Software Transactional Memory (STM)

Nir Shavit, Dan Touitou

Presented by
Ramesh Adhikari
Graduate Research Assistant
School of Computer and Cyber Sciences, Augusta University

March 2023

Introduction

- Software Transactional Memory (STM) is a concurrency control mechanism.

Introduction

- Software Transactional Memory (STM) is a concurrency control mechanism.
- STM allows multiple threads to safely share and update memory without the need for traditional synchronization mechanisms like locks or semaphores.

Introduction

- Software Transactional Memory (STM) is a concurrency control mechanism.
- STM allows multiple threads to safely share and update memory without the need for traditional synchronization mechanisms like locks or semaphores.
- STM works by defining a transactional region of code, which groups together a set of memory accesses that should be executed atomically.

Motivation

- Concurrent programming is essential to improve performance on a multi-core.

Motivation

- Concurrent programming is essential to improve performance on a multi-core.
- Locks and condition variables are fundamentally flawed and difficult to manage (Deadlock, Starvation, etc.)

Motivation

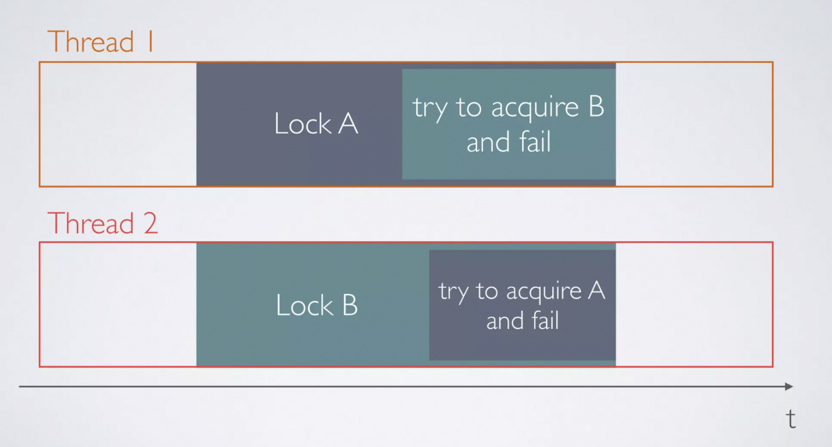
- Concurrent programming is essential to improve performance on a multi-core.
- Locks and condition variables are fundamentally flawed and difficult to manage (Deadlock, Starvation, etc.)
- Hard to avoid — especially when dealing with large, complex applications

Motivation

- Concurrent programming is essential to improve performance on a multi-core.
- Locks and condition variables are fundamentally flawed and difficult to manage (Deadlock, Starvation, etc.)
- Hard to avoid — especially when dealing with large, complex applications
- Proposed solution **Transactional memory** using concept of transaction

Issue in lock: Deadlock

A situation where two or more processes are waiting for each other



Transactional Programming

```
void deposit(account, amount) {
  lock(account);
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
  unlock(account);
}
```

```
void deposit(account, amount) {
  atomic {
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
  }
}
```

Background: Transaction

- A sequence of operations executed as an atomic unit of work
  - **Read-transactional**: reads the value of a shared location into a local register.
  - **Write-transactional**: stores the contents of a local register into a shared location.

Background: Transaction

- A sequence of operations executed as an atomic unit of work
  - **Read-transactional**: reads the value of a shared location into a local register.
  - **Write-transactional**: stores the contents of a local register into a shared location.
- The **data set** of a transaction is the set of shared locations accessed by the Read-transactional and write-transactional instructions

Background: Transaction

- A sequence of operations executed as an atomic unit of work
  - **Read-transactional**: reads the value of a shared location into a local register.
  - **Write-transactional**: stores the contents of a local register into a shared location.
- The **data set** of a transaction is the set of shared locations accessed by the Read-transactional and write-transactional instructions
- Any transaction may either by **commits**: all of its updates (changes) are visible atomically to other processes. or by **aborts**: has no effect (typically restarted)

Related Work

- **Hardware method:** Herlihy and Moss have proposed an ingenious hardware solution: transactional memory. By **adding a specialized associative cache** and making several minor changes to the cache consistency protocols.

Related Work

- **Hardware method:** Herlihy and Moss have proposed an ingenious hardware solution: transactional memory. By **adding a specialized associative cache** and making several minor changes to the cache consistency protocols.
- **Cooperative method:** whenever a process needs (depends on) a location already locked by another process it helps the locking process to complete its own operation, and this is done **recursively** along the dependency chain.

Issue on recursive Cooperative method

- Suppose Process **P** does a Compare&Swap on locations **a** and **b**, but process **Q** already owns **b**.

Issue on recursive Cooperative method

- Suppose Process **P** does a Compare&Swap on locations **a** and **b**, but process **Q** already owns **b**.
- In the cooperative method, **P** must help **Q** before acquiring and continuing with its operation.

Issue on recursive Cooperative method

- Suppose Process **P** does a Compare&Swap on locations **a** and **b**, but process **Q** already owns **b**.
- In the cooperative method, **P** must help **Q** before acquiring and continuing with its operation.
- However, **P**'s operation may fail if **Q** changes **b** after **P** reads it, requiring all processes waiting for **a** to help **P**, then **Q**, and again **P**.

Issue on recursive Cooperative method

- Suppose Process **P** does a Compare&Swap on locations **a** and **b**, but process **Q** already owns **b**.
- In the cooperative method, **P** must help **Q** before acquiring and continuing with its operation.
- However, **P**'s operation may fail if **Q** changes **b** after **P** reads it, requiring all processes waiting for **a** to help **P**, then **Q**, and again **P**.
- Moreover, after **P** has acquired **b**, all processes requesting **b** will redundantly help **P**.

Issue on recursive Cooperative method

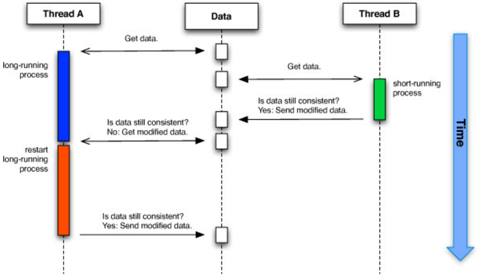
- Suppose Process **P** does a Compare&Swap on locations **a** and **b**, but process **Q** already owns **b**.
- In the cooperative method, **P** must help **Q** before acquiring and continuing with its operation.
- However, **P**'s operation may fail if **Q** changes **b** after **P** reads it, requiring all processes waiting for **a** to help **P**, then **Q**, and again **P**.
- Moreover, after **P** has acquired **b**, all processes requesting **b** will redundantly help **P**.
- **Alternatively**, if **P** uses STM, it will fail to acquire **b**, release **a** help **Q**, and restart.

Issue on recursive Cooperative method

- Suppose Process **P** does a Compare&Swap on locations **a** and **b**, but process **Q** already owns **b**.
- In the cooperative method, **P** must help **Q** before acquiring and continuing with its operation.
- However, **P**'s operation may fail if **Q** changes **b** after **P** reads it, requiring all processes waiting for **a** to help **P**, then **Q**, and again **P**.
- Moreover, after **P** has acquired **b**, all processes requesting **b** will redundantly help **P**.
- **Alternatively**, if **P** uses STM, it will fail to acquire **b**, release **a** help **Q**, and restart.
- Processes waiting for **a** will only help **P**, and those waiting for **b** won't have to help **P**.

Software Transactional Memory example

- **STM** relies on **transactions**, which are sequences of memory accesses that are executed atomically.
- If two transactions attempt to modify the same memory location concurrently, one of them will be aborted and retried later.

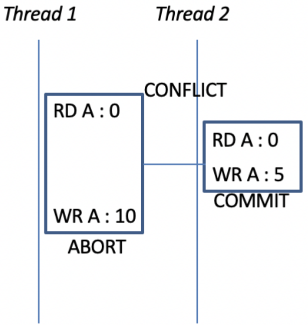


Transactional Memory- Atomicity and Isolation

Atomicity – On transaction **commit**, all memory updates appear to take effect at once; on transaction **abort**, none of the memory updates appear to take effect.

Isolation – No other code can observe updates before the commit.

```
void deposit(account, amount) {
  atomic {
    int t = bank.get(account); RD
    t = t + amount;
    bank.put(account, t);      WR
  }
}
```



Software Transactional Memory

- Based on implementing a k-word compare/swap using LL/SC instructions.
- Transactions access a pre-determined sequence of locations (static transaction).

```
k_word_C&S(Size, DataSet[], Old[ ], New[])
  BeginTransaction
    for i = 1 to Size do
        if Read-transactional(Dataset[i]]) ≠ Old[i]
          ReturnedValue = C&S-Failure
        ExitTransaction
    for i = 1 to Size do
      Write-transactional(DataSet[i], New[i])
    ReturnedValue = C&S-Success
  EndTransaction
end k_word_C&S
```

Fig. 2. A static transaction

STM system model: possible operations

In STM model, every shared memory location l of a multiprocessor machine's memory is formally modeled as an object which provides every processor i=1 … n four types of possible operations

1. $Read^i(l, v)$: reads shared memory location (l) and returns its value v.

STM system model: possible operations

In STM model, every shared memory location l of a multiprocessor machine's memory is formally modeled as an object which provides every processor i=1 ... n four types of possible operations

1. $Read^i(l, v)$: reads shared memory location (l) and returns its value v.
2. Load_Linked$^i(l, v)$: reads location l and returns its value v. Marks location l as "read by i".

STM system model: possible operations

In STM model, every shared memory location l of a multiprocessor machine's memory is formally modeled as an object which provides every processor i=1 ... n four types of possible operations

1. $Read^i(l, v)$: reads shared memory location (l) and returns its value v.
2. Load_Linked$^i(l, v)$: reads location l and returns its value v. Marks location l as "read by i".
3. Store_Conditional$^i(l, v)$: if location l is marked as "read by i," the operation writes the value v to l, erases all existing marks by other processors on l and returns a success status. Otherwise returns a failure status.

STM system model: possible operations

In STM model, every shared memory location l of a multiprocessor machine's memory is formally modeled as an object which provides every processor i=1 … n four types of possible operations

1. $Read^i(l, v)$: reads shared memory location (l) and returns its value v.
2. Load_Linked$^i(l, v)$: reads location l and returns its value v. Marks location l as "read by i".
3. Store_Conditional$^i(l, v)$: if location l is marked as "read by i," the operation writes the value v to l, erases all existing marks by other processors on l and returns a success status. Otherwise returns a failure status.
4. Write$^i(l, v)$: writes the value v to location l, erases all existing "read by" marks by other processors on l.

Non-blocking STM

1. Non-blocking means that executing a transaction repeatedly by a process will not block the whole system.

Non-blocking STM

1. Non-blocking means that executing a transaction repeatedly by a process will not block the whole system.
2. Non-blocking ensures that the system can always make progress

Non-blocking STM

1. Non-blocking means that executing a transaction repeatedly by a process will not block the whole system.
2. Non-blocking ensures that the system can always make progress
3. If a process repeatedly attempts to execute some transaction implies that some process (not necessarily the same one and with a possibly different transaction) will terminate successfully after a finite number of machine steps in the whole system

A non-blocking implementation of STM

Author implements a non-blocking static STM of size M using following data structure:

1. Memory [M], a vector which contains the data stored in the transactional memory.
2. Ownerships [M], a vector which determines for any cell in Memory, which transaction owns it.

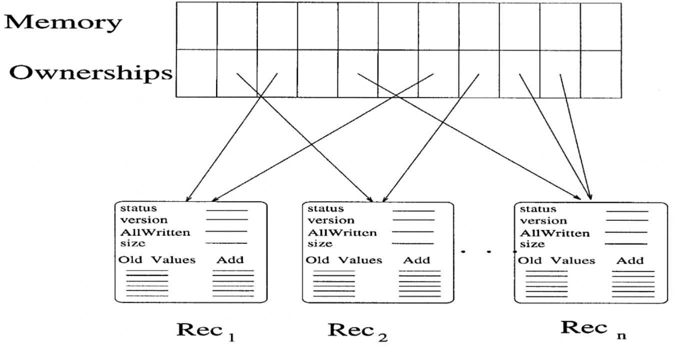
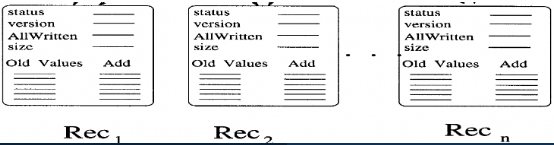


Fig. 3. STM implementation: shared data structures

A non-blocking implementation of STM

- Each process i keeps in the shared memory a record, pointed to by Rec_i, that will be used to store information on the current transaction it initiated.
- **Size** which contains the size of the data set.
- **Add[]** a vector which contains the data set addresses in increasing order
- **OldValues[]** a vector of the data set's size whose cells are initialized to Null at the beginning of every transaction. In case of a successful transaction, this vector will contain the former values stored in the involved locations.
- **Version** is an integer, initially 0, which determines the instance number of the transaction. This field is incremented every time the process terminates a transaction.



A non-blocking implementation of STM

A process *i* initiates the execution of a transaction by calling the **StartTransaction** routine.

```
StartTransaction(DataSet)
   Initialize(Rec_i, DataSet)
   Rec_i↑.stable = True
   Transaction(Rec_i, Rec_i↑.version, True)
   Rec_i↑.stable = False
   Rec_i↑.version + +
   If Rec_i↑.status = Success then
      return(Success, Rec_i↑.OldValues)
   else
      return Failure

Initialize (Rec_i, DataSet)
   Rec_i↑.status = Null
   Rec_i↑.AllWritten = Null
   Rec_i↑.size = |DataSet|
   for j = 1 to |DataSet| do
      Rec_i↑.Add[j] = DataSet[j]
      Rec_i↑.OldValues[j] = Null
```

Fig. 4. Start&Transaction

```
Transaction(rec, version, IsInitiator)
   AcquireOwnerships(rec, version)
   (status, failadd) = LL(rec↑.status)
   if status = Null then
      if (version ≠ rec↑.version) then return
      SC(rec↑.status, (Success, 0))
   (status, failadd) = LL(rec↑.status)
   if status = Success then
      AgreeOldValues(rec, version)
      NewValues = CalcNewValues(rec↑.OldValues)
      UpdateMemory(rec, version, NewValues)
      ReleaseOwnerships(rec, version)
   else
      ReleaseOwnerships(rec, version)
      if IsInitiator then
         failtran = Ownerships[failadd]
         if failtran = Nobody then
            return
         else
            failversion = failtran↑.version
            if failtran↑.stable
               Transaction(failtran, failversion, False)
```

Fig. 5. Transaction

A non-blocking implementation of STM

```
AcquireOwnerships(rec, version)
  transize = rec↑.size
  for j = 1 to size do
    while true do
      location = rec↑.add[j]
      if LL(rec↑.status) ≠ Null then return
      owner = LL(Ownerships[rec↑.Add[j]])
      if rec↑.version ≠ version return
      if owner = rec then exit while loop
      if owner = Nobody then
        if SC(rec↑.status, (Null,0)) then
          if SC(Ownerships[location], rec) then exit while loop
      else
        if SC(rec↑.status, (Failure, j)) then return

ReleaseOwnerships(rec, version)
  size = rec↑.size
  for j = 1 to size do
    location = rec↑.Add[j]
    if LL(Ownerships[location]) = rec then
      if rec↑.version ≠ version then return
      SC(Ownerships[location], Nobody)
```

```
AgreeOldValues(rec, version)
  size = rec↑.size
  for j = 1 to size do
    location = rec↑.Add[j]
    if LL(rec↑.OldValues[j]) = Null then
      if rec↑.version ≠ version then return
      SC(rec↑.OldValues[j], Memory[location])

UpdateMemory(rec, version, newvalues)
  size = rec↑.size
  for j = 1 to size do
    location = rec↑.Add[j]
    oldvalue = LL(Memory[location])
    if rec↑.AllWritten then return
    if version ≠ rec↑.version then return
    if oldvalues ≠ newvalues[j] then
      SC(Memory[location], newvalues[j])
  if (not LL(rec↑.AllWritten)) then
    if version ≠ rec↑.version then return
    SC(rec↑.AllWritten, True)
```

Fig. 6. *Ownerships* and *Memory* access

Correctness Proof Outline

The implementation is atomic and serializeable

1. All the existing processes of a transaction T read the same data set vector which was stored by $T's$ initiator. Any executing process of T which read a different data set will not be able to update any of the shared data

Correctness Proof Outline

The implementation is atomic and serializeable

1. All the existing processes of a transaction T read the same data set vector which was stored by $T's$ initiator. Any executing process of T which read a different data set will not be able to update any of the shared data
2. All the executing processes of a transaction T will never acquire ownership after the status of T has been set. All the ownership owned by T will be released before the version field of $T's$ record is incremented by $T's$ initiator.

Correctness Proof Outline

The implementation is atomic and serializeable

1. All the existing processes of a transaction T read the same data set vector which was stored by $T's$ initiator. Any executing process of T which read a different data set will not be able to update any of the shared data
2. All the executing processes of a transaction T will never acquire ownership after the status of T has been set. All the ownership owned by T will be released before the version field of $T's$ record is incremented by $T's$ initiator.
3. All the executing processes of a successful transaction T will update the memory before $T's$ AllWritten field is set to True.

STM: Liveness

STM achieves liveness through the following mechanisms:

- Non-blocking ensures that the system can always make

STM: Liveness

STM achieves liveness through the following mechanisms:

- Non-blocking ensures that the system can always make
- **Deadlock Avoidance (helping):** STM avoids deadlocks by allowing transactions to be composed of other transactions, enabling them to continue making progress even if they are blocked by other transactions.

Experimental Evaluation

- Compared the performance of STM and other methods on 64 processor bus and network architectures

Experimental Evaluation

- Compared the performance of STM and other methods on 64 processor bus and network architectures
- In the simulated bus architecture, processors communicate with shared memory modules through a common bus.

Experimental Evaluation

- Compared the performance of STM and other methods on 64 processor bus and network architectures
- In the simulated bus architecture, processors communicate with shared memory modules through a common bus.

Experimental Evaluation

- Compared the performance of STM and other methods on 64 processor bus and network architectures
- In the simulated bus architecture, processors communicate with shared memory modules through a common bus.
- Each processor had a cache with 2048 lines of 6 bytes

Experimental Evaluation

- Paper summarizes the highlights of the comparison of non-blocking (pure) methods in below Table,
- Entries are the throughput ratio of $\frac{STM}{OtherMethods}$.
- As we can be seen, STM outperforms the cooperative method in all benchmarks and outperforms Herlihy's in all except for the counter benchmark.

		10 processors		60 processors	
Throughput ratio of	*STM/other*	Herlihy's method	Cooperative method	Herlihy's method	Cooperative method
Counter	Bus	0.34	1.98	0.74	8.44
	Alewife	0.30	1.92	0.45	7.6
Doubly linked queue	Bus	6.07	1.44	58.9	3.36
	Alewife	2.44	1.75	12.9	7.28
Resource Allocation	Bus	22.5	1.09	85.61	1.69
	Alewife	24.14	1.12	59.8	2.35
Priority queue	BUS	0.42	1.26	2.8	2.16
	Alewife	0.41	1.27	1.1	2.24

Table 1: Pure implementation throughput ratio: STM / other methods

Conclusion

- This paper introduces a non-blocking software version of Herlihy and Moss's transactional memory approach.

Conclusion

- This paper introduces a non-blocking software version of Herlihy and Moss's transactional memory approach.
- STM is a concurrency control mechanism that allows multiple threads to execute transactions atomically and concurrently.

Conclusion

- This paper introduces a non-blocking software version of Herlihy and Moss's transactional memory approach.
- STM is a concurrency control mechanism that allows multiple threads to execute transactions atomically and concurrently.
- Offers significant advantages over traditional locking mechanisms

Conclusion

- This paper introduces a non-blocking software version of Herlihy and Moss's transactional memory approach.
- STM is a concurrency control mechanism that allows multiple threads to execute transactions atomically and concurrently.
- Offers significant advantages over traditional locking mechanisms
- Now-days Java, C++, Rust, Python supports the software transactional memory (STM)

Thank You!