS_i - ITS_i)$ to CTS_i, i.e.,

$$WTS_i = CTS_i + C * (CTS_i - ITS_i);$$

  where, C is any constant greater than 0.
- In other words, when the transaction T_i is issued for the first time, WTS_i is same as $CTS_i = ITS_i$.

SFKTO Algorithm

Modifying *PKTO* to Achieve Starvation-Freedom

- A transaction T_i instead of using the current time as CTS_i, uses a potentially a unique higher timestamp, *Working Timestamp - WTS* or WTS_i.
- Specifically, it adds $C * (CTS_i - ITS_i)$ to CTS_i, i.e.,

  $$WTS_i = CTS_i + C * (CTS_i - ITS_i);$$

  where, C is any constant greater than 0.
- In other words, when the transaction T_i is issued for the first time, WTS_i is same as $CTS_i = ITS_i$.
- However, as transaction keeps getting aborted, the drift between CTS_i and WTS_i increases. The value of WTS_i increases with each retry.

SFKTO Algorithm

Modifying *PKTO* to Achieve Starvation-Freedom

- A transaction T_i instead of using the current time as CTS_i, uses a potentially a unique higher timestamp, *Working Timestamp - WTS* or WTS_i.
- Specifically, it adds $C * (CTS_i - ITS_i)$ to CTS_i, i.e.,

  $$WTS_i = CTS_i + C * (CTS_i - ITS_i);$$

  where, C is any constant greater than 0.
- In other words, when the transaction T_i is issued for the first time, WTS_i is same as $CTS_i = ITS_i$.
- However, as transaction keeps getting aborted, the drift between CTS_i and WTS_i increases. The value of WTS_i increases with each retry.
- Eventually, a transaction with lowest ITS will finally have the highest WTS.

SFKTO Algorithm Cont'd

Drawback

- *SFKTO* uses *WTS* which may not correspond to the real-time, as *WTS* may be significantly larger than *CTS*.

SFKTO Algorithm Cont'd

Drawback

- *SFKTO* uses *WTS* which may not correspond to the real-time, as *WTS* may be significantly larger than *CTS*.
- Hence, the history generated by *SFKTO* algorithm ensures starvation-freedom but does not satisfies strict-serializability and local opacity.

SFKTO Algorithm Cont'd

Drawback

- *SFKTO* uses *WTS* which may not correspond to the real-time, as *WTS* may be significantly larger than *CTS*.
- Hence, the history generated by *SFKTO* algorithm ensures starvation-freedom but does not satisfies strict-serializability and local opacity.

Figure: Execution under SFKTO with equivalent serial schedule $T_1 T_3 T_2$

SFKTO Algorithm Cont'd

Drawback

- *SFKTO* uses *WTS* which may not correspond to the real-time, as *WTS* may be significantly larger than *CTS*.
- Hence, the history generated by *SFKTO* algorithm ensures starvation-freedom but does not satisfies strict-serializability and local opacity.

$CTS_1 = 50$, $WTS_1 = 50$ — T_1: $r_1(x,0)$, $w_1(x,10)$, C_1

$CTS_2 = 60$, $WTS_2 = 100$ — T_2: $r_2(y,0)$, $w_2(x,20)$, C_2

$CTS_3 = 80$, $WTS_3 = 80$ — T_3: $r_3(x,10)$, $r_3(z,25)$, C_3

Figure: Execution under SFKTO with equivalent serial schedule $T_1 T_3 T_2$

- We use the idea of timestamp ranges to follow the real-time order and proposes *KSFTM*.

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

KSFTM Algorithm

K-Version Starvation-Freedom Transactional Memory (KSFTM)

- This is the combination of *PKTO* and *SFKTO* that provides both correctness (strict-serializability and local-opacity) as well as starvation-freedom.

- This is the combination of *PKTO* and *SFKTO* that provides both correctness (strict-serializability and local-opacity) as well as starvation-freedom.
- Each transaction T_i uses three timestamps, *ITS, CTS and WTS*.

KSFTM Algorithm

K-Version Starvation-Freedom Transactional Memory (KSFTM)

- This is the combination of *PKTO* and *SFKTO* that provides both correctness (strict-serializability and local-opacity) as well as starvation-freedom.
- Each transaction T_i uses three timestamps, *ITS, CTS and WTS*.
- Along with this, each transaction T_i maintains a timestamp range: *Transaction Lower Timestamp Limit* or $TLTL_i$, and *Transaction Upper Timestamp Limit* or $TUTL_i$ to ensure real-time order.

- This is the combination of *PKTO* and *SFKTO* that provides both correctness (strict-serializability and local-opacity) as well as starvation-freedom.
- Each transaction T_i uses three timestamps, *ITS, CTS and WTS*.
- Along with this, each transaction T_i maintains a timestamp range: *Transaction Lower Timestamp Limit* or $TLTL_i$, and *Transaction Upper Timestamp Limit* or $TUTL_i$ to ensure real-time order.
- When T_i aborts and restarts later, it gets a new WTS. But it retains its original WTS as ITS.

KSFTM Algorithm

K-Version Starvation-Freedom Transactional Memory (KSFTM)

- This is the combination of *PKTO* and *SFKTO* that provides both correctness (strict-serializability and local-opacity) as well as starvation-freedom.
- Each transaction T_i uses three timestamps, *ITS, CTS and WTS*.
- Along with this, each transaction T_i maintains a timestamp range: *Transaction Lower Timestamp Limit* or $TLTL_i$, and *Transaction Upper Timestamp Limit* or $TUTL_i$ to ensure real-time order.
- When T_i aborts and restarts later, it gets a new WTS. But it retains its original WTS as ITS.
- It maintains K versions corresponding to each objects.

KSFTM Algorithm Cont'd

K-Version Starvation-Freedom Transactional Memory (KSFTM)

A transaction T_i invokes the methods as follows:

- *stm-begin()*:
    1. Initialize $\langle ITS, WTS, TLTL \rangle = \text{CTS}$
    2. $\text{TUTL} = \infty$
- *stm-read()* or *stm-tryC()*:
    - Increments the TLTL
    - Decrements the TUTL
    - **If**$(\text{TLTL} > \text{TUTL})$ **then** T_i returns abort.

KSFTM Algorithm Cont'd

Execution

$CTS_1 = 50$
$WTS_1 = 50$
$TLTL_1 = 50$
$TUTL_1 = \infty$

T_1: $r_1(x, 0)$, $w_1(x, 10)$, C_1

$TLTL_1 = 70$
$TUTL_1 = 70$

$CTS_2 = 60$
$WTS_2 = 100$
$TLTL_2 = 60$
$TUTL_2 = \infty$

T_2: $r_2(y, 0)$, $w_2(x, 20)$, C_2

$TLTL_2 = 71$
$TUTL_2 = 75$

$CTS_3 = 80$
$WTS_3 = 80$
$TLTL_3 = 80$
$TUTL_3 = \infty$

T_3: $r_3(x, 10)$, $r_3(z, 25)$, A_3

$TLTL_3 = 80$
$TUTL_3 = 70$

Figure: Correct execution under KSFTM with equivalent serial schedule $T_1 T_2$

KSFTM Algorithm Cont'd

Correctness[h]

Theorem (1)

Any history generated by KSFTM is strict-serializable and locally-opaque.

[h] For more detail please refer technical report link: https://arxiv.org/abs/1709.01033

KSFTM Algorithm Cont'd

Correctness[h]

Theorem (1)

Any history generated by KSFTM is strict-serializable and locally-opaque.

Theorem (2)

Any transaction with lowest ITS and highest WTS will never abort. So, KSFTM ensure starvation freedom.

[h] For more detail please refer technical report link: https://arxiv.org/abs/1709.01033

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

- The experimental system is a 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and 2 hyper-threads (HTs) per core, for a total of 56 threads.

- The experimental system is a 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and 2 hyper-threads (HTs) per core, for a total of 56 threads.
- The machine has 32GB of RAM and runs Ubuntu 16.04.2 LTS.

- The experimental system is a 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz with 14 cores per socket and 2 hyper-threads (HTs) per core, for a total of 56 threads.
- The machine has 32GB of RAM and runs Ubuntu 16.04.2 LTS.
- We consider counter application for three workloads as follows:
  1. (W1) Lookup intensive (90% read, 10% write),
  2. (W2) Mid intensive (50% read, 50% write) and
  3. (W3) Ui - Update intensive (10% read, 90% write).

Comparison of KSFTM with State-of-The-Art STMs

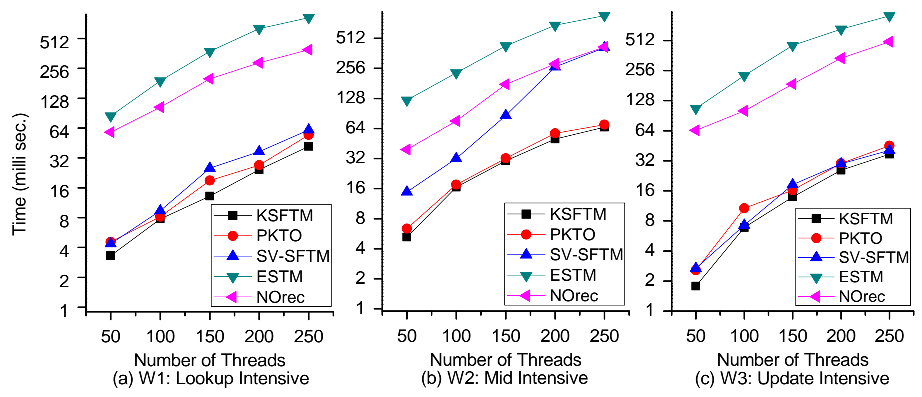

Figure: Performance analysis on workload $W1$, $W2$, $W3$

Comparison of KSFTM with State-of-The-Art STMs

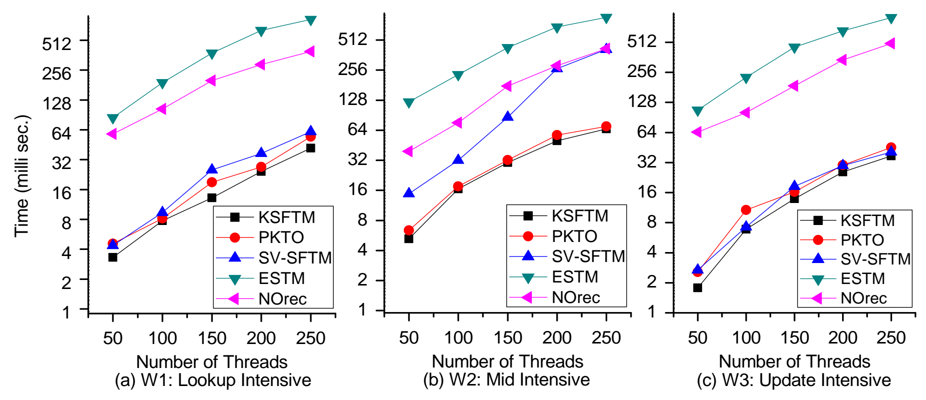

Figure: Performance analysis on workload $W1$, $W2$, $W3$

- KSFTM gives an average speedup on the worst-case time to commit of a transaction by a factor of 1.22, 1.89, 23.26 and 13.12 times over PKTO, SV-SFTM, NOrec STM [1] and ESTM [2] respectively for counter application.

Comparison of KSFTM with STAMP Benchmark

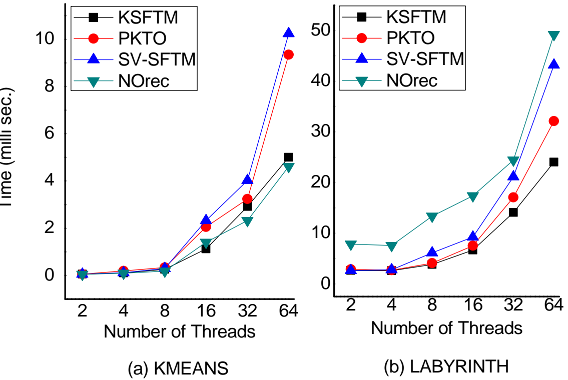

Figure: Performance analysis on KMEANS, LABYRINTH and KSFTM's Stability

Comparison of KSFTM with STAMP Benchmark

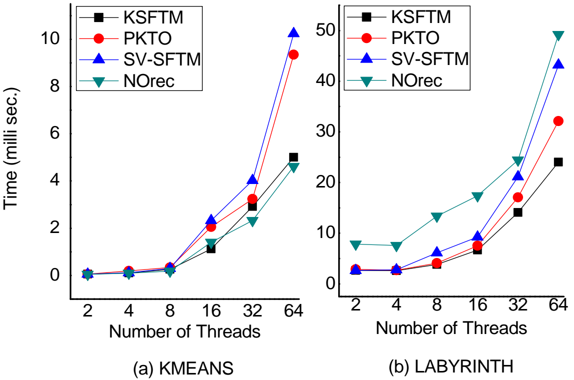

Figure: Performance analysis on KMEANS, LABYRINTH and KSFTM's Stability

- KSFTM performs 1.5 and 1.44 times better than PKTO and SV-SFTM but 1.09 times worse than NOrec for low contention KMEANS application of STAMP [5] benchmark.

Comparison of KSFTM with STAMP Benchmark

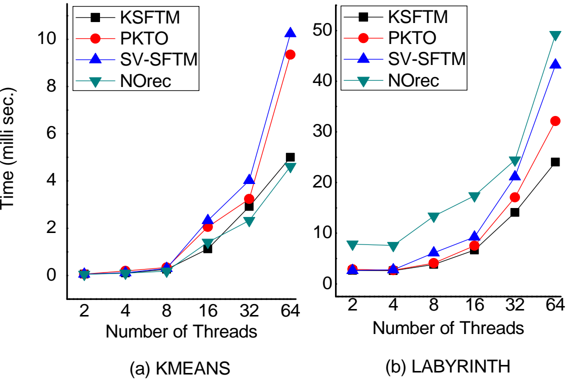

Figure: Performance analysis on KMEANS, LABYRINTH and KSFTM's Stability

- KSFTM performs 1.5 and 1.44 times better than PKTO and SV-SFTM but 1.09 times worse than NOrec for low contention KMEANS application of STAMP [5] benchmark.
- Whereas, KSFTM performs 1.14, 1.4 and 2.63 times better than PKTO, SV-SFTM and NOrec for LABYRINTH application

Outline

1. Introduction to STMs
2. Related Work on Starvation-Free STMs
3. Motivation towards Multi-Version STMs
4. Challenges for Multi-version STMs
5. PKTO Algorithm
6. SFKTO Algorithm
7. KSFTM Algorithm
8. Performance Analysis
9. Conclusion and Future Work

Conclusion and Future Work

- Motivation towards MVSTMs.

Conclusion and Future Work

- Motivation towards MVSTMs.
- We proposed *KSFTM* which ensures starvation-freedom while maintaining K versions for each transaction objects.

Conclusion and Future Work

- Motivation towards MVSTMs.
- We proposed *KSFTM* which ensures starvation-freedom while maintaining K versions for each transaction objects.
- It uses two insights to ensure starvation-freedom in the context of MVSTMs: (1) using ITS to ensure that older transactions are given a higher priority, and (2) using WTS to ensure that conflicting transactions do not commit too quickly before the older transaction could commit.

Conclusion and Future Work

- Motivation towards MVSTMs.
- We proposed *KSFTM* which ensures starvation-freedom while maintaining K versions for each transaction objects.
- It uses two insights to ensure starvation-freedom in the context of MVSTMs: (1) using ITS to ensure that older transactions are given a higher priority, and (2) using WTS to ensure that conflicting transactions do not commit too quickly before the older transaction could commit.
- We show KSFTM satisfies strict-serializability and local-opacity.

Conclusion and Future Work

- Motivation towards MVSTMs.
- We proposed *KSFTM* which ensures starvation-freedom while maintaining K versions for each transaction objects.
- It uses two insights to ensure starvation-freedom in the context of MVSTMs: (1) using ITS to ensure that older transactions are given a higher priority, and (2) using WTS to ensure that conflicting transactions do not commit too quickly before the older transaction could commit.
- We show KSFTM satisfies strict-serializability and local-opacity.
- Our experiments show that KSFTM performs better than starvation-free state-of-the-arts STMs as well as non-starvation free STMs under long running transactions with high contention workloads.

Conclusion and Future Work

- Motivation towards MVSTMs.
- We proposed *KSFTM* which ensures starvation-freedom while maintaining K versions for each transaction objects.
- It uses two insights to ensure starvation-freedom in the context of MVSTMs: (1) using ITS to ensure that older transactions are given a higher priority, and (2) using WTS to ensure that conflicting transactions do not commit too quickly before the older transaction could commit.
- We show KSFTM satisfies strict-serializability and local-opacity.
- Our experiments show that KSFTM performs better than starvation-free state-of-the-arts STMs as well as non-starvation free STMs under long running transactions with high contention workloads.
- **Future Work:** Nesting, Starvation-Free Object Semantic.

References

Luke Dalessandro, Michael F. Spear, and Michael L. Scott.
NOrec: Streamlining STM by Abolishing Ownership Records.
PPoPP 2010, 2010.

Pascal Felber, Vincent Gramoli, and Rachid Guerraoui.
Elastic transactions.
J. Parallel Distrib. Comput., 100(C):103–127, February 2017.

Petr Kuznetsov and Sathya Peri.
Non-interference and Local Correctness in Transactional Memory.
In *ICDCN*, pages 197–211, 2014.

Petr Kuznetsov and Sathya Peri.
Non-interference and local correctness in transactional memory.
Theor. Comput. Sci., 688, 2017.

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun.
STAMP: stanford transactional applications for multi-processing.

Thanks for Listening
Question Please!!

Comparison among variants of KSFTM

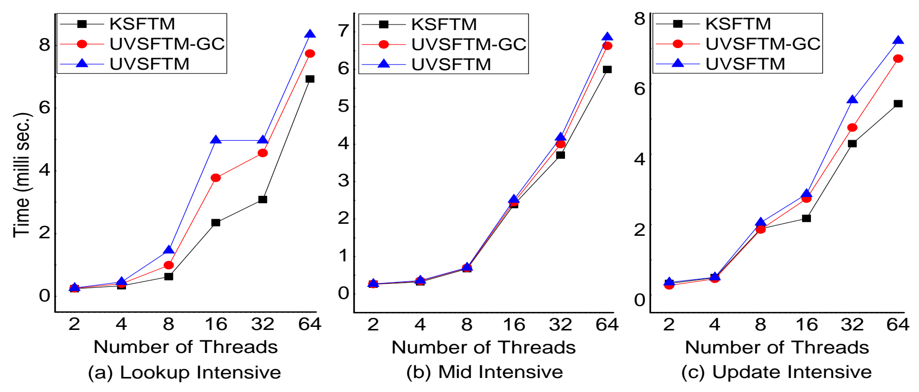

Figure: Time comparison among variants of KSFTM

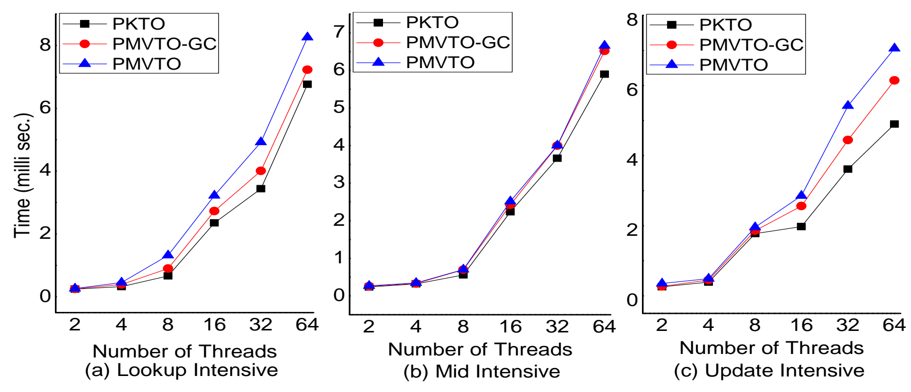

Figure: Time comparison among variants of PKTO

Comparison of Abort Count

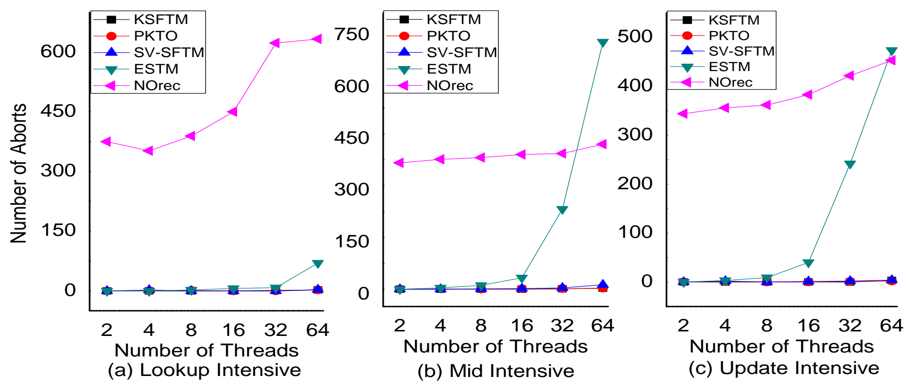

Figure: Abort Count on workload $W1, W2, W3$