Software Transactional Memory (STM)

Nir Shavit, Dan Touitou

Presented by
Ramesh Adhikari

April 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.

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.
- STM works by defining an atomic (**transnational**)region of code, which groups together a set of memory accesses that should be executed atomically.

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

Motivation

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

Motivation

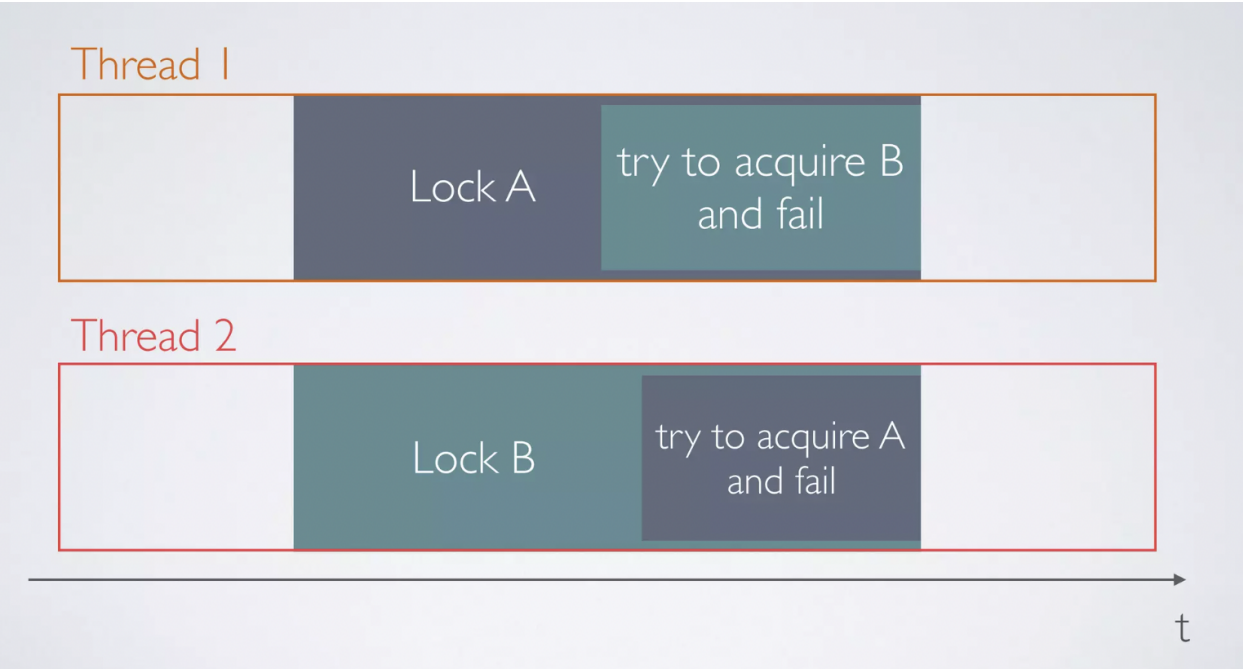
- Concurrent programming is essential to improve performance on a multi-core.
- Locks and condition variables are fundamentally flawed and difficult to manage such as Deadlock

Motivation

- Concurrent programming is essential to improve performance on a multi-core.
- Locks and condition variables are fundamentally flawed and difficult to manage such as Deadlock
- Deadlock is a situation where two or more processes are waiting for each other

Motivation

- Concurrent programming is essential to improve performance on a multi-core.
- Locks and condition variables are fundamentally flawed and difficult to manage such as Deadlock
- Deadlock is a situation where two or more processes are waiting for each other



Transactional Programming

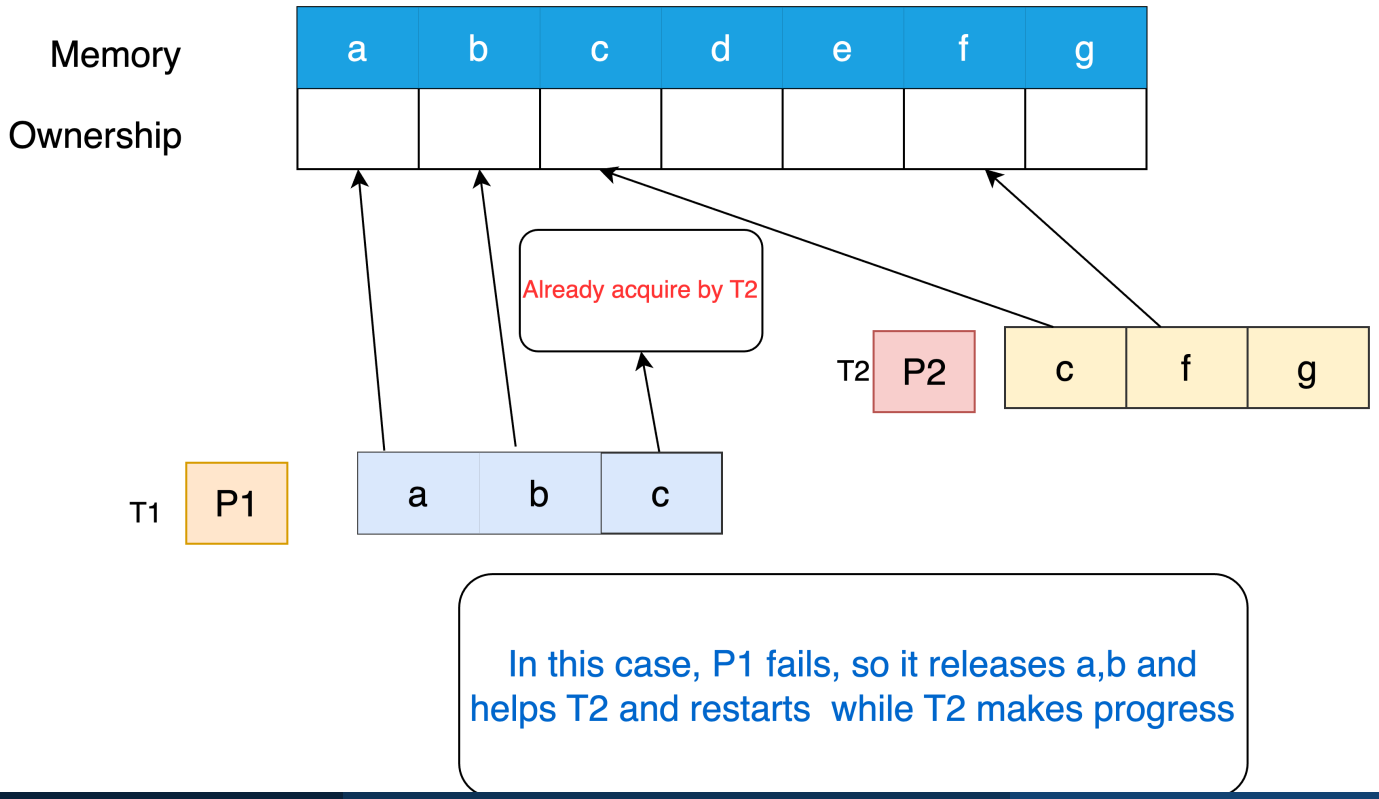
Proposed solution **Transactional memory** using the concept of transaction

```
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);
  }
}
```

Software Transactional Memory example

- Paper is about **static transactions** i.e. Transactions access a pre-determined sequence of locations.
- Transaction is executed by **acquiring the ownerships** needed in some increasing order
- If two transactions attempt to modify the same memory location concurrently, one of them will be aborted and retried later.



Instruction discussed on Paper

1. Compare-and-swap (CAS) Instruction
2. Load_Linked(LL)/Store_Conditional (SC) instruction

Compare-and-swap (CAP) instruction

1. Compare-and-swap (CAS) is an **atomic instruction** used in multithreading to achieve synchronization

Compare-and-swap (CAP) instruction

1. Compare-and-swap (CAS) is an **atomic instruction** used in multithreading to achieve synchronization
2. Special hardware instructions are used to achieve atomic operation; For example, in x86 the **LOCK CMPXCHG**

Compare and Swap (CAP) instruction

1. Writes **new value** if the **current value** is equal to the **expected/old value**.

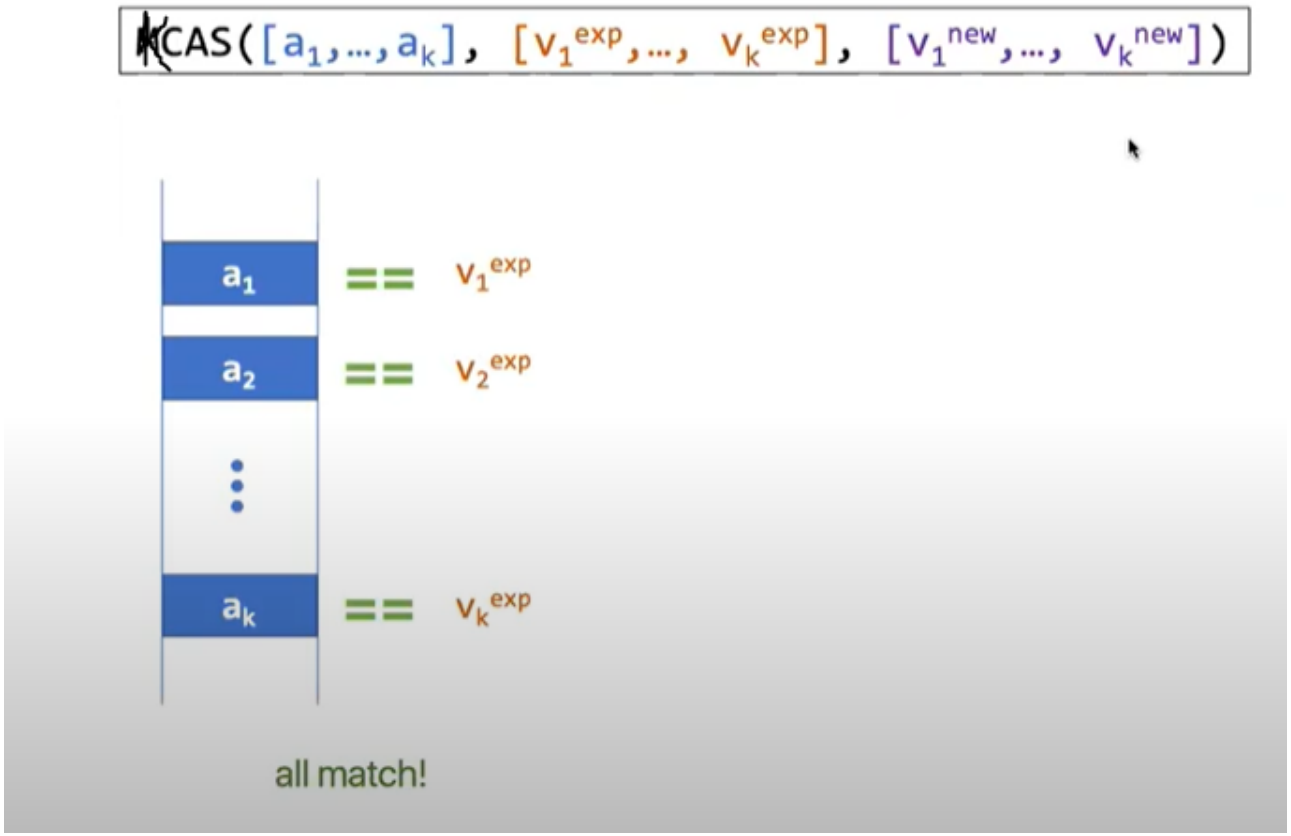
```
k_word_C&S(Size, DataSet[], Old[], New[])
   BeginTransaction
      for i = 1 to Size do
            if Read-transactional(Dataset[i]]) ≠ Old[i]
               ReturnedValue = C&S-Failure
            ExitTransaction
      for i = 1 to Size do
         Write-transactional(DataSet[i], New[i])
    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

Fig. 2. A static transaction

2.

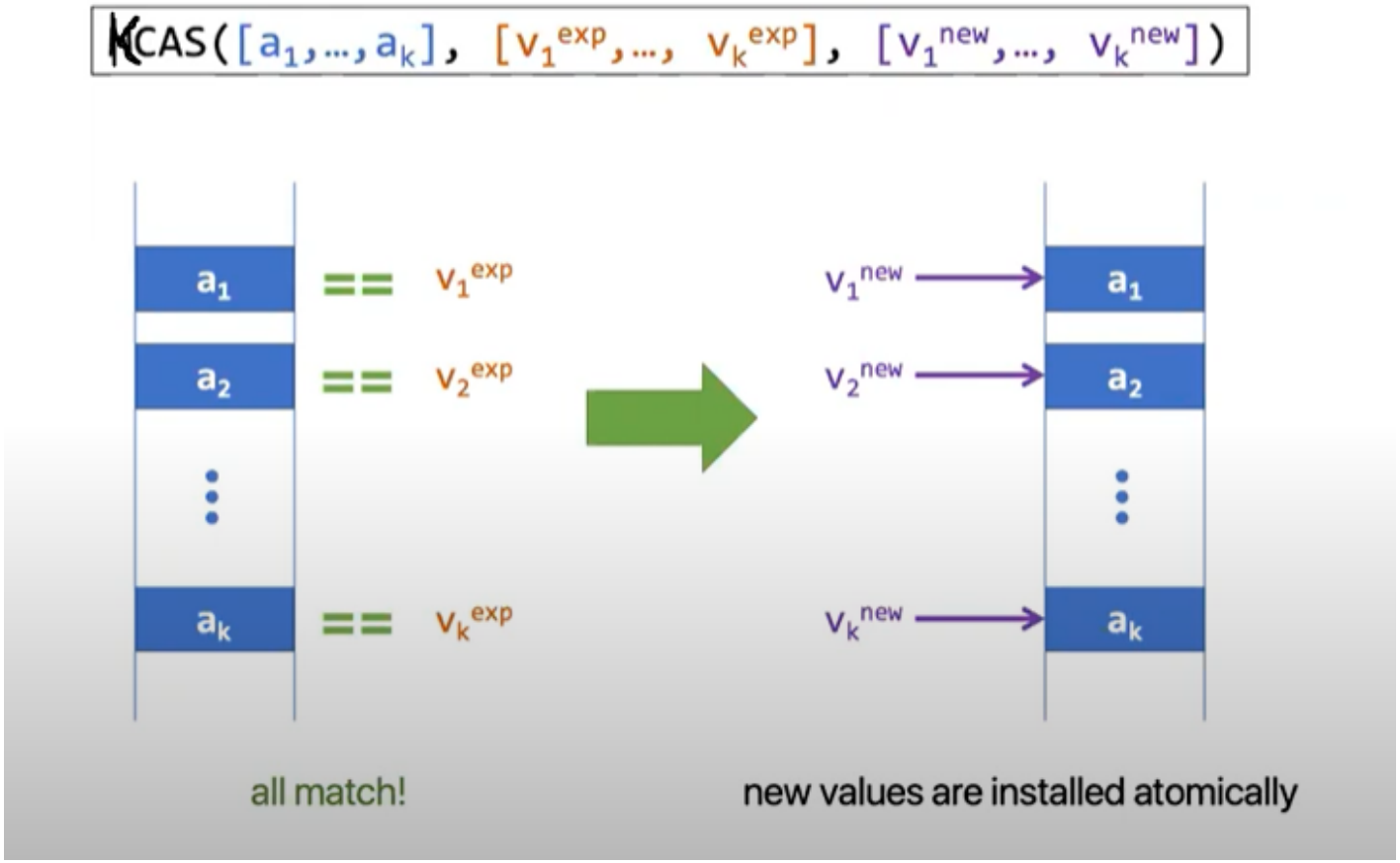
K-word Compare And Swap

- It takes a list of **memory addresses**, a list of **old/expected values** and a list of **new values**
- It compares the value of the memory address to the expected value



K-word Compare And Swap

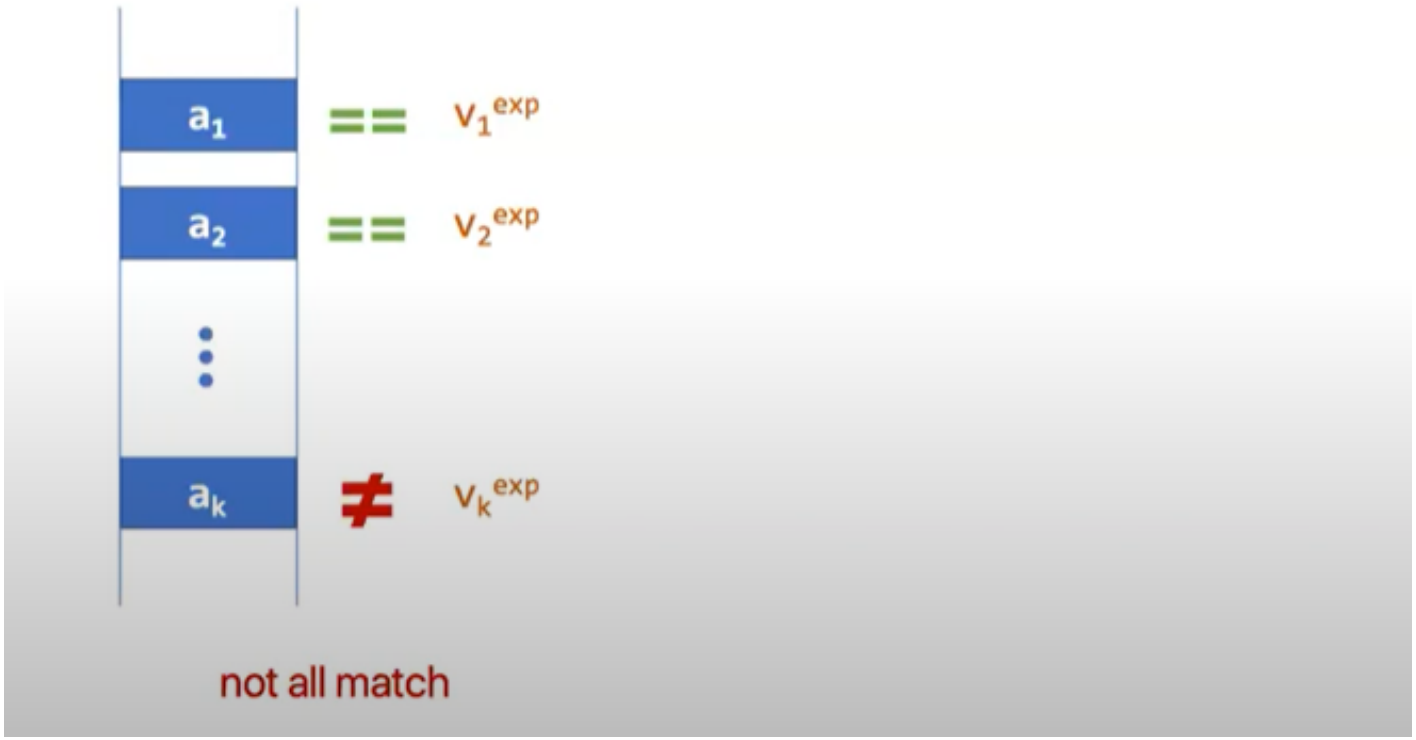
- If all match, C&S instruction atomically add new values to the respected location



K-word Compare And Swap

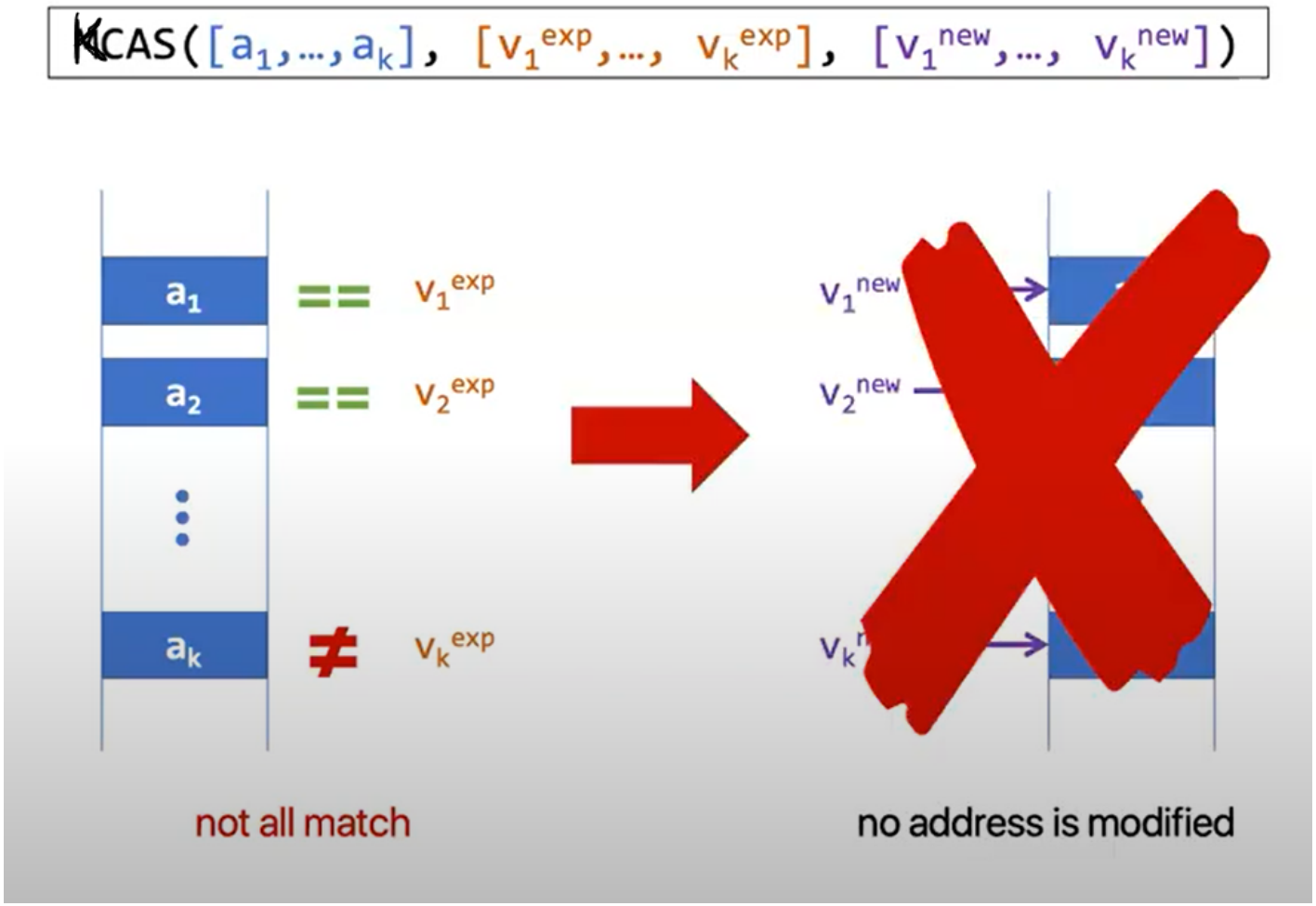
- If one or more address value does not match with expected value

$$KCAS([a_1,\ldots,a_k],\ [v_1^{exp},\ldots,\ v_k^{exp}],\ [v_1^{new},\ldots,\ v_k^{new}])$$



K-word Compare And Swap

- If one or more address value does not match
- No address will be modified



Load_linked/Sored_Conditional Instruction

Pair of instructions for atomic Read Modify Write (RMW) on shared memory

- **Load_Linked^i(l)**: Process i reads location l and returns its value v. Marks location l as "read by i".

Load_linked/Sored_Conditional Instruction

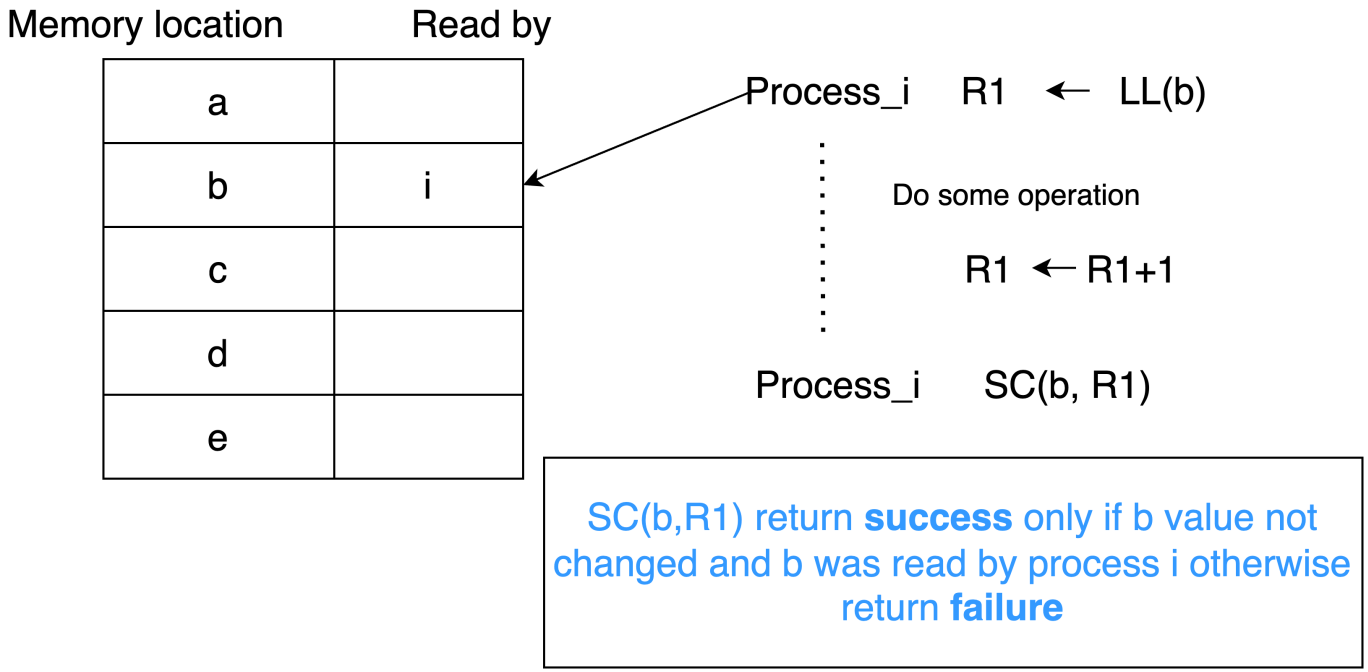
Pair of instructions for atomic Read Modify Write (RMW) on shared memory

- **Load_Linked$^i(l)$**: Process i reads location l and returns its value v. Marks location l as "read by i".
- **Store_Conditional$^i(l, v)$**: if location l is marked as "read by i," the process i writes the value v to l, and returns a success status. Otherwise returns a failure status.

Load_linked/Sored_Conditional Instruction

Pair of instructions for atomic Read Modify Write (RMW) on shared memory

- **Load_Linked$^i(l)$**: Process i reads location l and returns its value v. Marks location l as "read by i".
- **Store_Conditional$^i(l, v)$**: if location l is marked as "read by i," the process i writes the value v to l, and returns a success status. Otherwise returns a failure status.



Related Work

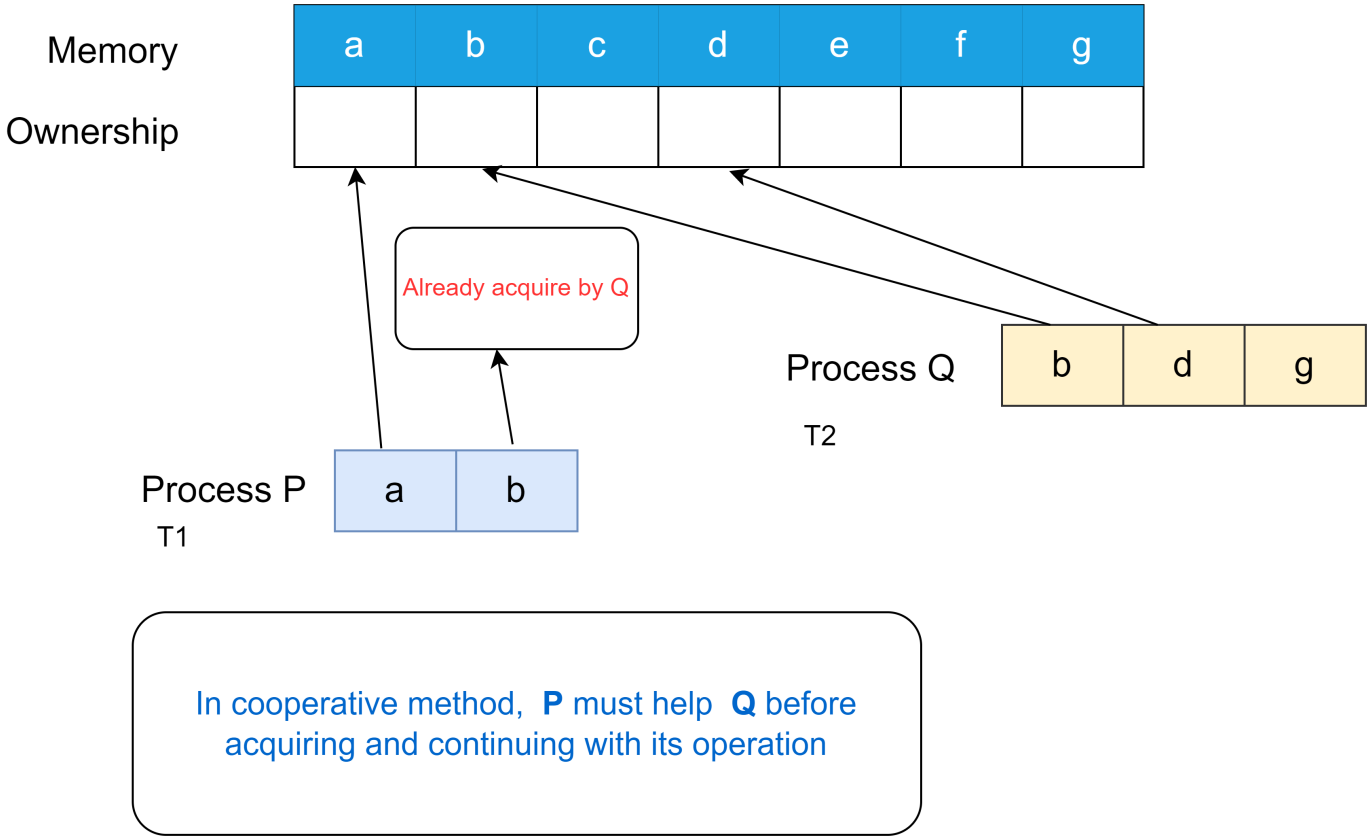
- **Hardware method:** Herlihy and Moss have proposed an ingenious hardware solution: transactional memory. By **adding a specialized associative cache** and making several minor changes to the cache consistency protocols.

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.

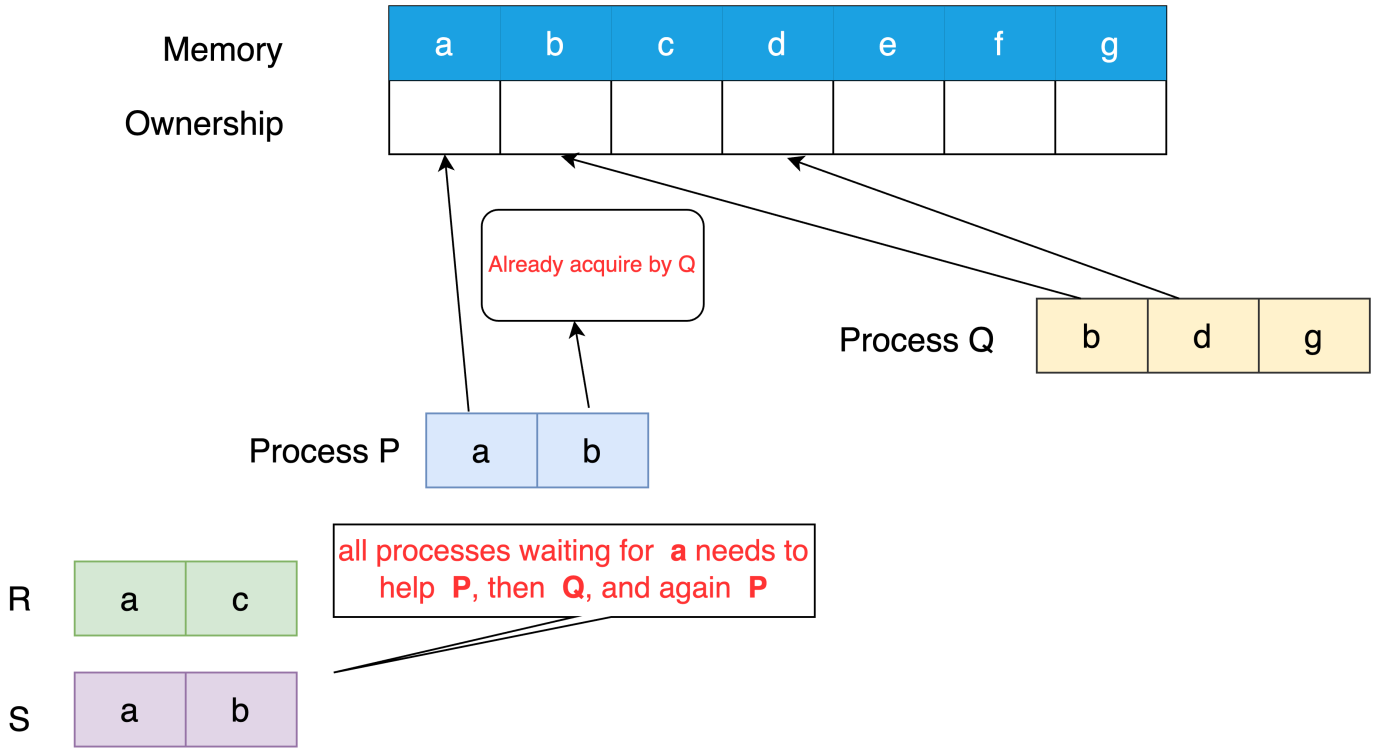
Issue on recursive Cooperative method

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



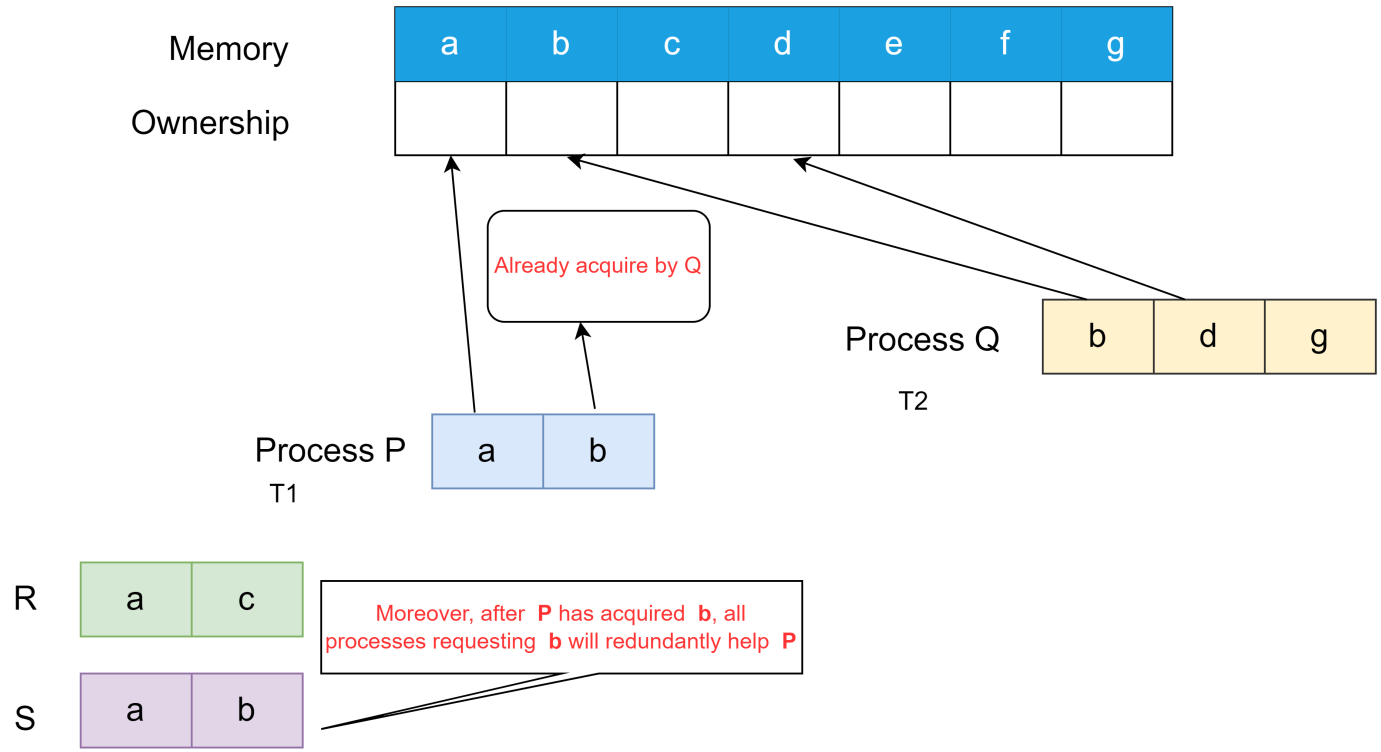
Issue on recursive Cooperative method

All processes waiting for **a** to help **P**, then **Q**, and again **P**.



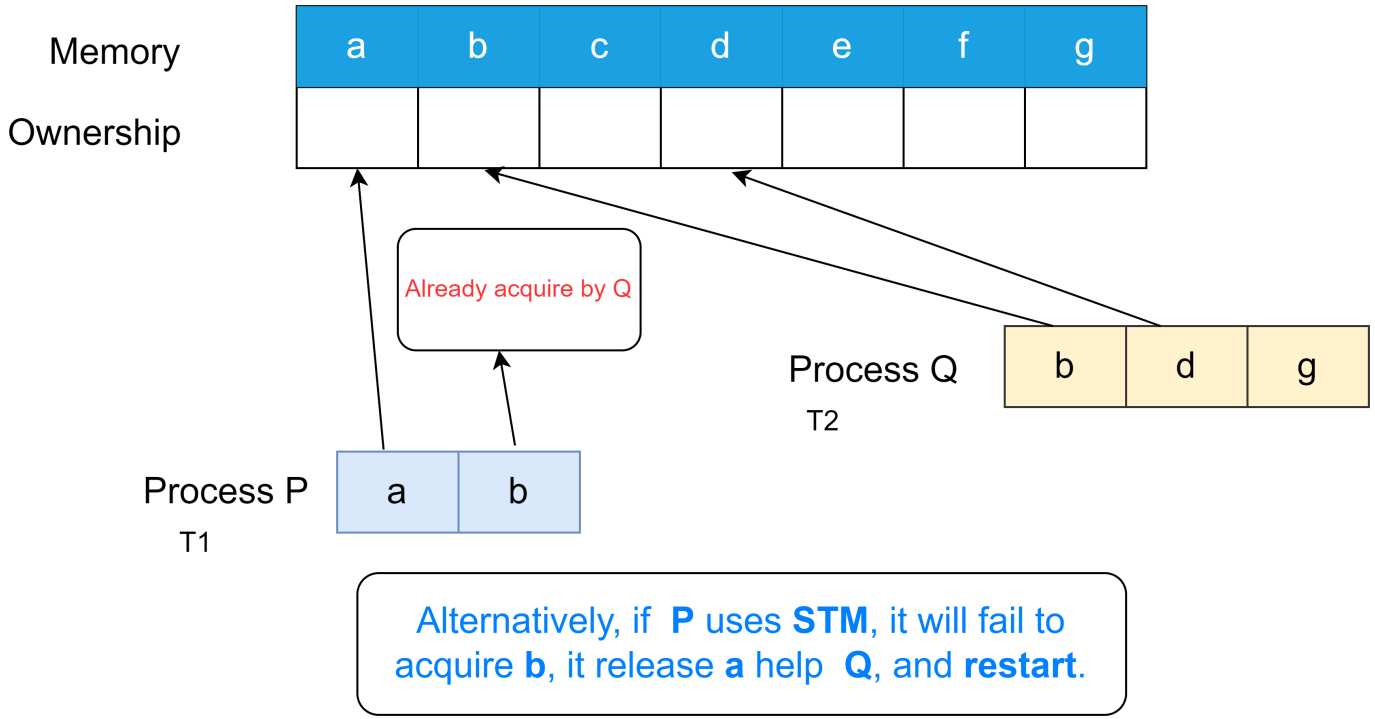
Issue on recursive Cooperative method

Moreover, after **P** has acquired **b**, all processes requesting **b** will redundantly help **P**.



Cooperative method to STM

- **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**

A non-blocking implementation of STM

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

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

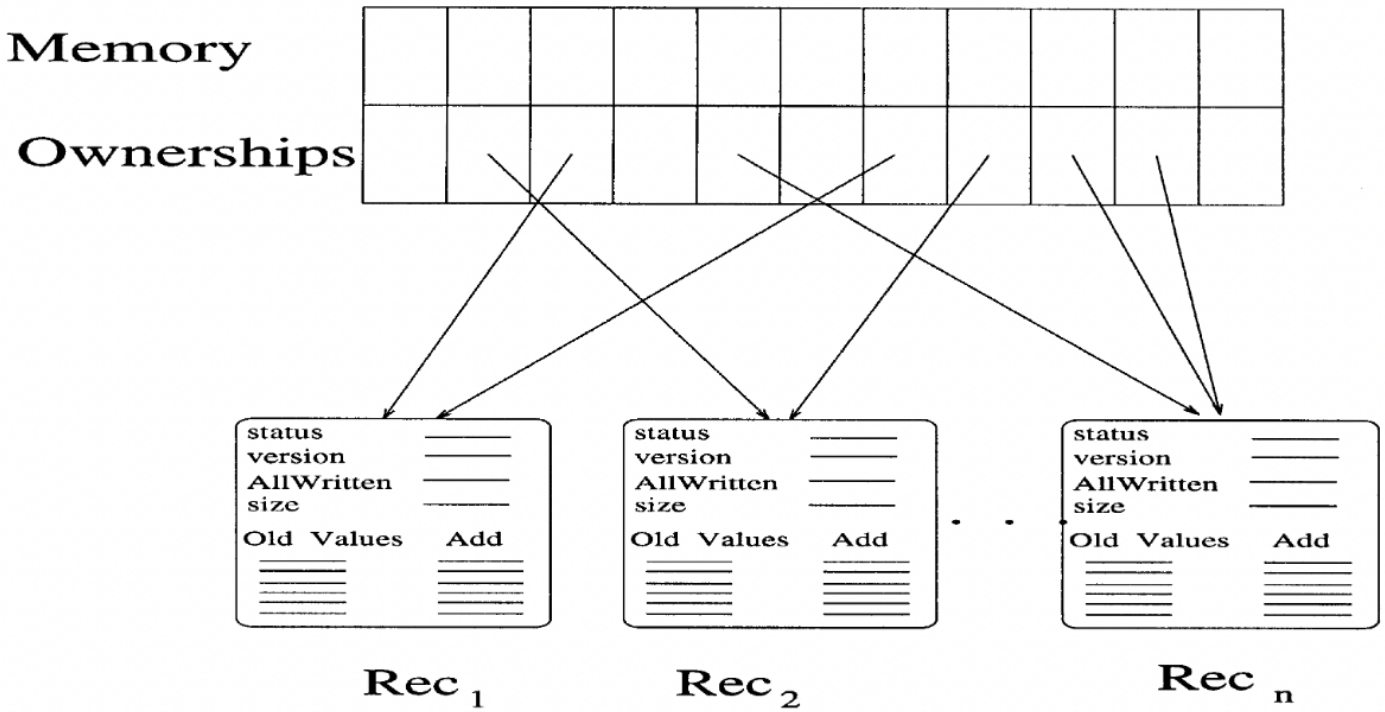
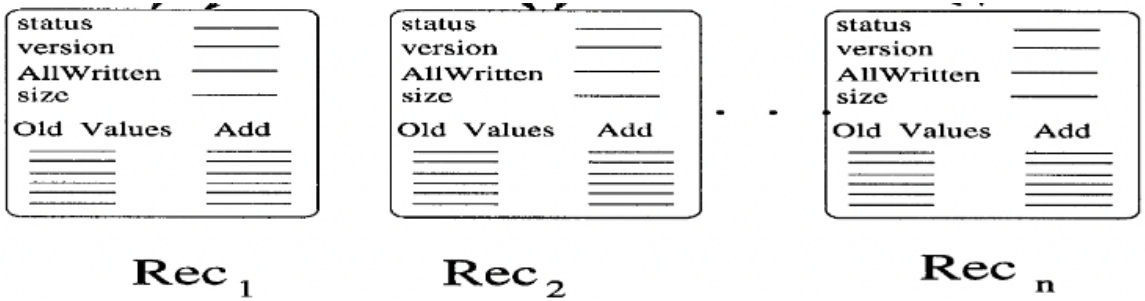


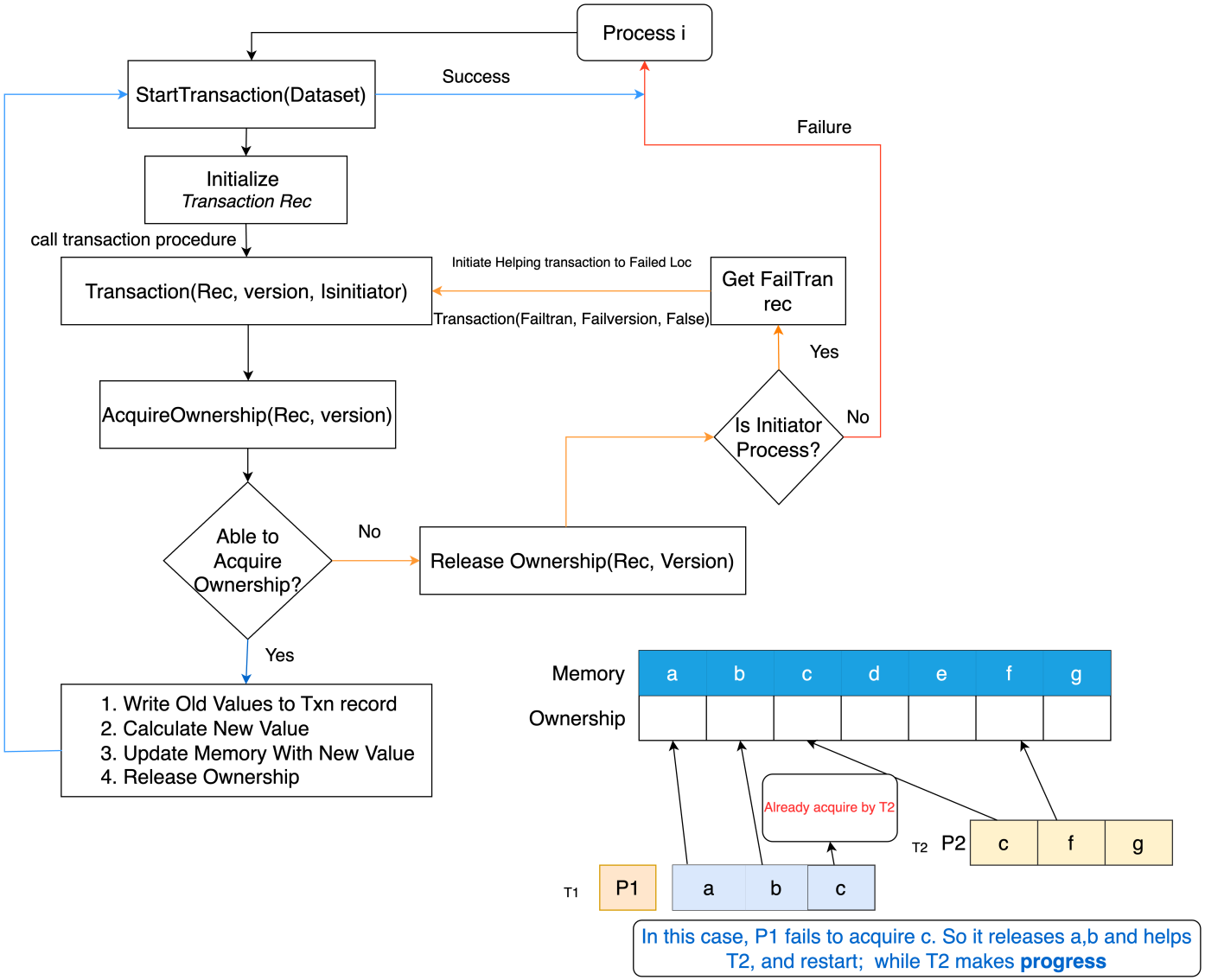
Fig. 3. STM implementation: shared data structures

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. This field is incremented every time the process terminates a transaction.



Flow of a Transaction



Implementation- Initialization

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

The Location of Memory it needs to own

```
StartTransaction(DataSet)
    Initialize(Reci, 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
```

Gloabl Memory

Memory	a	b		
Ownership				

Thread Local memory

Rec_i	
Version	0
Stable	True
Status	NULL
AllWritten	NULL
Add[]	[a,b]
OldValues[]	[NULL,NULL]

Implementation- Transaction Procedure

The procedure Transaction, gets as parameters Rec, the address of the record the transaction executed, and a boolean value IsInitiator.

rec- The thread that execute this transaction
version-serial number of transaction

```
Transaction(rec, version, IsInitiator)
    AcquireOwnerships(rec, version) //Try to own locations

    (status, failadd)=LL(rec.status)
    if status==NULL then // Success in acquireOwnerships
        if (version!=rec.version) then return
        SC(rec.status, (Success, 0))
    (status, failadd)=LL(rec.status)
    if status==Success then //execute Transaction
        AgreeOldValues(rec, version)
        NewValues=CalcNewValues(rec.OldValues)
        UpdateMemory(rec, version, NewValues)
        ReleaseOwnerships(rec, version)
    else //Failed in acquireOwnerships
    ReleaseOwnerships(rec, version)
    if IsInitiator then
        failtran=Ownerships[failadd]
        if failtran=Nobody then
            return
        else  //execute the transaction that owns the location you want
            failversion=failtran.version
            if failtran.stable
                Transaction(failtran, failversion, FALSE)
```

Another thread own the location I need and it has not finish its transaction yet
So I go out and execute its transaction in order to help it

Gloabl Memory

Memory	a	b		
Ownership				

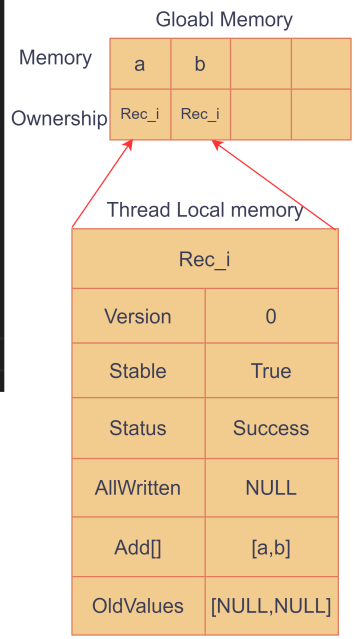
Thread Local memory

Rec_i	
Version	0
Stable	True
Status	NULL
AllWritten	NULL
Add[]	[a,b]
OldValues	[NULL,NULL]

Implementation- AcqureOwnership

- AcquireOwnerships of may be called either by the initiator or by the helping processes, we must ensure that
  - From the moment that the status of the transaction becomes fixed, no additional ownerships are allowed for that transaction.

```
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 //location is already mine
            if owner==Nobody then // acquire location
                if SC(rec.status, (Null,0)) then
                    if SC(Ownerships[location], rec) then exit while loop
            else //location is taken by someone else
                if SC(rec.status, (Failure, j)) then return
```

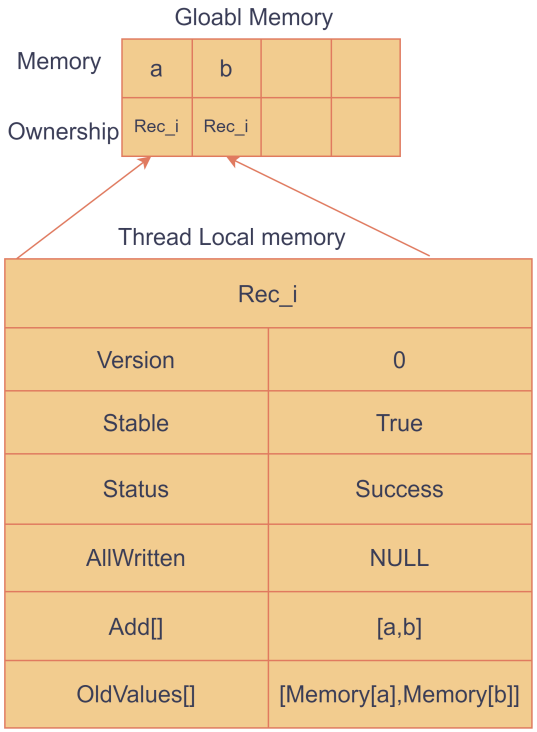


Implementation- AgreeOldValue

Writes the old values into the transaction's record

```
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])
```

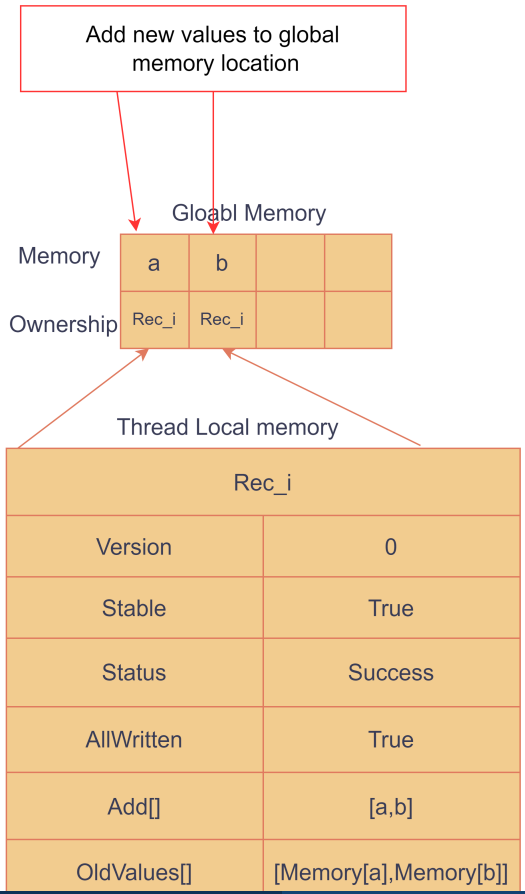
Write old value of memory location to transaction record



Implementation- UpdateMemory

- When writing the new values the processes synchronize in order to prevent a slow process from updating the memory after the ownerships have been released.
- To do so every process sets the AllWritten field to be True, after updating the memory and before releasing the ownerships.

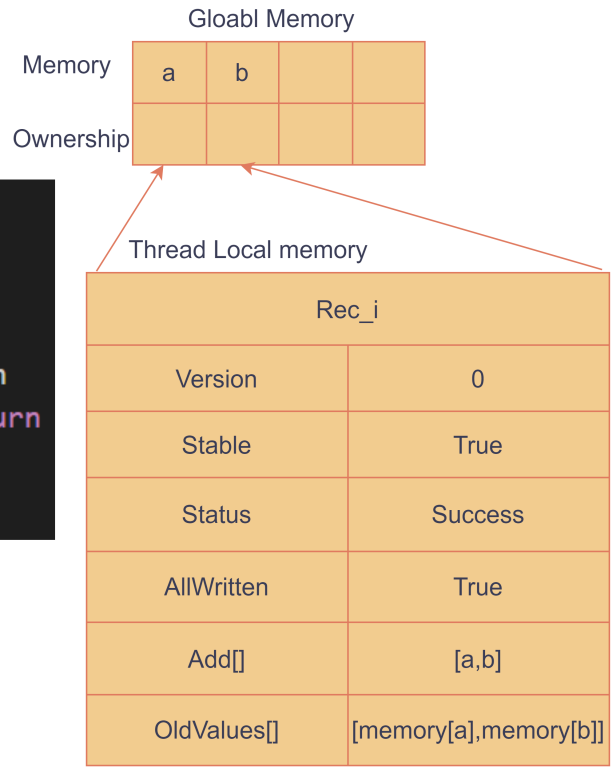
```
UpdateMemory(rec, version, newvalues)
    size=rec.size
    for j=1 to size do
        location=rec.Add[j]
        oldvalue=LL(Memory[location])
        if rec.AllWritten then return //work is done
        if version!=rec.version then return
        if oldvalues!=newvalues[j] then
            SC(Memory[location], newvalues[j])

    if (not LL(recC.AllWritten)) then
        if version!=rec.version then return
        SC(rec.AllWritten, TRUE)
```



Implementation- ReleaseOwnership

Releases the ownership by storing Nobody in the ownership field

```
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)
```



Implementation- Complete

After executing the transaction the process checks if the transaction has succeeded, and if so returns the content of the vector OldValues otherwise return Failure.

```
StartTransaction(DataSet)
    Initialize(Reci, 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
```

Gloabl Memory

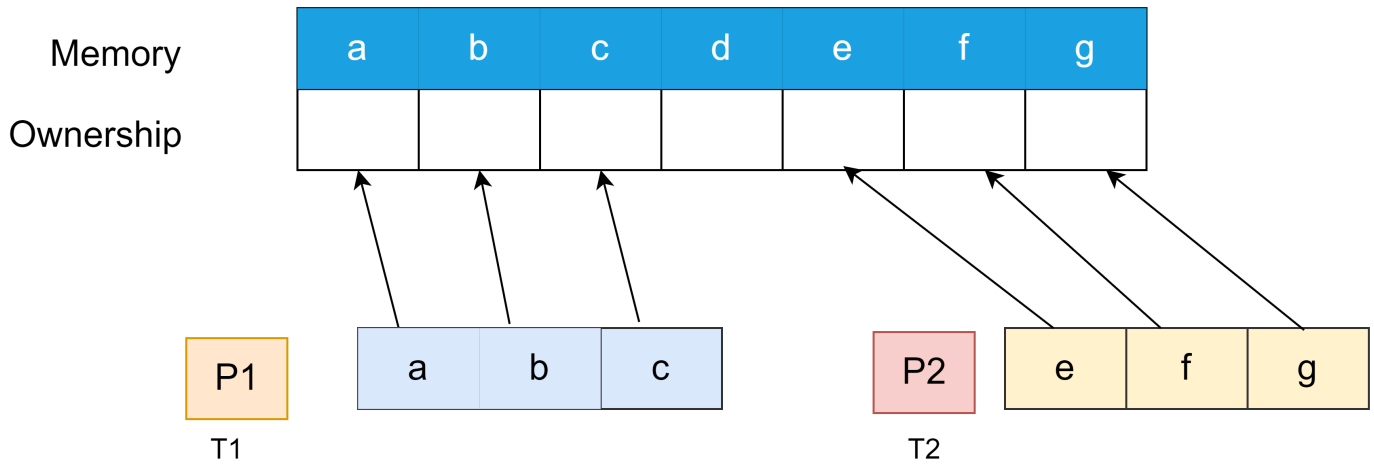
Memory	a	b		
Ownership				

Thread Local memory

Rec_i	
Version	1
Stable	False
Status	Success
AllWritten	True
Add[]	[a,b]
OldValues[]	[Memory[a],Memory[b]]

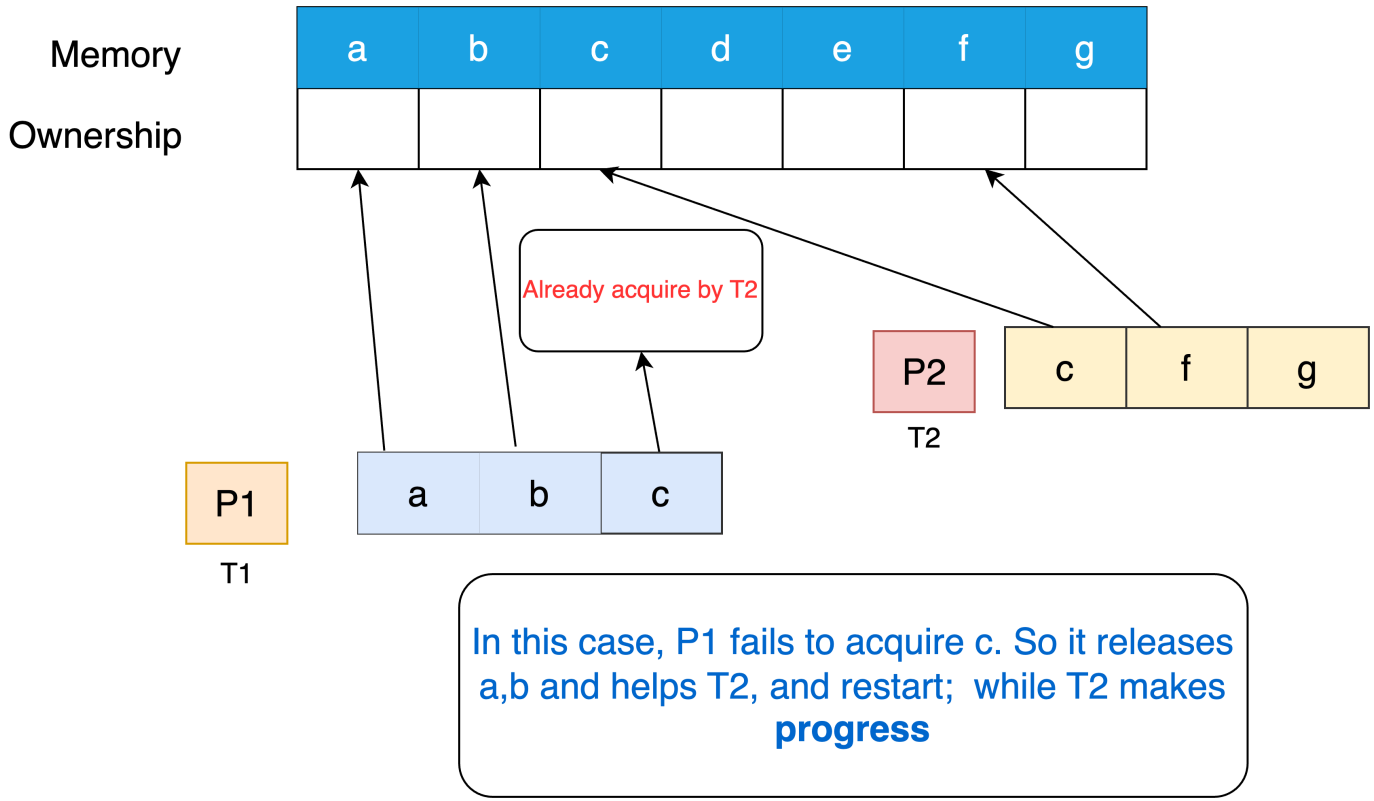
STM: Liveness (Non-blocking)

Non-blocking ensures that the system can always make progress



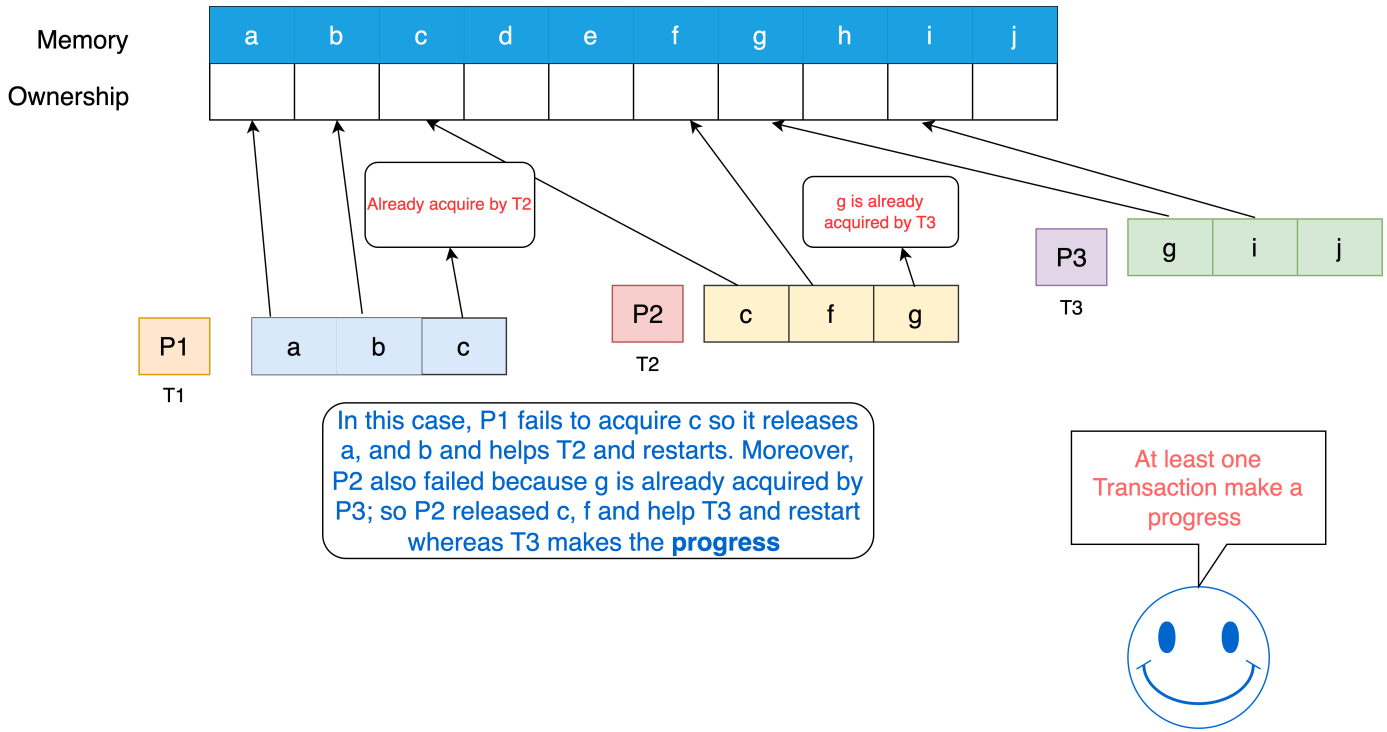
STM: Liveness (Non-blocking)

Non-blocking ensures that the system can always make progress



STM: Liveness (Non-blocking)

Non-blocking ensures that the system can always make progress



STM: Liveness (Non-blocking)

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

Implementation is atomic

1. Atomic of operations is ensured by acquiring exclusive ownership of memory locations accessed by an operation.
2. If a transaction cannot acquire ownership, it fails and releases any previously acquired ownership and restarts.
3. If a transaction can acquire ownership, it succeeds in executing the operation and frees any acquired ownership.

Experimental Evaluation

- Paper summarizes the highlights of the comparison of non-blocking 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

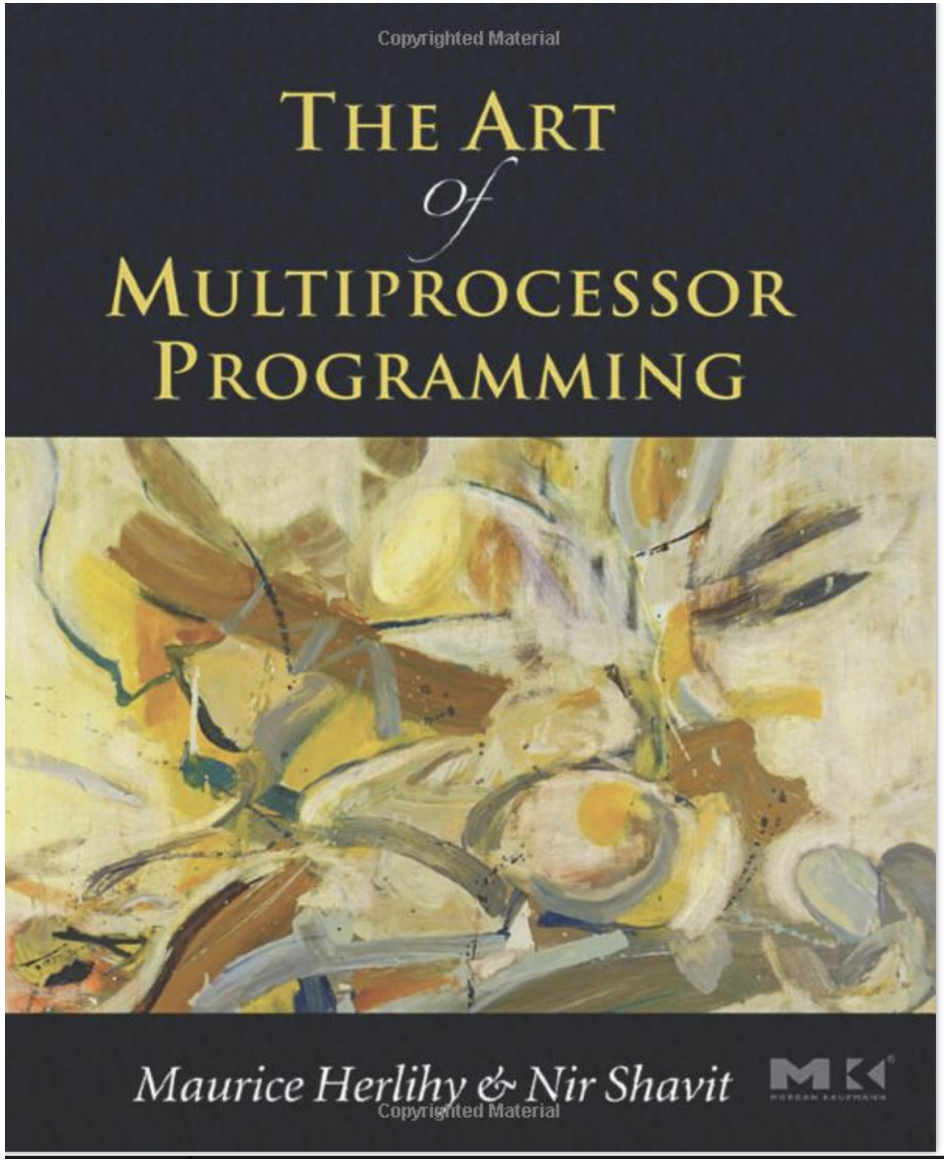
Limitation of STM

- Static - Information about the transaction is required before the start of the transaction
  - Size
  - Dataset

Conclusion

- STM is a concurrency control mechanism that allows multiple threads to execute transactions atomically and concurrently.
- Nowadays Java, C++, Rust, Python supports the software transactional memory (STM)

Thank You!


Copyrighted Material
THE ART
of
MULTIPROCESSOR
PROGRAMMING
Maurice Herlihy & Nir Shavit
Copyrighted Material