

Wait-free data structures and wait-free transactions

Pedro Ramalhete

<http://www.concurrencyfreaks.com>

pramalhe@gmail.com

Who am I

- My name is Pedro Ramalhete
- I've been working on concurrent synchronization algorithms and data structures for the past 10 years
- I write scalable software for a large tech company
- Occasionally, I write some blog posts at <http://www.concurrencyfreaks.com>
- A lot of what I'm going to talk about today has been made in collaboration with Andreia Correia and Pascal Felber



Disclaimer: The opinions expressed in during this talk are my own and unrelated to the company I work or the people I collaborate with. Any errors or omissions are my own.

Table of Contents

- Definitions (progress, consistency, ACID, DAP)
- Concurrent data structures, how hard can it be?
- Lock-Free Software Transactional Memory
- Wait-Free Software Transactional Memory
- Persistent Memory
- Now and the Future

Definitions

Progress Condition

Blocking: an unexpected delay by one thread can prevent others from making progress

Non-Blocking:

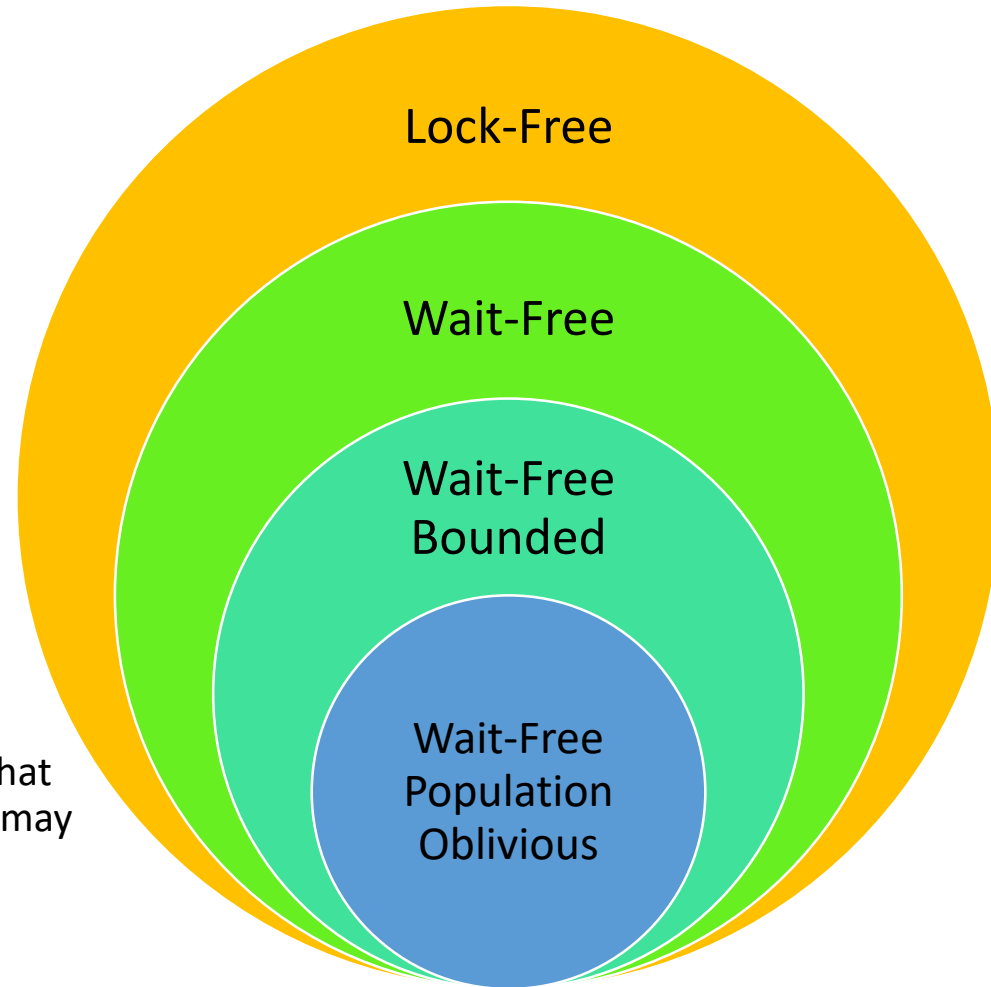
Obstruction-Free: A method is Obstruction-Free if, from any point after which it executes in isolation, it finishes in a finite number of steps.

Lock-Free: A method is Lock-Free if it guarantees that infinitely often some thread calling this method finishes in a finite number of steps.

Wait-Free: A method is Wait-Free if it guarantees that every call finishes its execution in a finite number of steps.

Wait-Free Bounded: A method is Wait-Free Bounded if it guarantees that every call finishes its execution in a finite and *bounded* number of steps. This bound may depend on the number of threads.

Wait-Free Population Oblivious: A Wait-Free method whose performance does not depend on the number of active threads is called Wait-Free Population Oblivious.

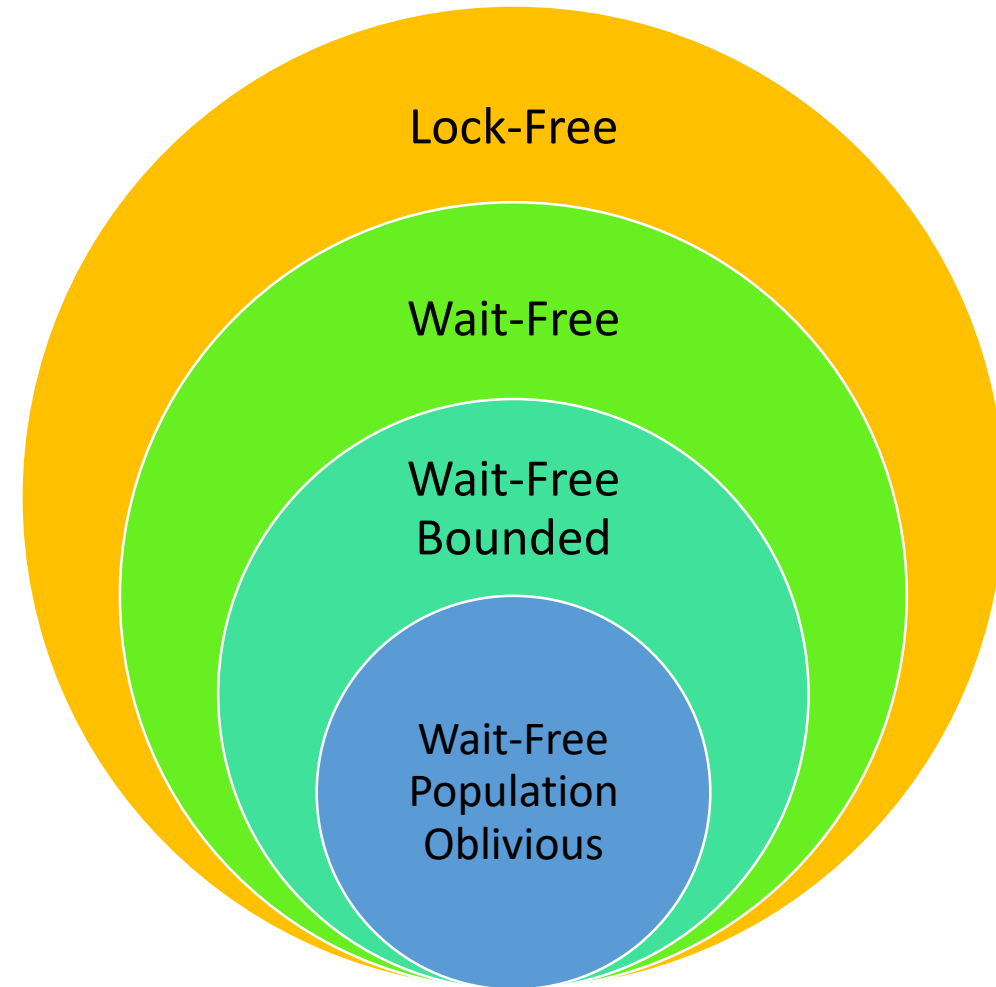


A data structure is lock-free if...

A data structure is said to be lock-free if **all** its methods are lock-free (or wait-free)

A data structure is said to be wait-free if **all** its methods are wait-free

... and lock-free memory reclamation?



Different properties for different progress

	Resilient to Failures	Deterministic Tail Latency (cut-off)
Blocking Locks, reader-writer locks, java.util.concurrent.ConcurrentHashMap	no	no (starvation-free is an exception)
Lock-Free Michael-Scott queue, Maged-Harris Linked list set Java.util.concurrent.ConcurrentLinkedQueue	yes	no
Wait-Free Kogan-Petrunk queue, Sim queue, Turn queue	yes	yes

Throughput and scalability are not a factor here...

Progress and failures

Blocking

If a thread dies (while holding the lock) the other threads can no longer execute work



Progress and failures

Lock-Free

If a thread dies, the other threads can still execute work



Consistency Models

(in the context of concurrent data structures)

Strict: Absolute time ordering for all shared access

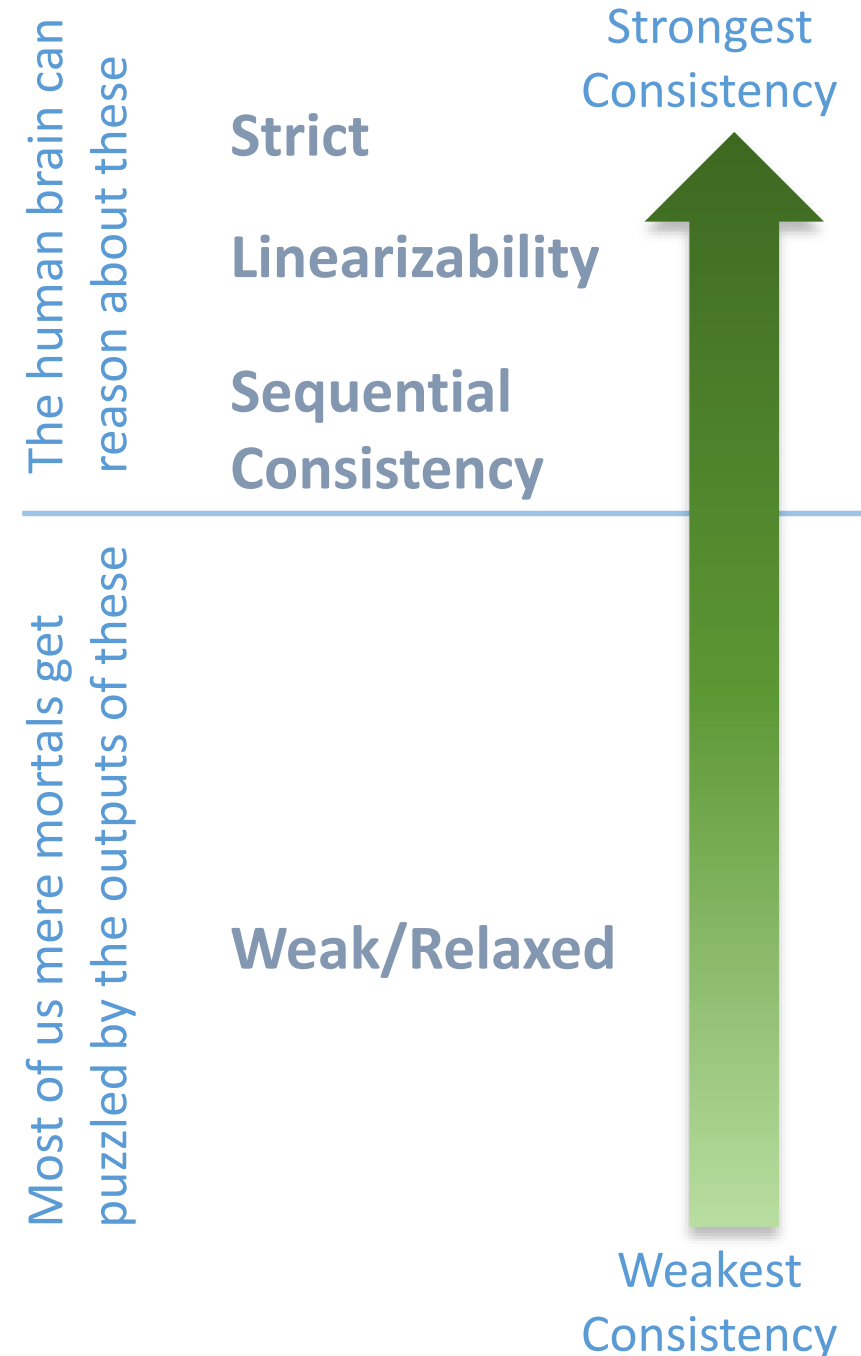
Linearizability: There is a point in time where the operation appears to the rest of the system to occur instantaneously

Sequential Consistency: Result of an execution appears as if:

- All operations executed in some sequential order
- Memory operations of each process in program order

Relaxed: anything goes? You know what most developers call data structures that have this consistency?

Buggy!



Definitions

What is DAP?

Roughly speaking, **DAP** (Disjoint Access Parallel) is a technical term used to describe *operations which modify different pieces of data*.

Read-only operations are, by definition, DAP.

To be DAP or not to be DAP is a **property of the entire workload** and not of individual operations.

In the context of data structures, some data structures are intrinsically non-DAP.

Examples of non-DAP data structures are stacks and queues.

Example of a DAP data structure is a fixed-size hashtable.

Non-DAP example Stack



In a concurrent *stack*, all threads executing an operation will modify the same variable, head. A stack is **not DAP**.

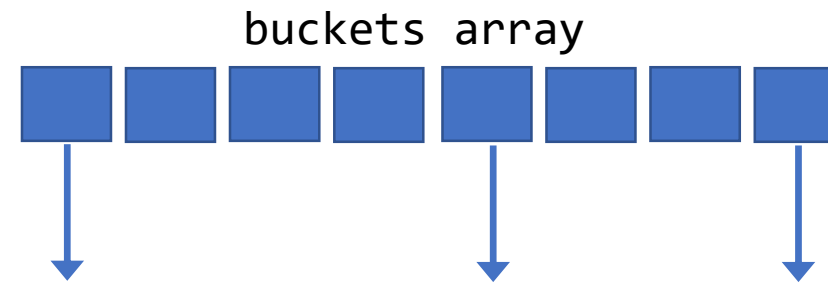


Disjoint Access Parallel (DAP) example

Fixed-size hashmap



In a concurrent *hashmap*, statistically, threads will modify *different* nodes and variables of the data structure.



Other terminology

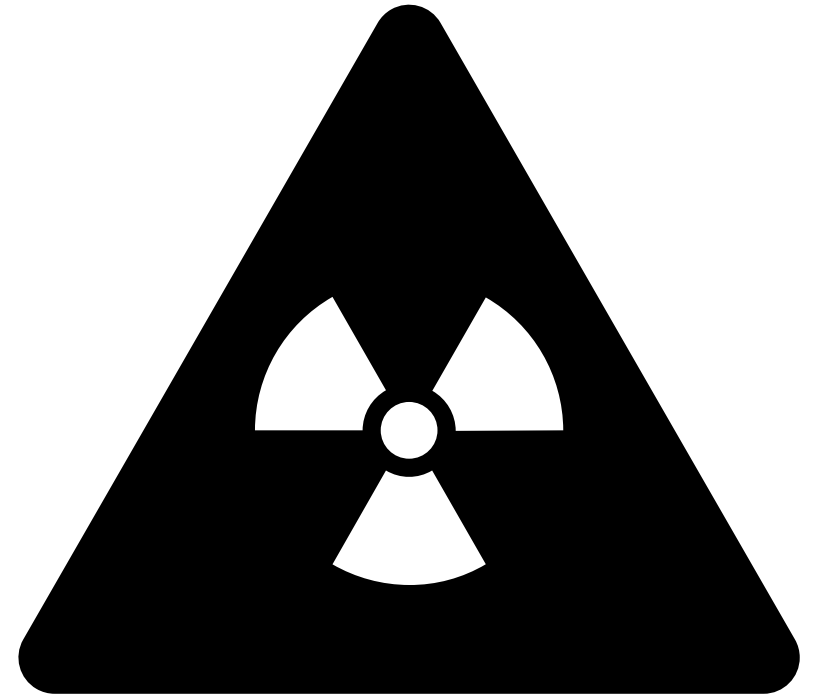
Sequential data structure: A data structure written for single-threaded applications. Its methods have no concurrency mechanism.

CAS: Compare-And-Swap. Modifies a single word atomically if and only if the contents match the expected value. Named `atomic_compare_exchange_strong()` in C/C++ and `compareAndSet()` in Java

DCAS: Double-word Compare-And-Swap. Modifies two adjacent words atomically, if and only if contents match the expected values. **CMPXCHG16B** in x86.

How hard can they be?

Lock-Free data structures



Concurrent data structures

How hard can it be?

Easy

Protecting a sequential data structure with a global lock (blocking)



Simple enough for a novice developer to do correctly

Protecting a data structure with many locks (blocking)



Requires a senior software with a good deal of expertise in concurrency

years of hard work to gain the expertise needed to cross this gap

Hard

Design a lock-free data structure

Design a wait-free data structure



Expert-only domain. It takes someone like: Maurice Herlihy, Michael Scott, Panagiota Fatourou, Erez Petrank, etc

How about something simple... like a queue?

Sequential implementation + global lock

Blocking

Easy to modify (correctly)

Only ~12 lines of code

```
bool enqueue(T* item) {  
    std::lock_guard<std::mutex> lock(g_mutex);  
    if (item == nullptr) return false;  
    Node* newNode = new Node(item);  
    tail->next = newNode;  
    tail = newNode;  
    return true;  
}
```

```
T* dequeue() {  
    std::lock_guard<std::mutex> lock(g_mutex);  
    Node* lhead = head;  
    if (lhead == tail) return nullptr;  
    head = lhead->next;  
    delete lhead;  
    return head->item;  
}
```


How about something simple... like a queue?

Lock-free implementation

Lock-Free

Can not be easily modified

Still small: ~23 lines of code

```
T* dequeue() {
    Node* node = hp.protect(kHpHead, head);
    while (node != tail.load()) {
        Node* lnext = hp.protect(kHpNext, node->next);
        if (casHead(node, lnext)) {
            T* item = lnext->item;
            hp.clear();
            hp.retire(node);
            return item;
        }
        node = hp.protect(kHpHead, head);
    }
    hp.clear();
    return nullptr;
}
```

```
bool enqueue(T* item) {
    if (item == nullptr) return false;
    Node* newNode = new Node(item);
    while (true) {
        Node* ltail = hp.protectPtr(kHpTail, tail);
        if (ltail == tail.load()) {
            Node* lnext = ltail->next.load();
            if (lnext == nullptr) {
                if (ltail->casNext(nullptr, newNode)) {
                    casTail(ltail, newNode);
                    hp.clear();
                    return;
                }
            } else {
                casTail(ltail, lnext);
            }
        }
    }
}
```

How about something simple... like a queue?

Wait-free implementation

Wait-free

Can not be easily modified

Large: ~80 lines of code

```
pointer_t newDeqIndex;
newDeqIndex.u.seq = lpointer.u.seq + 1;
newDeqIndex.u.index = myIndex;
myState->head.store(newHead, std::memory_order_relaxed);
node = lstate->head;
if (deqPointer.compare_exchange_strong(lpointer, newDeqIndex)) {
    while (node != newHead) {
        Node* next = node->next.load();
        hp.retire(node, tid);
        node = next;
    }
    break;
}
}
hp.clear(tid);
return deqReused[deqPointer.load().u.index].items[tid].load();
}
```

```
void enqueue(T* item) {
    if (item == nullptr) return;
    // Publish enqueue request
    items[tid].store(item, std::memory_order_relaxed);
    bool newrequest = !enqueueers[tid].load(std::memory_order_relaxed);
    enqueueers[tid].store(newrequest);
    for (int iter = 0; iter < 2; iter++) {
        pointer_t lpointer = deqPointer.load();
        DeqState* lstate = &deqReused[lpointer.u.index];
        // Check if my request has been done
        if (lstate->applied[tid].load() == newrequest) {
            if (lpointer.raw == deqPointer.load().raw) break;
        }
        // Help opened dequeue requests, starting from turn+1
        Node* newHead = hp.protectPtr(kHpNode, lstate->head, tid);
        if (lpointer.raw != deqPointer.load().raw) continue;
        const int myIndex = (lpointer.u.index == 2*tid) ? 2*tid+1 : 2*tid;
        DeqState* const myState = &deqReused[myIndex];
        Node* node = newHead;
        for (int j = 0; j < maxThreads; j++) {
            // Check if it is an open request
            const bool applied = lstate->applied[j].load();
            if (enqueueers[j].load() == applied) {
                myState->items[j].store(lstate->items[j], std::memory_order_relaxed);
                myState->applied[j].store(applied, std::memory_order_relaxed);
                continue;
            }
            myState->applied[j].store(!applied, std::memory_order_relaxed);
            if (node->next.load() == nullptr) {
                myState->items[j].store(nullptr, std::memory_order_relaxed);
            } else {
                node = hp.protectPtr(kHpNode, node->next, tid);
                if (lpointer.raw != deqPointer.load().raw) break;
                myState->items[j].store(node->item, std::memory_order_relaxed);
                newHead = node;
            }
        }
        if (lpointer.raw != deqPointer.load().raw) continue;
    }
}
```

```
T* dequeue(const int tid) {
    // Publish dequeue request
    bool newrequest = !dequeuers[tid].load(std::memory_order_relaxed);
    dequeuers[tid].store(newrequest);
    for (int iter = 0; iter < 2; iter++) {
        pointer_t lpointer = deqPointer.load();
        DeqState* lstate = &deqReused[lpointer.u.index];
        // Check if my request has been done
        if (lstate->applied[tid].load() == newrequest) {
            if (lpointer.raw == deqPointer.load().raw) break;
        }
        // Help opened dequeue requests, starting from turn+1
        Node* newHead = hp.protectPtr(kHpNode, lstate->head, tid);
        if (lpointer.raw != deqPointer.load().raw) continue;
        const int myIndex = (lpointer.u.index == 2*tid) ? 2*tid+1 : 2*tid;
        DeqState* const myState = &deqReused[myIndex];
        Node* node = newHead;
        for (int j = 0; j < maxThreads; j++) {
            // Check if it is an open request
            const bool applied = lstate->applied[j].load();
            if (dequeuers[j].load() == applied) {
                myState->items[j].store(lstate->items[j], std::memory_order_relaxed);
                myState->applied[j].store(applied, std::memory_order_relaxed);
                continue;
            }
            myState->applied[j].store(!applied, std::memory_order_relaxed);
            if (node->next.load() == nullptr) {
                myState->items[j].store(nullptr, std::memory_order_relaxed);
            } else {
                node = hp.protectPtr(kHpNode, node->next, tid);
                if (lpointer.raw != deqPointer.load().raw) break;
                myState->items[j].store(node->item, std::memory_order_relaxed);
                newHead = node;
            }
        }
        if (lpointer.raw != deqPointer.load().raw) continue;
    }
}
```

Even experts have difficulties dealing with lock-free data structures

Designing and implementing takes time: Creating a new lock-free data structure takes months of work by experts.

Customizing takes time: Modifying functionality takes weeks to months.

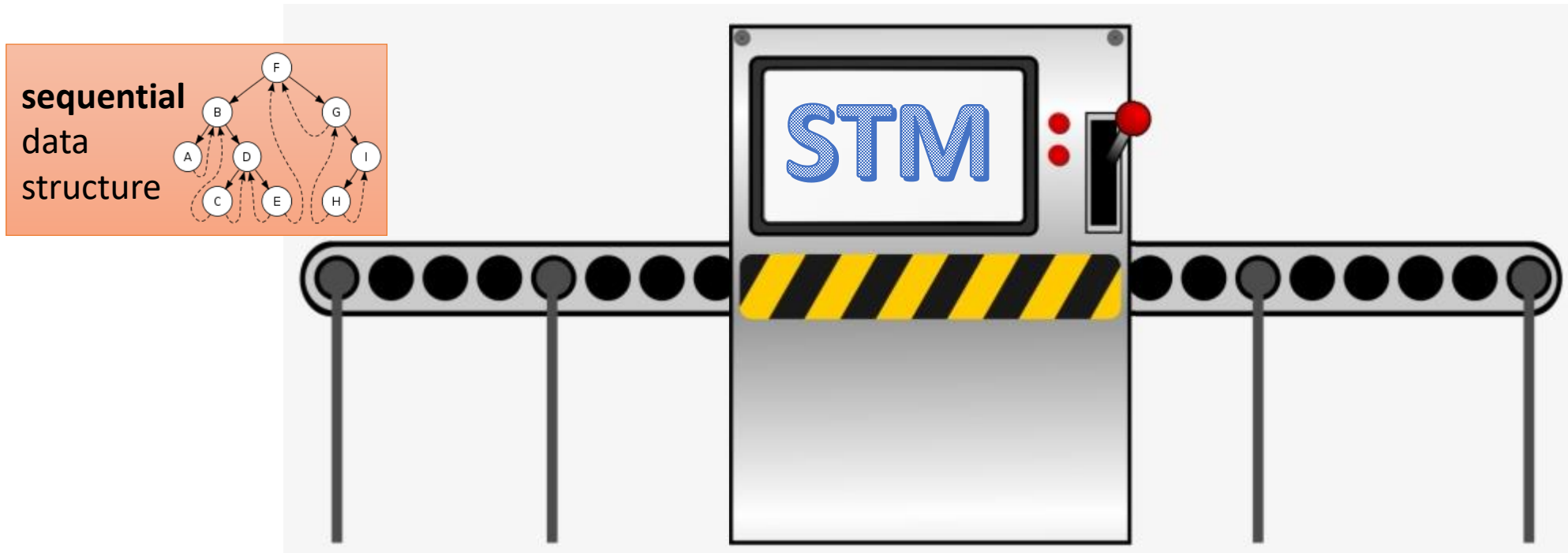
Correctness: Proofs can have errors.

Memory Reclamation: Adding lock-free memory reclamation adds time and work.

What if...

What if there was a device that could turn sequential data structures into correct concurrent data structures with wait-free progress?

There is and it is called **wait-free software transactional memory (STM)** or UC



Back to 1991...

In 1991 Maurice Herlihy introduced the concept of a **Universal Construction** with wait-free progress and then showed an algorithm for a UC that actually worked.

“Wait-Free Synchronization”, M. Herlihy, 1991

Using this UC, for the first time ever it was possible to make wait-free data structures without expert knowledge.

A UC provides wait-free progress for a single data structure instance. What if we want to have operations over multiple instances?



Back to 1993...

In 1993 Maurice Herlihy and J. Elliot B. Moss introduced the concept of a **Transactional Memory**.

“Transactional Memory: Architectural Support for Lock-Free Data Structures”, M. Herlihy and J. Moss, 1991

Later in 1997, Nir Shavit and Dan Touitou presented the first software only implementation of a TM, a **Software Transactional Memory (STM)**.

The decade of 2000-2010 saw an explosion in the field of STMs.



Back to three weeks ago...

Three weeks ago *we* (Pedro Ramalhete, Andreia Correia, Pascal Felber and Nachshon Cohen) presented OneFile the first wait-free STM.

“OneFile: A Wait-Free Persistent Transactional Memory”, P. Ramalhete, A. Correia, P. Felber and N. Cohen, 2019

OneFile is user-level library where you just include one header file with 1 kLOC and you’re ready to make wait-free data structures.

It serializes mutative transactions and needs DCAS.

It has wait-free memory reclamation and works on Persistent Memory.

With Onefile, making a new wait-free data structure is just a matter of annotating a sequential data structure implementation.

What do you mean by *annotate*?

Transforming sequential code to a wait-free code requires executing the following steps:

1. Add the include for OneFile header
2. Annotate the basic types with `tmttype<T>`
3. Complex types (nodes) should extend `tmbase`
4. Place the code of the functions inside a lambda and pass it to OneFile
5. Replace calls to `new<T>` with `tmNew<T>`
6. Replace calls to `delete x` with `tmDelete(x)`


```

template<typename T> struct OFWFQueue {
    struct Node {
        T* item;
        Node* next;
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    Node* head {nullptr};
    Node* tail {nullptr};
    OFWLinkedListQueue() {
        Node* sentinelNode = new Node(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        Node* newNode = new Node(item);
        tail->next = newNode;
        tail = newNode;
        return true;
    }
    T* dequeue() {
        Node* lhead = head;
        if (lhead == tail) return nullptr;
        head = lhead->next;
        delete lhead;
        return head->item;
    }
};



```

```

#include "OneFileWF.hpp"
template<typename T> struct OFWFQueue {
    struct Node {
        T* item;
        Node* next;
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    Node* head {nullptr};
    Node* tail {nullptr};
    OFWLinkedListQueue() {
        Node* sentinelNode = new Node(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        Node* newNode = new Node(item);
        tail->next = newNode;
        tail = newNode;
        return true;
    }
    T* dequeue() {
        Node* lhead = head;
        if (lhead == tail) return nullptr;
        head = lhead->next;
        delete lhead;
        return head->item;
    }
};

```

```

#include "OneFileWF.hpp"
template<typename T> struct OFWFQueue {
    struct Node {
        T* item;
        ofwf::tmttype<Node*> next; 
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    ofwf::tmttype<Node*> head {nullptr}; 
    ofwf::tmttype<Node*> tail {nullptr}; 
    OFWFLinkedListQueue() {
        Node* sentinelNode = new Node(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        Node* newNode = new Node(item);
        tail->next = newNode;
        tail = newNode;
        return true;
    }
    T* dequeue() {
        Node* lhead = head;
        if (lhead == tail) return nullptr;
        head = lhead->next;
        delete lhead;
        return head->item;
    }
};

```

```

#include "OneFileWF.hpp"
template<typename T> struct OFWFQueue {
    struct Node {
        T* item;
        Node* next;
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    Node* head {nullptr};
    Node* tail {nullptr};
    OFWFLinkedListQueue() {
        Node* sentinelNode = new Node(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        Node* newNode = new Node(item);
        tail->next = newNode;
        tail = newNode;
        return true;
    }
    T* dequeue() {
        Node* lhead = head;
        if (lhead == tail) return nullptr;
        head = lhead->next;
        delete lhead;
        return head->item;
    }
};

```

```

#include "OneFileWF.hpp"
template<typename T> struct OFWFQueue {
    struct Node {
        T* item;
        ofwf::tmtype<Node*> next;
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    ofwf::tmtype<Node*> head {nullptr};
    ofwf::tmtype<Node*> tail {nullptr};
    OFWFLinkedListQueue() {
        Node* sentinelNode = new Node(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        Node* newNode = new Node(item);
        tail->next = newNode;
        tail = newNode;
        return true;
    }
    T* dequeue() {
        Node* lhead = head;
        if (lhead == tail) return nullptr;
        head = lhead->next;
        delete lhead;
        return head->item;
    }
};



```

```

#include "OneFileWF.hpp"
template<typename T> struct OFWFQueue : public ofwf::tmbase{
    struct Node : public ofwf::tmbase {
        T* item;
        ofwf::tmtype<Node*> next;
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    ofwf::tmtype<Node*> head {nullptr};
    ofwf::tmtype<Node*> tail {nullptr};
    OFWFLinkedListQueue() {
        Node* sentinelNode = new Node(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        Node* newNode = new Node(item);
        tail->next = newNode;
        tail = newNode;
        return true;
    }
    T* dequeue() {
        Node* lhead = head;
        if (lhead == tail) return nullptr;
        head = lhead->next;
        delete lhead;
        return head->item;
    }
};

```

```

#include "OneFileWF.hpp"
template<typename T> struct OFWFQueue : public ofwf::tmbase {
    struct Node : public ofwf::tmbase {
        T* item;
        ofwf::tmtype<Node*> next;
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    ofwf::tmtype<Node*> head {nullptr};
    ofwf::tmtype<Node*> tail {nullptr};
    OFWFLinkedListQueue() {
        Node* sentinelNode = new Node(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        return ofwf::updateTx<bool>([=] () -> bool { 
            Node* newNode = new Node(item);
            tail->next = newNode;
            tail = newNode;
        });
        return true;
    }
    T* dequeue() {
        return (T*)ofwf::updateTx<T*>([=] () -> T* { 
            Node* lhead = head;
            if (lhead == tail) return nullptr;
            head = lhead->next;
            delete lhead;
        });
        return head->item;
    }
}

```

```

#include "OneFileWF.hpp"
template<typename T> struct OFWFQueue : public ofwf::tmbase {
    struct Node : public ofwf::tmbase {
        T* item;
        ofwf::tmtype<Node*> next;
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    ofwf::tmtype<Node*> head {nullptr};
    ofwf::tmtype<Node*> tail {nullptr};
    OFWFLinkedListQueue() {
        Node* sentinelNode = new Node(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        Node* newNode = new Node(item);
        tail->next = newNode;
        tail = newNode;
        return true;
    }
    T* dequeue() {
        Node* lhead = head;
        if (lhead == tail) return nullptr;
        head = lhead->next;
        delete lhead;
        return head->item;
    }
};

```

```

#include "OneFileWF.hpp"
template<typename T> struct OFWFQueue : public ofwf::tmbase {
    struct Node : public ofwf::tmbase {
        T* item;
        ofwf::tmtype<Node*> next;
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    ofwf::tmtype<Node*> head {nullptr};
    ofwf::tmtype<Node*> tail {nullptr};
    OFWFLinkedListQueue() {
        Node* sentinelNode = new Node(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        return ofwf::updateTx<bool>([=] () -> bool {
            Node* newNode = new Node(item);
            tail->next = newNode;
            tail = newNode;
        });
        return true;
    }
    T* dequeue() {
        return (T*)ofwf::updateTx<T*>([=] () -> T* {
            Node* lhead = head;
            if (lhead == tail) return nullptr;
            head = lhead->next;
            delete lhead;
        });
        return head->item;
    }
}

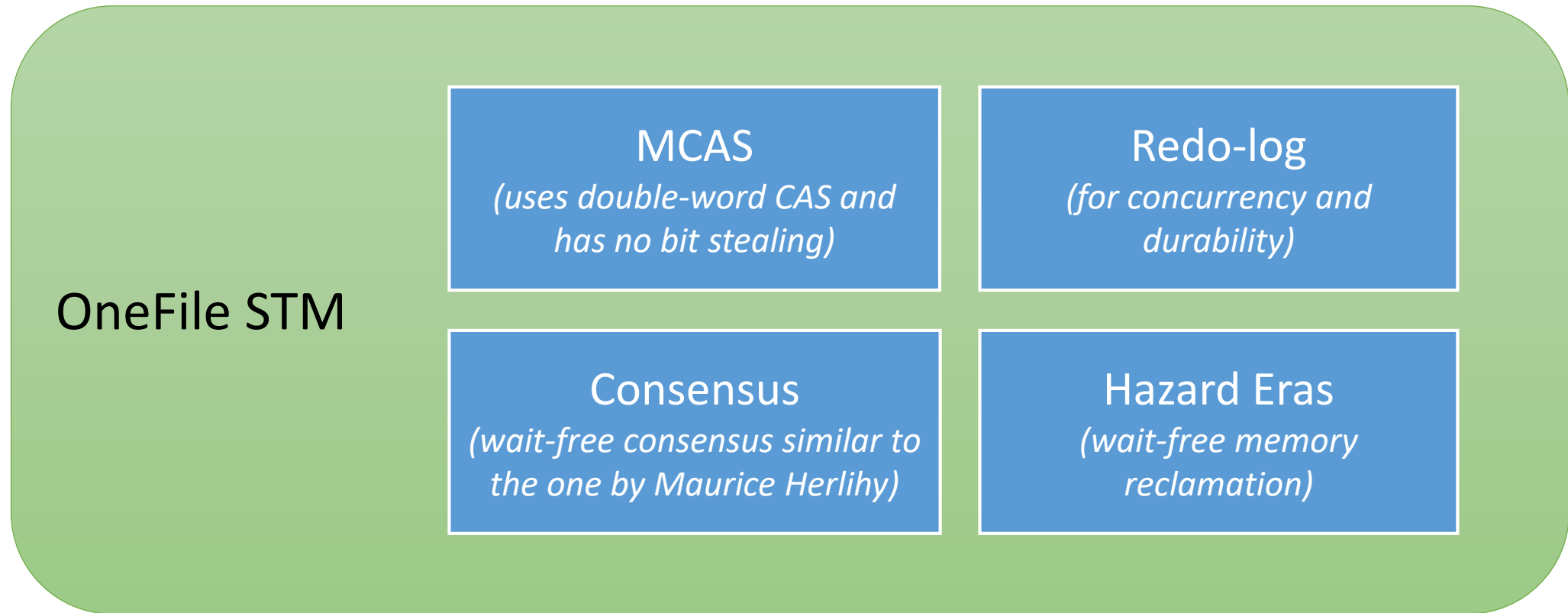
```

```

#include "OneFileWF.hpp"
template<typename T> struct OFWFQueue : public ofwf::tmbase {
    struct Node : public ofwf::tmbase {
        T* item;
        ofwf::tmtype<Node*> next;
        Node(T* userItem) : item{userItem}, next{nullptr} { }
    };
    ofwf::tmtype<Node*> head {nullptr};
    ofwf::tmtype<Node*> tail {nullptr};
    OFWFLinkedListQueue() {
        Node* sentinelNode = ofwf::tmNew<Node>(nullptr);
        head = sentinelNode;
        tail = sentinelNode;
    }
    bool enqueue(T* item) {
        if (item == nullptr) throw std::invalid_argument("null item");
        return ofwf::updateTx<bool>([=] () -> bool {
            Node* newNode = ofwf::tmNew<Node>(item);
            tail->next = newNode;
            tail = newNode;
            return true;
        });
    }
    T* dequeue() {
        return (T*)ofwf::updateTx<T*>([=] () -> T* {
            Node* lhead = head;
            if (lhead == tail) return nullptr;
            head = lhead->next;
            ofwf::tmDelete(lhead);
            return head->item;
        });
    }
}

```

Anatomy of a wait-free STM



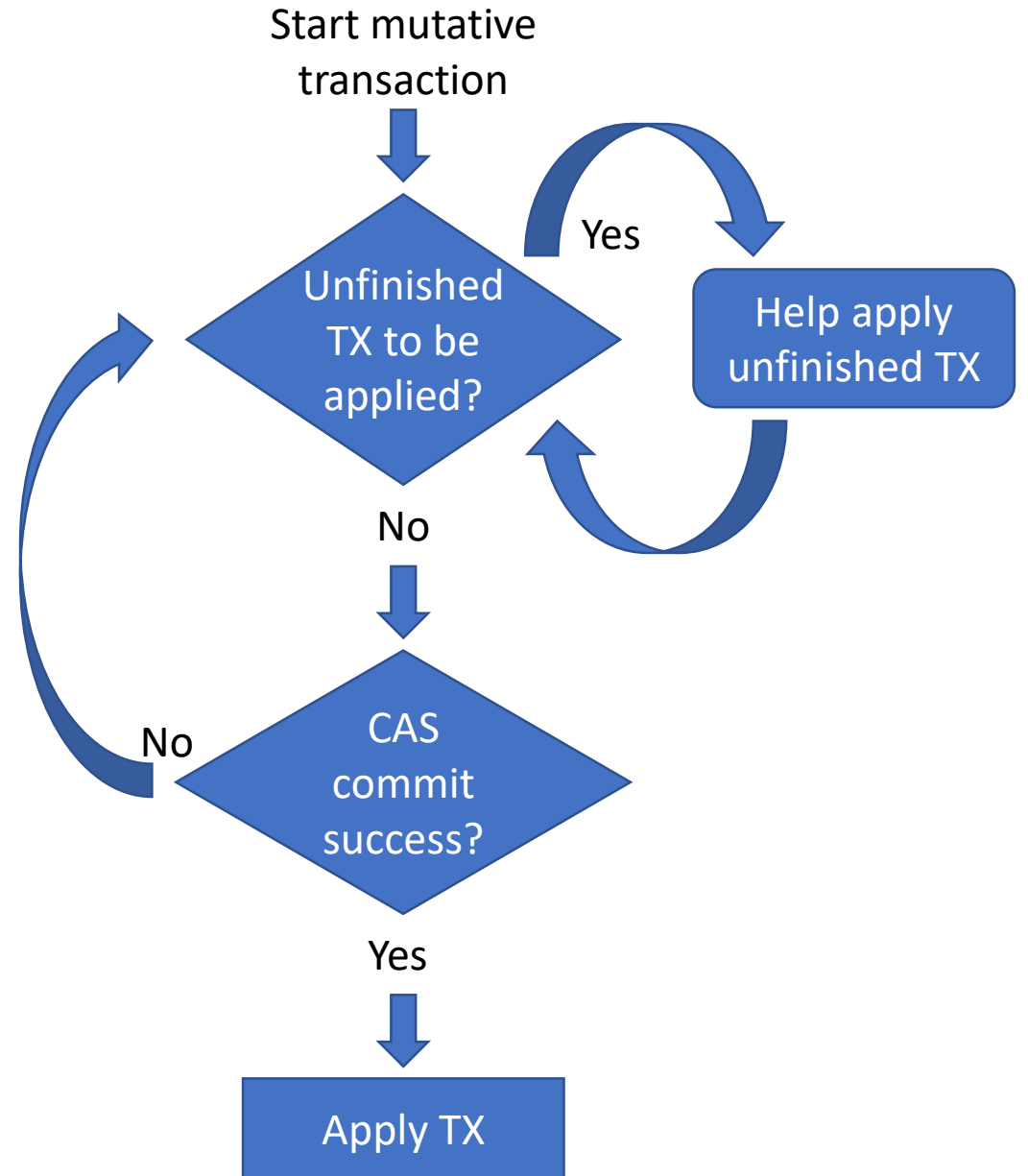


OneFile Lock-Free Algorithm

Lock-free mutative transactions

Mutative (write) transactions in OneFile have three distinct phases:

1. **Transform:** The thread takes a lambda or `std::function`, simulates execution and saves every modification on a write-set (redo-log)
If a newer (inconsistent) modification is observed, it aborts simulation and restarts
2. **Commit:** The thread attempts to commit its write-set using a CAS on a global variable (`curTx`) as being the next modification to be applied
3. **Apply:** The modifications in the committed write-set are applied in memory using one DCAS instruction per modified word



Using OneFile lock-free on a stack

```
[=] () { // push 42 on stack
    Node* node = tmNew<Node>();
    node->key = 42;
    node->next = head;
    head = node;
}
```



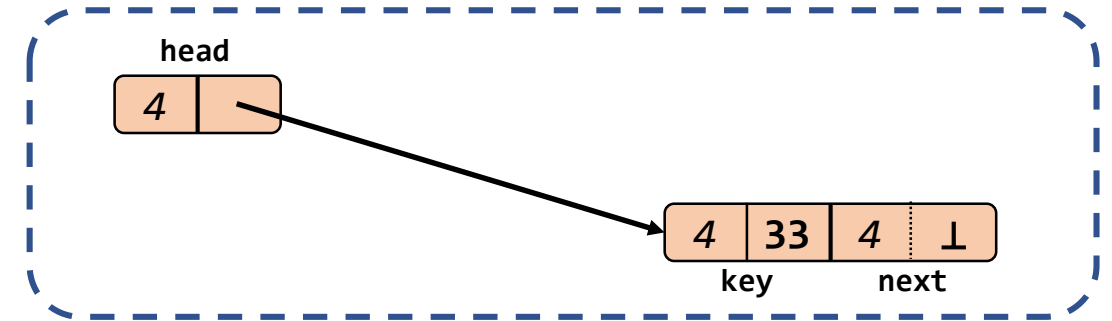
```
[=] () { // pop from stack
    Node* node = head;
    int key = head->key;
    head = head->next;
    tmDelete(node);
    return key;
}
```



```
[=] () { // push 21 on stack
    Node* node = tmNew<Node>();
    node->key = 21;
    node->next = head;
    head = node;
}
```



4	0	curTx
seq	tid	
4	0	request
seq	tid	



write-set for
thread id 0

addr	value

write-set for
thread id 1

addr	value

write-set for
thread id 2

addr	value

Using OneFile lock-free on a stack

```
[=] () { // push 42 on stack
  Node* node = tmNew<Node>();
  node->key = 42;
  node->next = head;
  head = node;
}
```



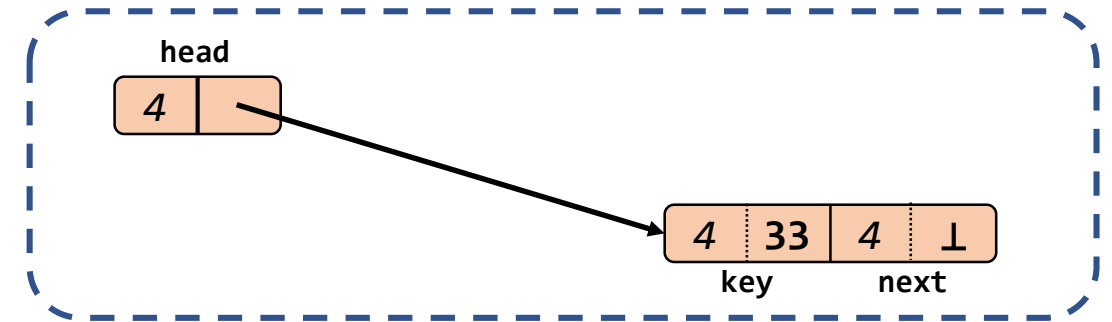
```
[=] () { // pop from stack
  Node* node = head;
  int key = head->key;
  head = head->next;
  tmDelete(node);
  return key;
}
```



```
[=] () { // push 21 on stack
  Node* node = tmNew<Node>();
  node->key = 21;
  node->next = head;
  head = node;
}
```



4	0	curTx
seq	tid	
4	0	request
seq	tid	



write-set for
thread id 0

addr value



write-set for
thread id 1

addr value



write-set for
thread id 2

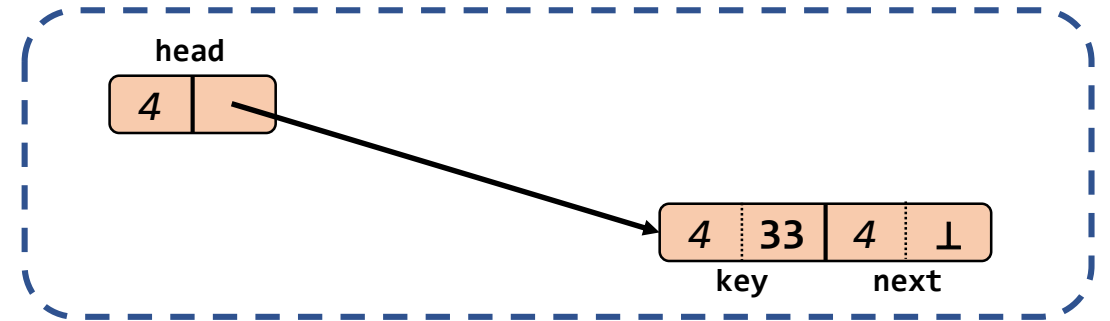
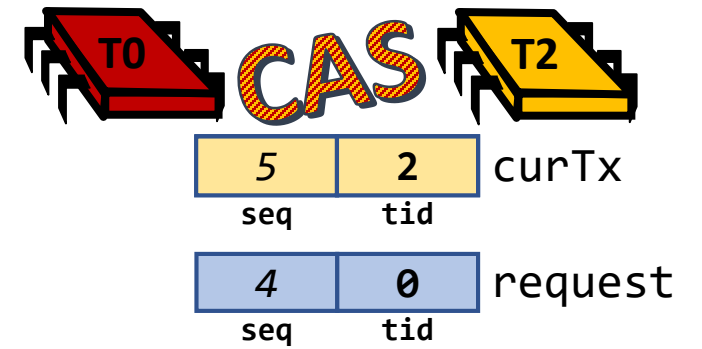
addr value

Using OneFile lock-free on a stack

```
[=] () { // push 42 on stack
    Node* node = tmNew<Node>();
    node->key = 42;
    node->next = head;
    head = node;
}
```

```
[=] () { // pop from stack
    Node* node = head;
    int key = head->key;
    head = head->next;
    tmDelete(node);
    return key;
}
```

```
[=] () { // push 21 on stack
    Node* node = tmNew<Node>();
    node->key = 21;
    node->next = head;
    head = node;
}
```



write-set for thread id 0

&key	42
&next	head
&head	node

addr value



write-set for thread id 1

&head	next

addr value



write-set for thread id 2

&key	21
&next	head
&head	node

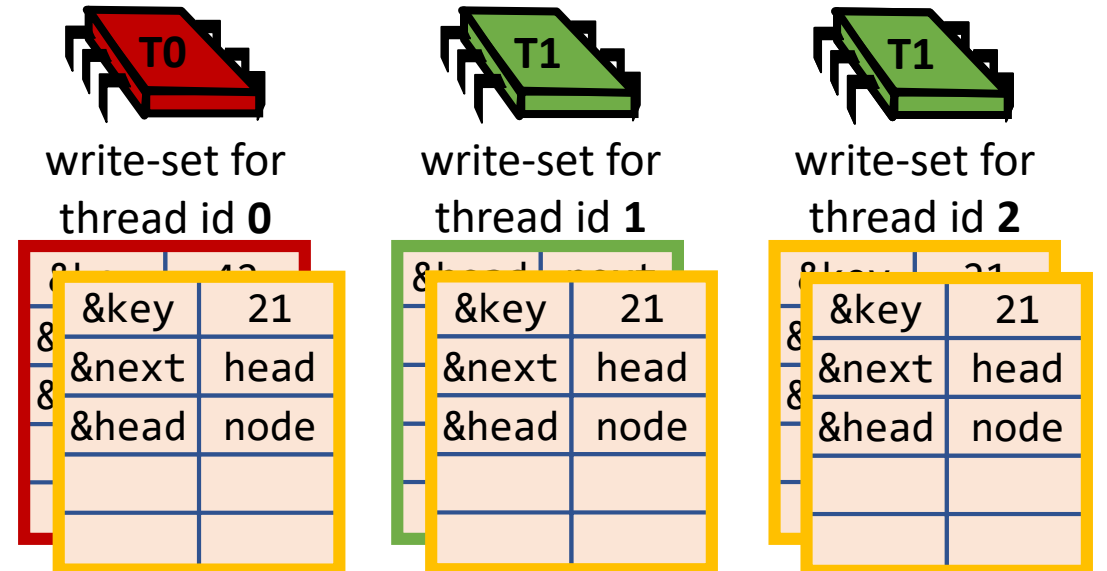
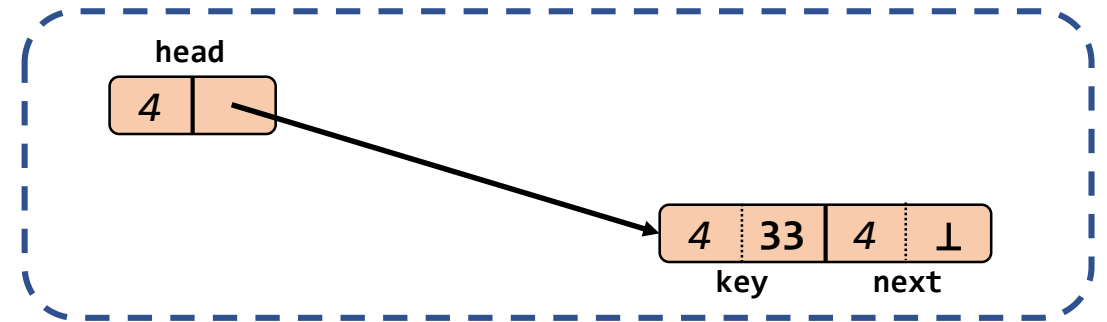
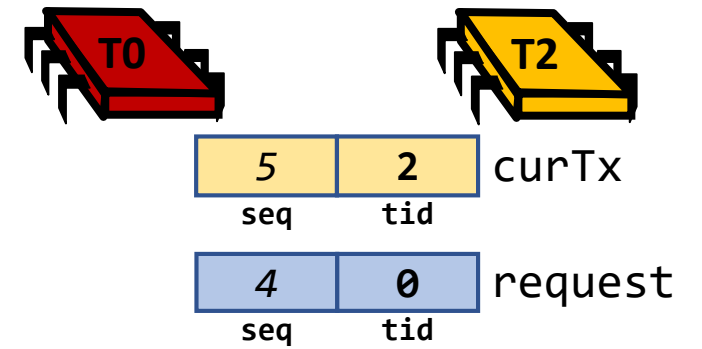
addr value

Using OneFile lock-free on a stack

```
[=] () { // push 42 on stack
    Node* node = tmNew<Node>();
    node->key = 42;
    node->next = head;
    head = node;
}
```

```
[=] () { // pop from stack
    Node* node = head;
    int key = head->key;
    head = head->next;
    tmDelete(node);
    return key;
}
```

```
[=] () { // push 21 on stack
    Node* node = tmNew<Node>();
    node->key = 21;
    node->next = head;
    head = node;
}
```



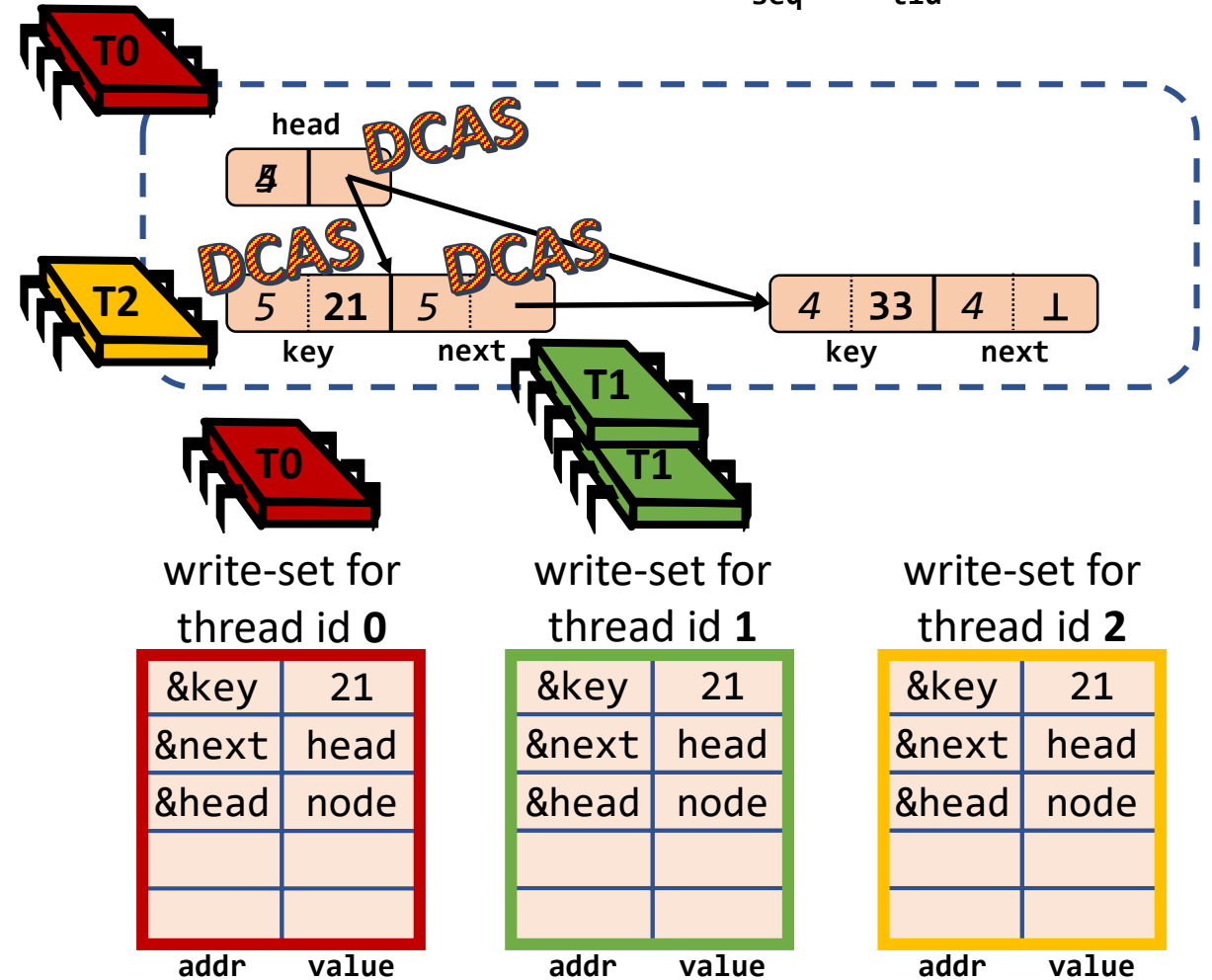
Using OneFile lock-free on a stack

```
[=] () { // push 42 on stack
    Node* node = tmNew<Node>();
    node->key = 42;
    node->next = head;
    head = node;
}
```

```
[=] () { // pop from stack
    Node* node = head;
    int key = head->key;
    head = head->next;
    tmDelete(node);
    return key;
}
```

```
[=] () { // push 21 on stack
    Node* node = tmNew<Node>();
    node->key = 21;
    node->next = head;
    head = node;
}
```

5	2	curTx
seq	tid	
4	0	request
seq	tid	

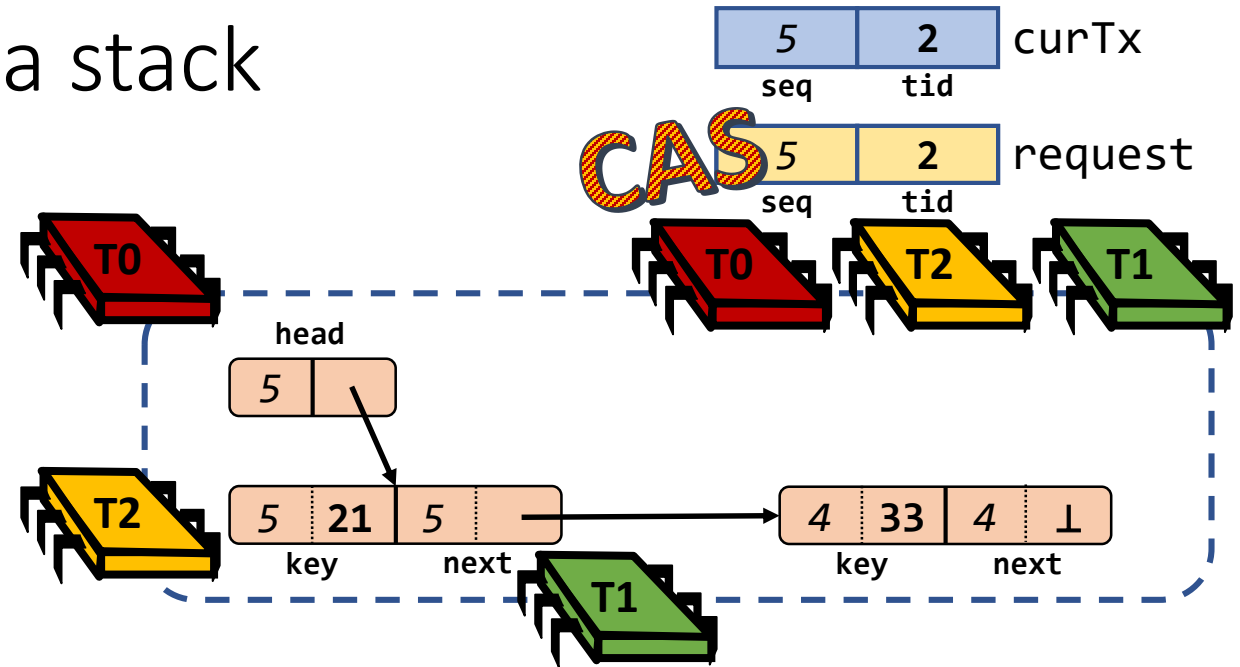


Using OneFile lock-free on a stack

```
[=] () { // push 42 on stack
    Node* node = tmNew<Node>();
    node->key = 42;
    node->next = head;
    head = node;
}
```

```
[=] () { // pop from stack
    Node* node = head;
    int key = head->key;
    head = head->next;
    tmDelete(node);
    return key;
}
```

```
[=] () { // push 21 on stack
    Node* node = tmNew<Node>();
    node->key = 21;
    node->next = head;
    head = node;
}
```



write-set for
thread id 0

&key	21
&next	head
&head	node
addr	value

write-set for
thread id 1

&key	21
&next	head
&head	node
addr	value

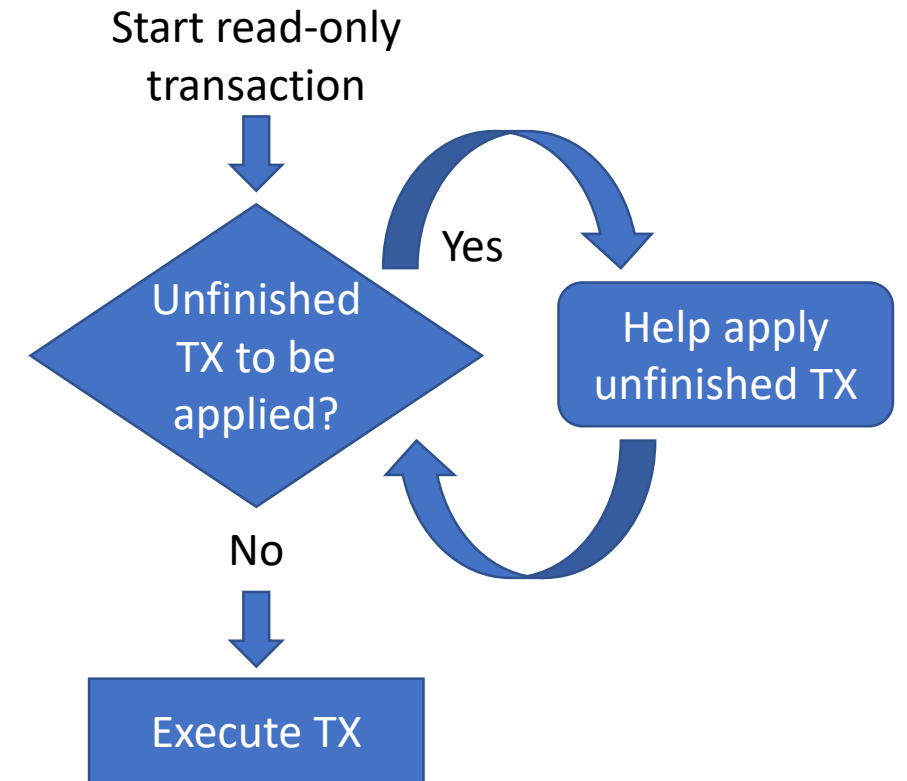
write-set for
thread id 2

&key	21
&next	head
&head	node
addr	value

Lock-free read-only transactions

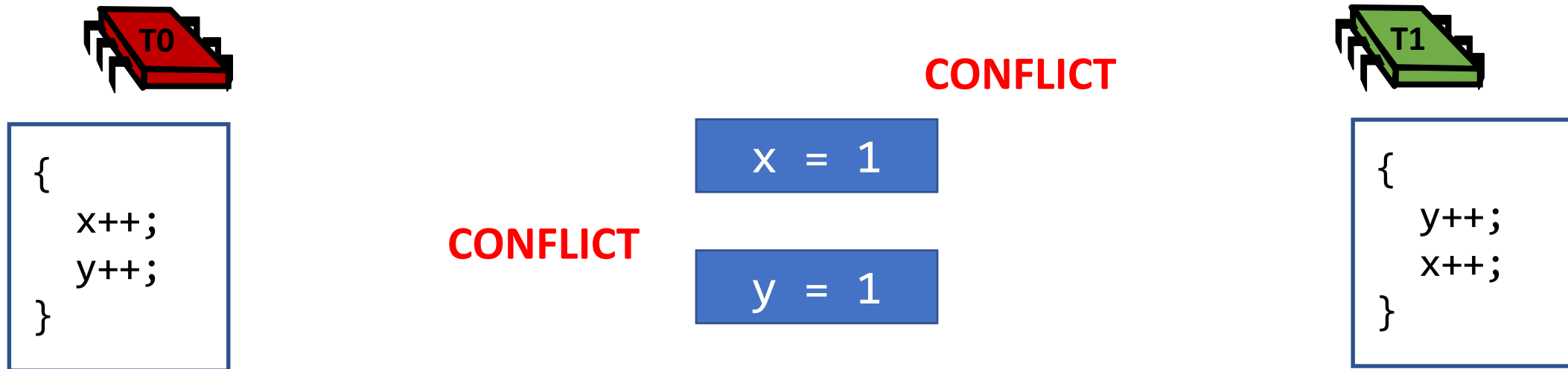
Read-only transactions in OneFile use an optimistic technique, similar to TinySTM/TL2

- Before starting execution of the lambda or `std::function`, the thread checks if there is an unfinished mutative transaction and helps apply it if there is
- Then, it reads the global clock (`curTx`) and saves the timestamp
- The load interposing does not keep a read-set: for every word read, it checks if the associated timestamp precedes the timestamp read from the global clock at the beginning of the transaction



What about Hardware Transactional Memory (HTM)?

In all current hardware implementations of HTM there is no guarantee of progress
Even simple operations can execute an unbounded number of abort/retry attempts





OneFile Wait-Free Algorithm

Reaching consensus

As Maurice Herlihy pointed out, to make a wait-free algorithm we need a *wait-free consensus*.

In shared memory concurrency, there are three algorithms to achieve wait-free consensus:

1. Lamport's Bakery Algorithm

"A New solution of Dijkstra's Concurrent Programming Problem", L. Lamport, 1974

Used in the Kogan-Petrunk wait-free queue, A. Kogan and E. Petrunk, 2011

2. Herlihy's Combining Consensus

"A Methodology for Implementing Highly Concurrent Data Objects", M. Herlihy, 1993


Used by PSim and the SimQueue wait-free queue, P. Fatourou and N. Kallimanis, 2014

3. Turn consensus

"Mutual Exclusion — Two linear wait solutions", A. Correia and P. Ramalhete, 2015

Used by TurnQueue wait-free queue, P. Ramalhete and A. Correia, 2017

In OneFile we use a modified variant of Herlihy's Combining Consensus

A photograph of a person in dark clothing climbing a very tall, narrow, and jagged rock spire. The climber is positioned about halfway up the spire, which is a light tan color. The background shows a vast, cloudy sky and distant, snow-capped mountain ranges. A white curved line separates the image from the dark grey background on the right.

Sure, it's wait-free, but does it scale?

Scaling is easy... for read-mostly workloads

If most of the operations are **read-only**, then there are multiple *generic* techniques that provide good scalability:

Reader-Writer locks: *C-RW-WP/RP* are two RW-Lock algorithms that provide high read scalability.

UCs: *Left-Right* and *CX* scale well for read-only workloads and accept anything you can put on a lambda and doesn't have side-effects.

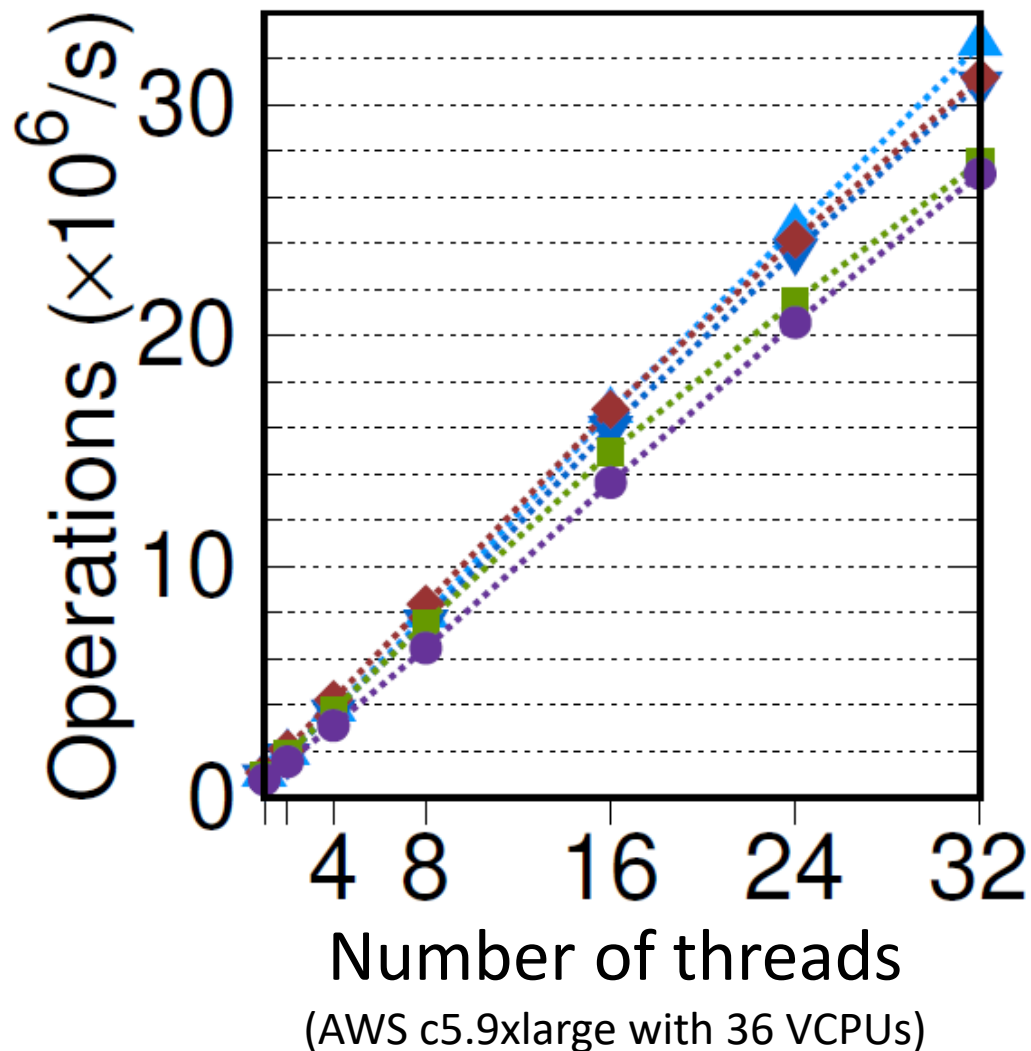
STMs: Some STMs like *TL2*, *TinySTM* and *OneFile* have highly scalable read-only operations.

Not-so-generic-techniques that scale well for reads: RLU, RCU, and others

RW-Lock ▲
 Left-Right ▼
 CX ◆
 OneFile-WF ■
 TL2 ●

As long as it's just read-only operations, all these techniques scale with the number of cores.

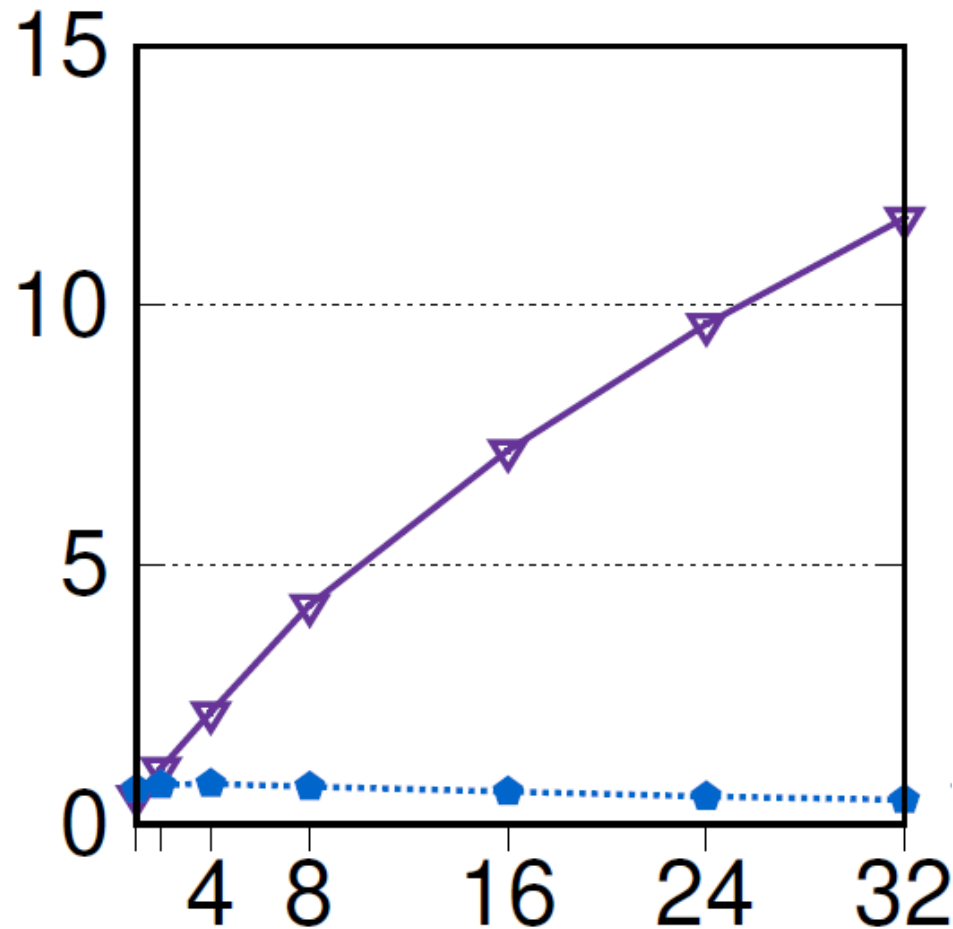
Operations are lookups on a red-black tree with 1M keys



Generic techniques that scale well for readers	Writers progress	Readers progress
C-RW-WP	Blocking starvation-free	Blocking
C-RW-RP	Blocking	Blocking starvation-free
Left-Right UC	Blocking starvation-free	Wait-free population oblivious
CX UC	Wait-free bounded	Wait-free bounded
TL2, TinySTM *	Blocking	Blocking
OneFile *	Wait-free bounded	Wait-free bounded

* STMs need load and store annotation

Scaling is hard... for mutative operations



Sequential code has cache locality but it is limited by how fast one core can execute the code.

Sequential code protected with a mutual exclusion lock has flat or negative scalability.

STMs have the overhead of logging, synchronization and others.

OneFile and CX scale for read-only transactions but are flat for mutative transactions

Originally, blocking STMs were able to scale up to ~16 threads, but recently they can scale almost linearly for high cores counts, even for short mutative transactions (assuming DAP workloads)

The D in A.C.I.D.

A is for Atomicity

C is for Consistency

I is for Isolation

D is for Durability

Consistency in databases

DBMS	Default Isolation	Maximum Isolation
Aerospike	Read Committed	Read Committed
MySQL 5.6	Repeatable Read	Serializability
MS SQL Server 2012	Read Committed	Serializability
NuoDB	Consistent Read	Consistent Read
Oracle 11g	Read Committed	Snapshot Isolation
Oracle Berkley DB	Serializability	Serializability
Postgres 9.2.2	Read Committed	Serializability
SAP HANA	Read Committed	Snapshot Isolation
VoltDB	Serializability	Serializability

Source: Peter Bailis <http://www.bailis.org/blog/when-is-acid-acid-rarely/>

Consistency in ACID transactions

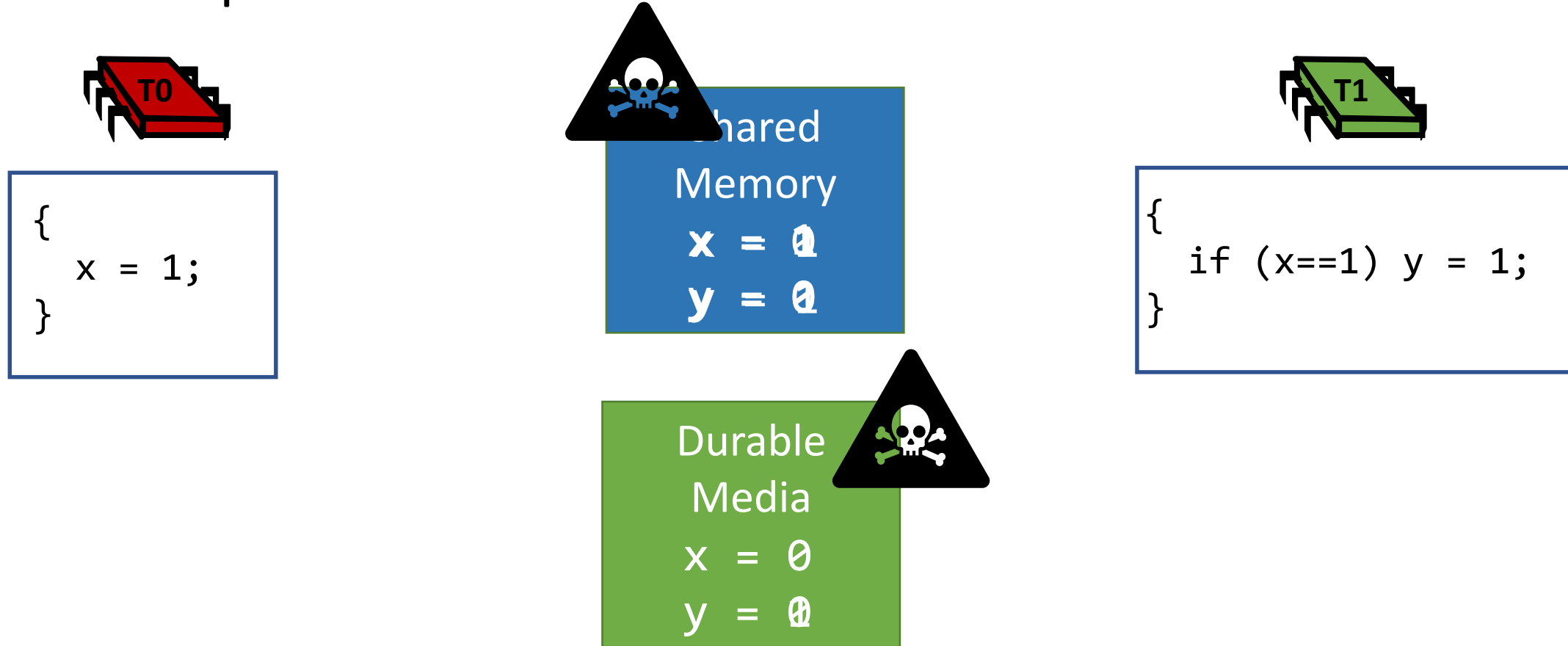
When transactions are both concurrent and durable (ACID) then what kind of consistency should we aim for?

If we want to keep strong consistency then the obvious choice is ***durable linearizability***, (introduced by Izraelevitz, Mendes and Scott, DISC 2016)

Roughly speaking, a durable linearizable transaction can not become visible to transactions on other threads unless its effects are also durable.

Consistency in ACID transactions

Example of **not** Durable Linearizable



Durable Linearizability

A.C.I.D.
Transactions
(Mnemosyne, OneFile)

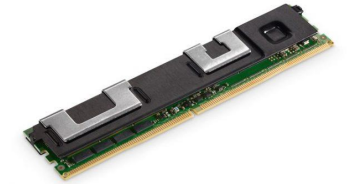
Durable
Media

Durable
Transactions

Concurrent
Transactions

Persistence
Library
(vista, libpmemobj)

STM Library
(TL2, TinySTM ...)



Persistent Memory
~170 – 320 ns



SSD/Disk storage
~1 ms – 100 ms

Network replication
~100 ms – 10000 ms+

Ensuring Durability

Ensuring durable consistency (persistence) has a cost which depends on the media's latency.
At least **one round trip** message or synchronization fence is needed **to guarantee durability**.

User



Durable Linearizability

A.C.I.D.
Transactions
(Mnemosyne, OneFile)

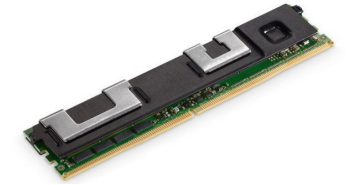
Durable
Media

Durable
Transactions

Concurrent
Transactions

Persistence
Library
(vista, libpmemobj)

STM Library
(TL2, TinySTM ...)



Persistent Memory
~170 – 320 ns
3 million tx/s



SSD/Disk storage
~1 ms – 100 ms
1000 tx/s

Network replication
~10 ms – 10000 ms
100 tx/s



Transactions in Persistent Memory (PTM)

What is Persistent Memory?

Persistent Memory is a durable media that can be **accessed through load and store instructions**.

Physically, it fits into a DIMM slot

Solutions exist for several years by HPE, Micron and Viking, but all these are battery backed:

<https://www.vikingtechnology.com/products/nvdimm/>

<https://www.hpe.com/nl/en/servers/persistent-memory.html>

<https://www.micron.com/campaigns/persistent-memory>

Just last month, Intel has released the Optane DC Persistent Memory which does not require battery and has latencies close to DRAM. Capacities go up to 512 GiB

<https://arxiv.org/pdf/1903.05714.pdf>

How to use Persistent Memory

Just like there is a *Memory Model* to program with atomics (concurrency), there is an equivalent *Persistent Model* to program with persistent memory.

- **pwb**: flushes a cache line. Ordering with respect to previous stores on the same cache line.
- **pfence**: asynchronous ordering fence. No pwb or stores will be re-ordered.
- **psync**: synchronous ordering fence. No pwb or stores will be re-ordered. After this, data is guaranteed to be persisted, i.e. *durable*.

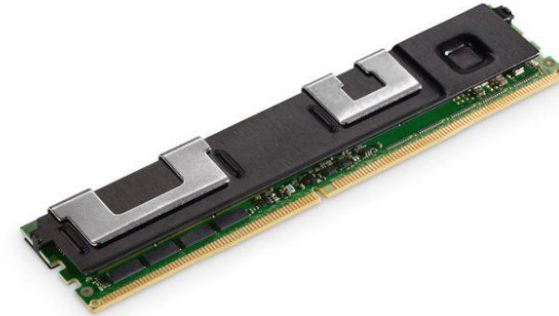
Introduced by Izraelevitz, Mendes and Scott, DISC 2016

How to execute failure-resilient operations?



On block devices (disks) the typical way that databases and filesystems ensure that data is stored consistently is to use:

- Undo-log, redo-log or Copy-On-Write (shadow copy)
- `fsync()` or `fdatasync()` to order writes to the device
- `fsync()` or `fdatasync()` to guarantee data is *durable*



On persistent memory the typical way that databases and filesystems ensure data is stored consistently is to use:

- Undo-log, redo-log or Copy-On-Write (shadow copy)
- `pwb (CLWB)` + `pfence (SFENCE)` to order writes to the device
- `psync (SFENCE)` to guarantee data is *durable*

Recovering from failures

One of the hardest tasks when implementing a DBMS or filesystem is the design and implementation of the recovery mechanism.

A failure can occur **anywhere** in the code... how to handle the different scenarios?

Recovery has to be *correct* and it should be *efficient*.

What if there was a way to not have a recovery at all?

Null Recovery

Null Recovery is a concept introduced by Izraelevitz, Mendes and Scott, DISC 2016

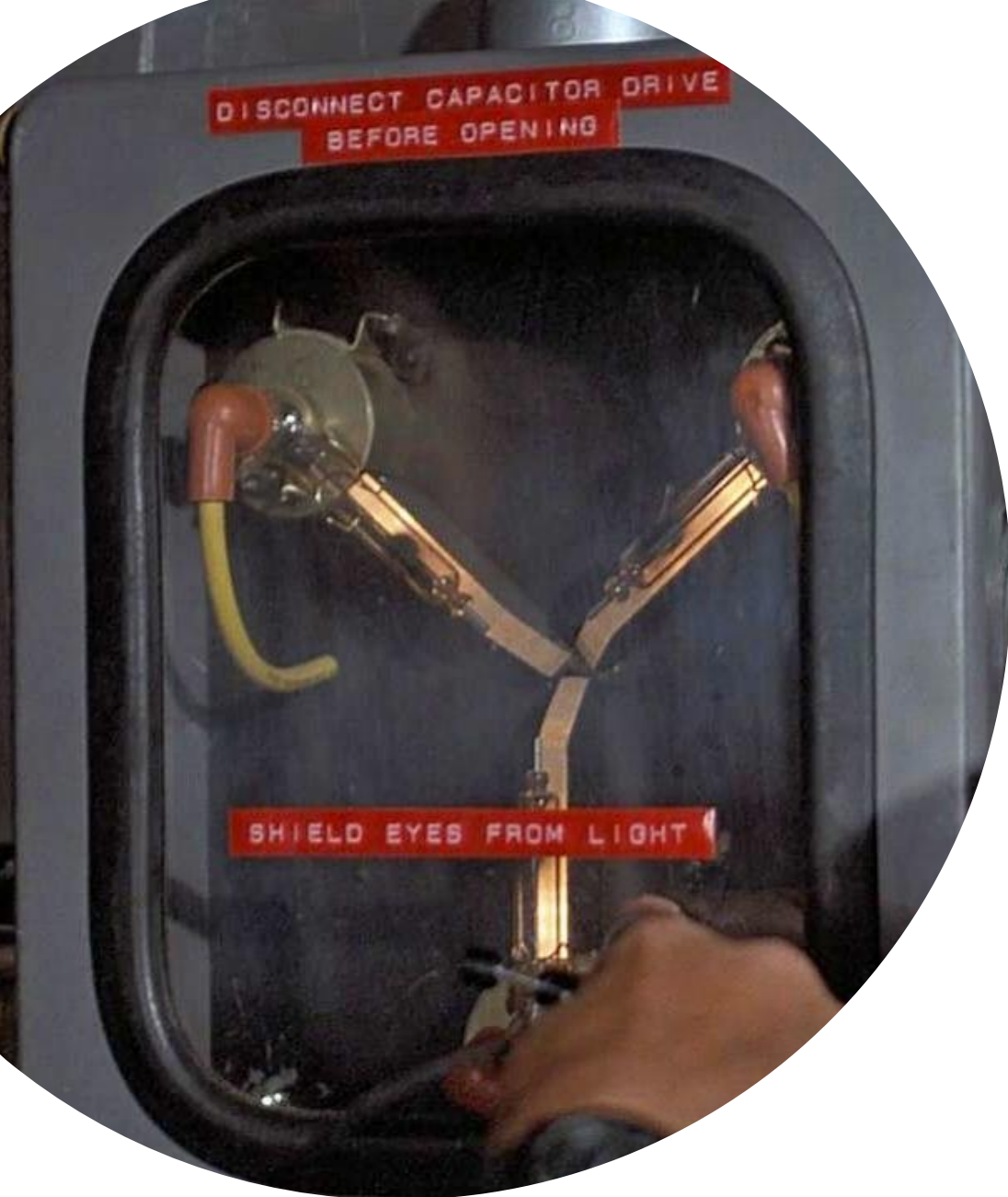
Roughly speaking, a *Lock-Free algorithm with null recovery does not need a recovery procedure when deployed in Persistent Memory*

Not all lock-free algorithms have this property of null recovery.

The OneFile PTM is itself a lock-free (or wait-free) algorithm and it *has* this property.

In the event of a failure, committed transactions in OneFile PTM are applied where they left off.

With **null recovery** and **Persistent Memory**, **recovery in OneFile PTM** is at most the time it takes to apply the last transaction, typically **less than a microsecond** (depends on the transaction size).



The Future

What we can do *now*

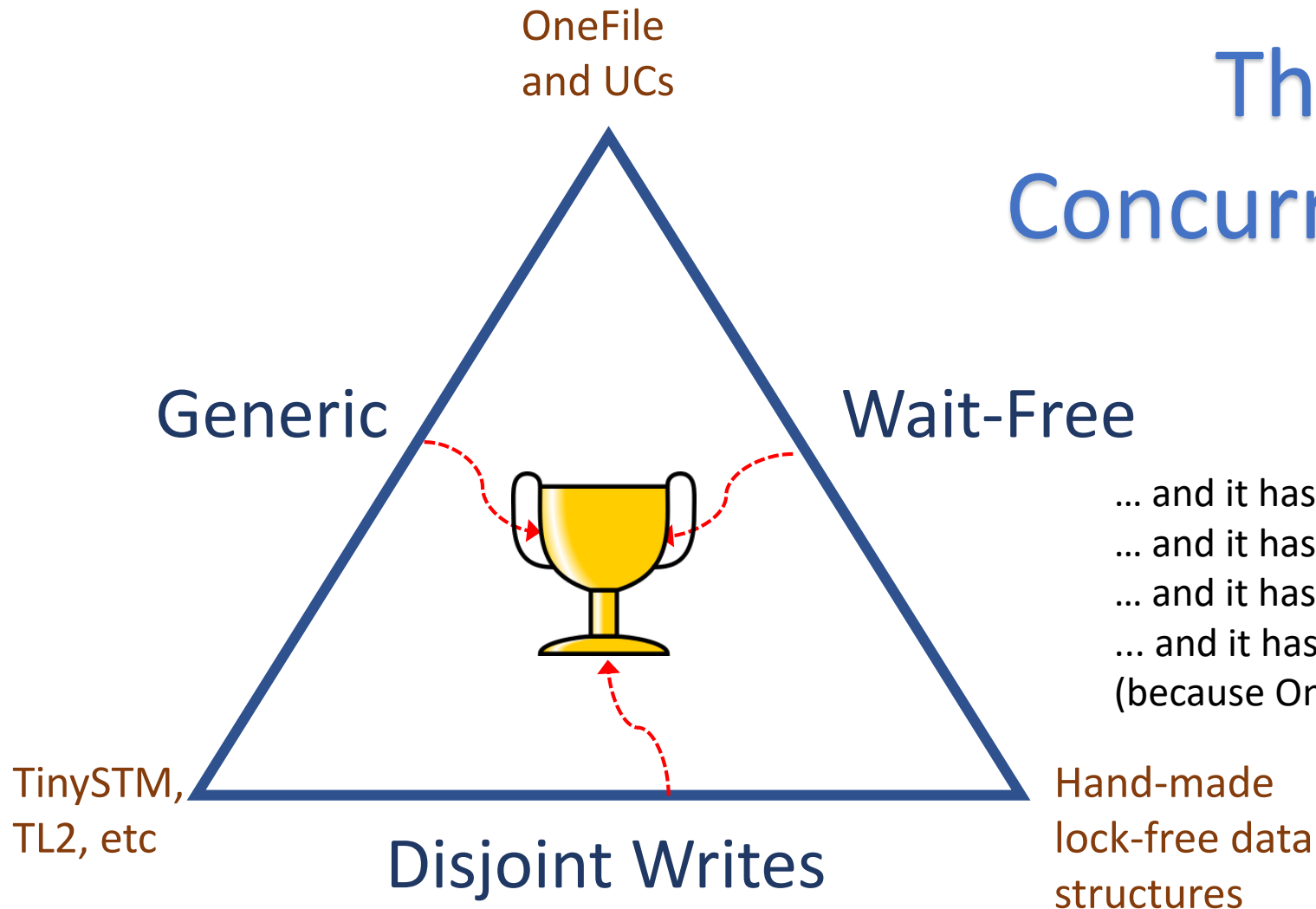


- **STMs are here to stay:** Transactions are a simple way of letting users have *concurrent atomicity* without having to worry about locks
- **Persistent Memory is here to stay:** Due to the overhead of drivers and filesystems, for the foreseeable future, making data durable on a block device will be orders of magnitude slower than achieving consistent durable data in PM
- **PTMs are the next step:** They combine durable and concurrent transactions (ACID) under a single programming interface.
- **Lock-Free PTMs:** This is the natural evolution for a PTM. Besides, STMs were originally developed with the intuit of making lock-free data structures.
- **Wait-Free Databases:** A PTM is in effect a **database engine**. By using a wait-free PTM we can make a wait-free database. *Look ma, no locks!*

The (*near*) Future

Is it possible to construct a technique that does all three?

The Holy Grail of Concurrent Data Structures



... and it has to scale for reads
... and it has to have wait-free memory reclamation
... and it has to have wait-free memory allocation/de-allocation
... and it has to work in persistent memory
(because OneFile already does all these)

The (*not so distant*) Future of K/V stores

What can we do with wait-free persistent (durable linearizable) transactions ?

What will a K/V store in PM look like in the future?

- We can make wait-free databases whose code is effectively sequential: concurrency and durability are taken care of by the PTM
- There is no need to write a recovery operation for the database: it's taken care by the null recovery in the PTM
- Any indexing data structure can be used for the tables in the DB and any kind of metadata in the DBMS will be under the transaction, including the indexing data structure and modifications to the records
- Operations on a wait-free DB have low tail latency: there are no locks

... but if we have generic dynamic transactions, do we even need a DBMS anymore?



What did we learn today?

- Lock-free is (mostly) about progress in the presence of failures, wait-free is (mostly) about low latency at the tail and progress. Neither are about throughput.
- All wait-free algorithms use a wait-free consensus. Only three consensus algorithms are known.
- Scaling for reads is easy. Scaling for DAP writes is hard. Scaling for non-DAP writes is impossible (conjecture).
- A wait-free STM or UC can transform a sequential data structure into a wait-free data structure with minimum modifications.

Bibliography

Creative Common Images

https://commons.wikimedia.org/wiki/File:Heraldic_hourglass.svg

https://commons.wikimedia.org/wiki/File:Simple_gold_cup.svg

The End

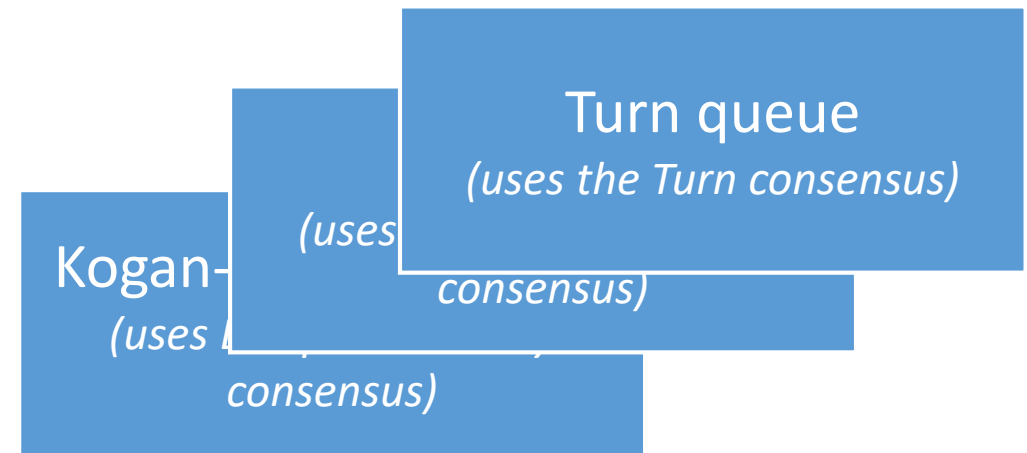
Pedro Ramalhete

<http://www.concurrencyfreaks.com>

pramalhe@gmail.com

Backup slides

Anatomy of a wait-free UC



CX UC

Multiple replicas of
the data

MPMC Wait-Free
queue

Strong Try RW-Lock

Hazard Pointers +
Ref counting
(wait-free memory
reclamation)

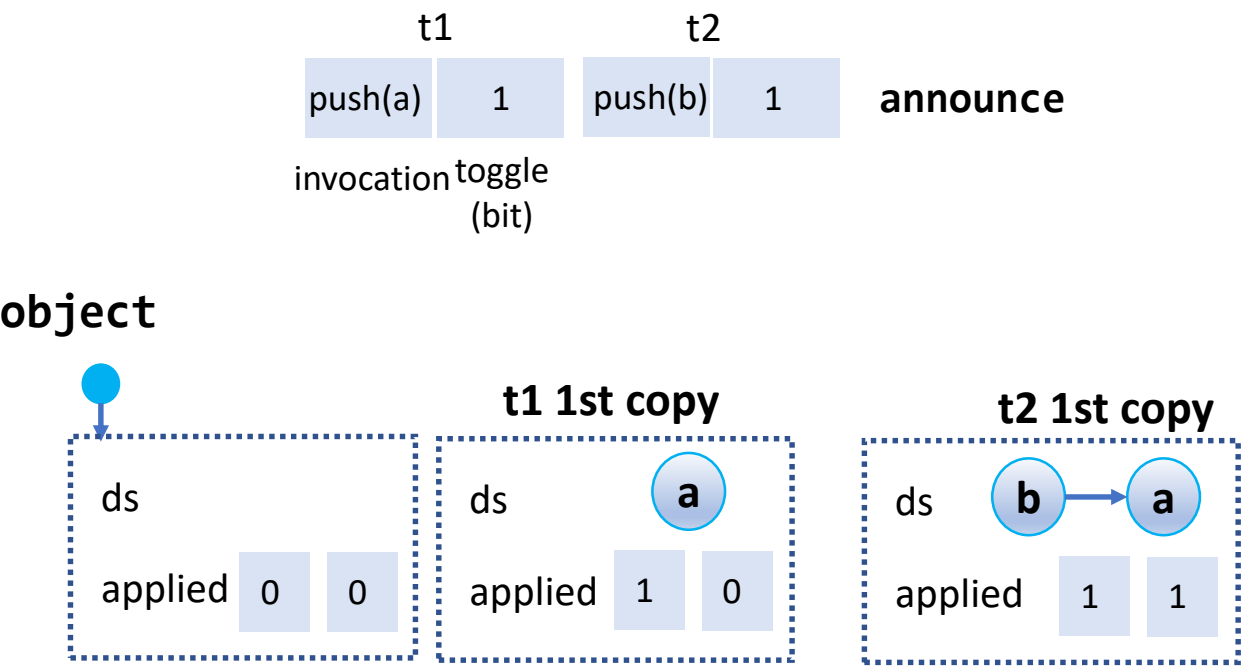
Reaching consensus

Herlihy's consensus

In Herlihy's original combining algorithm, each thread *announces* its operation. This is done by publishing a function pointer in `invocation` and then changing the `toggle` bit.

The thread that wins the LL/SC or CAS, proposes the new object with a new object where a particular set of operations have been applied and their respective responses.

To know which operations have already been applied, there is one reserved bit in each entry of both the `announce` and the responses. When the bit is different for a given position, the operation is yet to be applied.



Reaching consensus

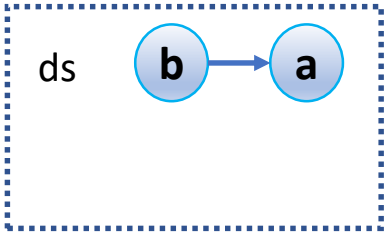
OneFile's consensus (variant of Herlihy)

Herlihy's consensus assumes that each thread works on its own private copy and applies all operations to it. It also assumes that each private copy has an associated **applied** array. OneFile has a single copy and a single array shared among all threads.

Instead of a bit, we use the sequence of the **announce** and **applied** entries. If the sequences match, then the operation is yet to be applied.

As part of the simulation, each operation will have trigger a store on respective entries of the **applied** array. The log will contain the modifications of all operations, including setting the respective **applied** entries to the new values, along with the new sequence which, by definition, will be different from the previous.

t1		t2		announce
push(a)	3	push(b)	4	
invocation		seq		
true	3	false	4	applied



What about proof checkers?

Lock-free algorithms (and implementations) can have errors. Is there any hope for tools that verify the correctness of such algorithms?

- SPIN
- JSF
- TLA+

However, the number of threads these can run is limited to 5 or 6 (state explosion for high thread count).

Invariant checking is possible, but it's up to the designer of the lock-free algorithm to come up with these invariants. Invariants may not hold when you modify the data structure... back to square 1.

Cognitive Load

The typical software developer has to:

- be proficient in one or more languages (development, UI, test)
- know multiple frameworks
- have basic knowledge of how the compiler/JIT/JVM works
- understand the HW (or at least the abstraction of the language)
- many others...

... and on top of this, understand concurrency?!?

