

A simple deterministic algorithm for guaranteeing the forward progress of transactions

Leiserson, 2016

Presented by Elie Cuevas
MIT 6.5060 – Algorithm Engineering

What This Talk Will Cover

- A brief review of currency control in parallel computing and existing mechanisms
- An explanation of *Transactional Memory* built on *Transactions*
- **A novel algorithm to ensure forward progress in any set of transactions**
- Correctness arguments for that algorithm
- Real-world complications of the algorithm
- Open problems and other notes

Concurrency Control

```
void main() {  
    // function1 and function2 read and write to the same memory locations  
    spawn function1();  
    function2();  
    return;  
}
```

- Functions that access the same memory locations called in parallel might exhibit **nondeterministic** behavior if the programmer is not careful.
- Inconsistent interweaving of memory accesses due to scheduling differences cause **data races**.
- Concurrency control ensures that results are correct and consistent.

Common Solution: Locking

- Locks require a thread to “obtain” permission from another source to access memory locations.
- Common locking mechanisms include **mutexes** and **semaphores**.

```
void function1(int value) {  
    // the array A in this example is locked by a mutex  
    acquire(A_LOCK);  
    A[1] = value;  
    release(A_LOCK);  
    return;  
}
```

Locking can be Problematic:

- Deadlocks: unbreakable sequence of threads waiting on each other
- Priority inversion: high-priority threads have to wait on completion of low-priority threads
- Overhead per resource: locks might be cumbersome to use in practice
- **LOSS OF PARALLELISM!**

Common Solution: Nonblocking Algorithms

- Nonblocking mechanisms cannot cause a thread to suspend because of another thread's suspension.
- An example of a nonblocking mechanism is the Compare-And-Swap (CAS)

Nonblocking can also be problematic:

- **HARD TO DESIGN!**

```
bool CAS(int * array, int index, int old, int new) {  
    //ENTER ATOMIC  
    if (array[index] == old) {  
        array[index] = new;  
        return true;  
    }  
    return false;  
    //END ATOMIC  
}  
  
void function1(int value) {  
    CAS(A, 1, expected_old, value);  
    return;  
}
```

Transactions

```
void main() {  
    with_transaction {  
        //all instructions in this scope are part of the transaction  
        function1();  
        function2();  
    }  
    return;  
}
```

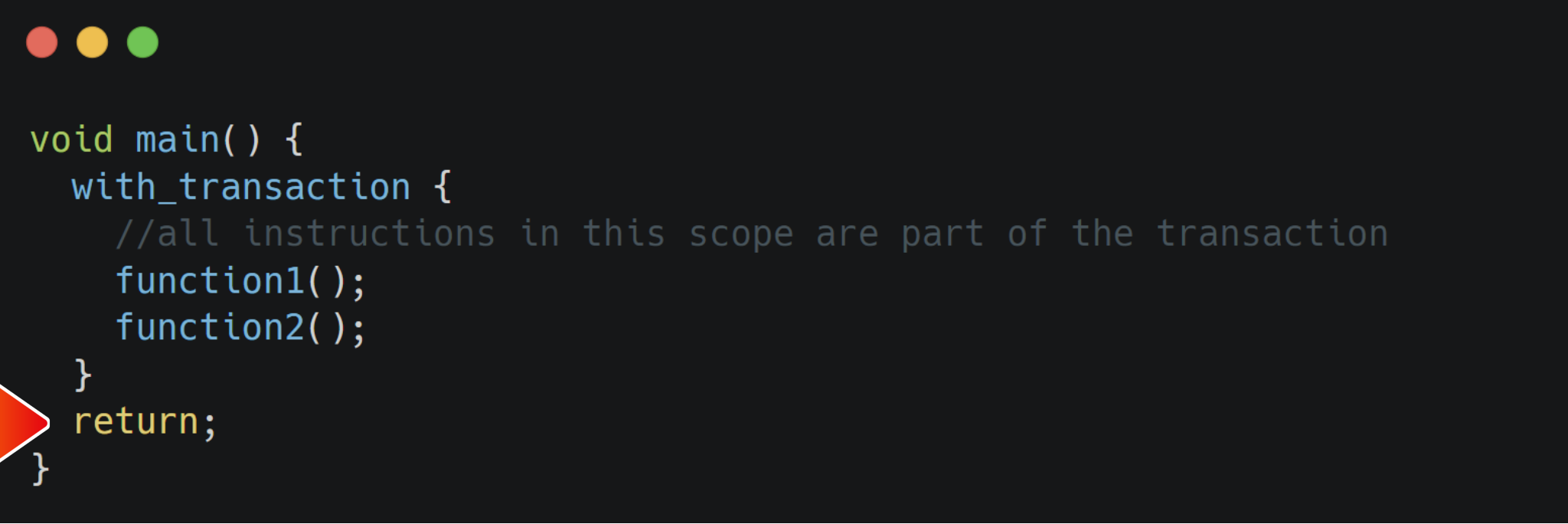
- Set of instructions that perform work if and only if no **conflict** is present
- A **conflict** is when multiple transactions or threads attempt to access the same block of transactional memory at once.
- Transactions can:
 - *Commit* – upon “making it through,” the work is confirmed to be done correctly
 - *Abort* – upon a conflict, the transaction will be reverted: none of its work will be done, and it can be restarted

Transactions make concurrent programming easy for developers!


Transactions (cont.)

```
void main() {  
    with_transaction {  
        //all instructions in this scope are part of the transaction  
        function1();  
        function2();  
    }  
    return;  
}
```

Transactions (cont.)



```
void main() {  
    with_transaction {  
        //all instructions in this scope are part of the transaction  
        function1();  
        function2();  
    }  
    return;  
}
```

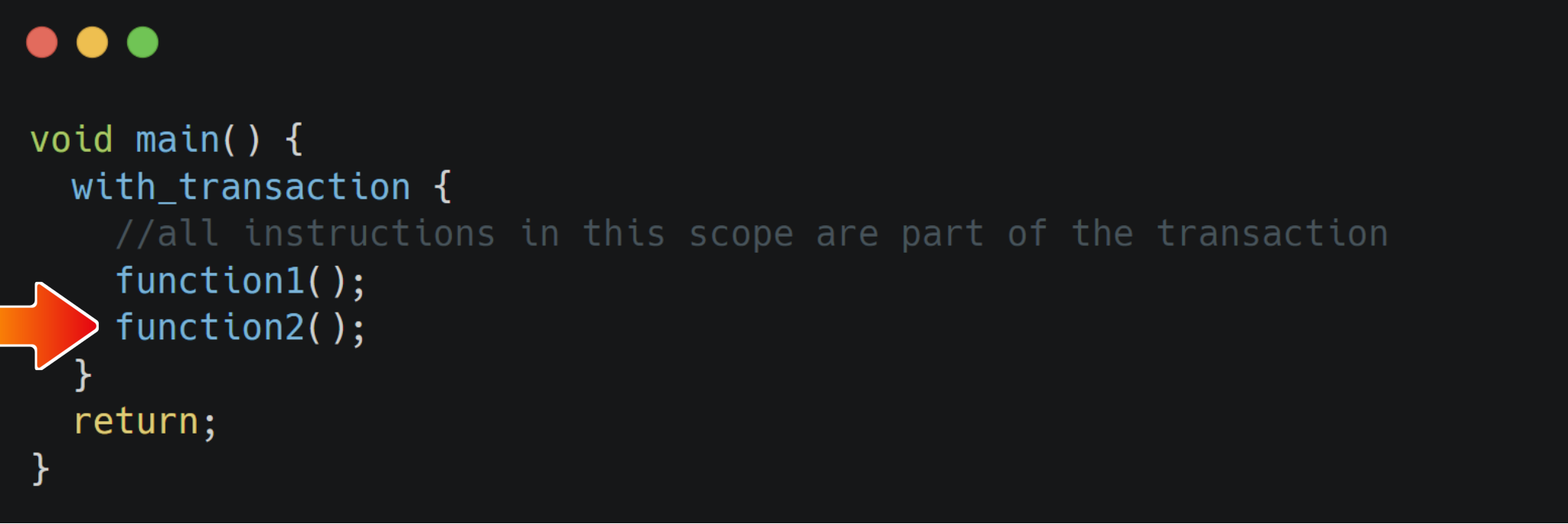


Transactions (cont.)

```
void main() {  
  with_transaction {  
    //all instructions in this scope are part of the transaction  
    function1();  
    function2();  
  }  
  return; MADE IT OUT OF TRANSACTION, WE CAN COMMIT  
}
```

In this example, the work done by function1 and function2 has taken effect in memory.

Transactions (cont.)



```
void main() {  
    with_transaction {  
        //all instructions in this scope are part of the transaction  
        function1();  
        function2();  
    }  
    return;  
}
```

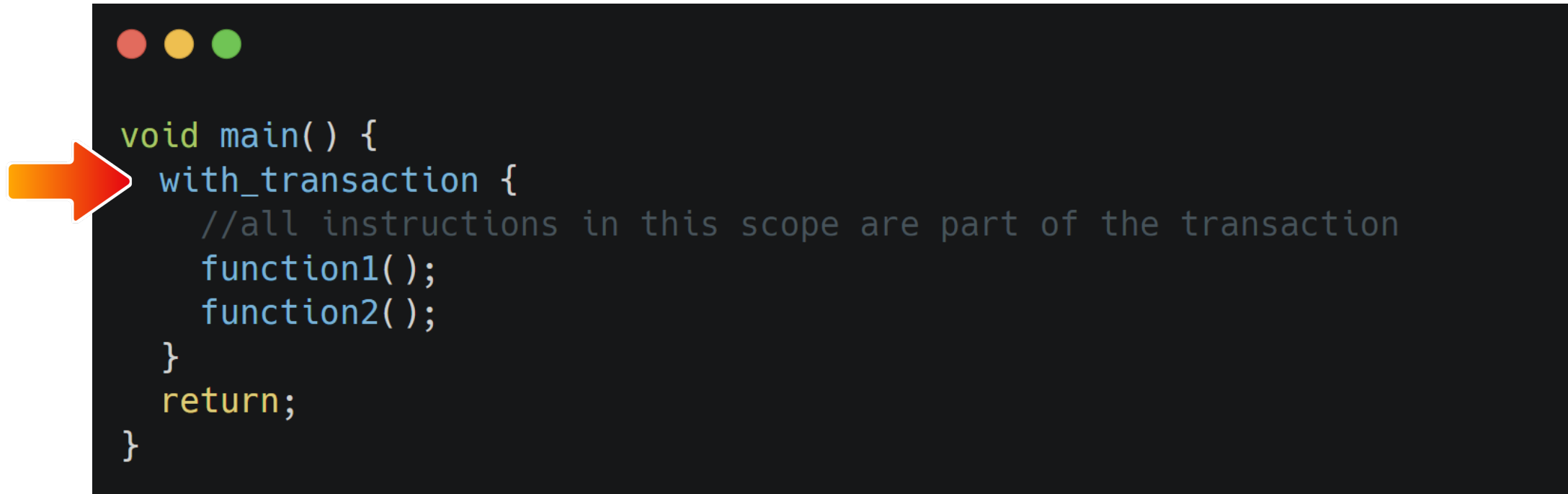
Transactions (cont.)

```
void main() {  
    with_transaction {  
        //all instructions in this scope are part of the transaction  
        function1();  
        function2();  
    }  
    return;  
}
```



INTERRUPTION – WE MUST **ABORT** AND **ROLL BACK**

Transactions (cont.)



```
void main() {  
  with_transaction {  
    //all instructions in this scope are part of the transaction  
    function1();  
    function2();  
  }  
  return;  
}
```

In this example, the work done by function1 and function2 has **NOT** taken effect in memory.

Transactional Memory

- Shared memory based on transactions to manage concurrency
- Allows for high-level abstraction rather than low-level synchronization

Transactional memory can still be problematic:

- Transactions can deadlock or find themselves starved of resources
- Transactions can livelock, endlessly aborting and restarting

Preventing these issues can get complicated (timestamping, probabilistic backoff, pessimistic/optimistic control, etc.)!

What Would be Nice

The goal is a transactional memory structure and algorithm that:

- cannot deadlock
- cannot livelock
- always makes forward progress (always gets closer to a *commit*)
- is deterministic (same behavior every time)
- is easy to reason about

Idea #1 – The Ownership Array

- Owner Array A : global array of locks (mutexes)
 - Every transactional memory location will be mapped to a single lock, but locks probably map to more than one memory location
 - All locks support the following instructions:
 - Acquire(lock): Try to hold the lock, block until it is available
 - Try_Acquire(lock): Try to hold the lock, and return true or false for a success or failure
 - Release(lock): Release the lock
- Owner function h : function that does the above-mentioned mapping
 - Known globally (by all transactions)
 - Probably a hash function
 - If M represents all transactional memory, then $h(m)$ is in A for all m in M .

Idea #2 : Local Transaction States

- Each transaction will keep a set L of all the locks it currently has acquired
- Each transaction will also keep state so that it can be rolled-back
 - Some transactions are *irrevocable*, but this is ok! Details later

The Formal Algorithm

```
SAFE-ACCESS( $x, L$ )
1  if  $h(x) \in L$ 
2     // do nothing
3  else
4      $M = \{i \in L : i > h(x)\}$ 
5      $L = L \cup \{h(x)\}$ 
6     if  $M == \emptyset$ 
7         ACQUIRE( $lock[h(x)]$ ) // blocking
8     elseif TRY-ACQUIRE( $lock[h(x)]$ ) // nonblocking
9         // do nothing
10    else
11        roll back transaction state (without releasing locks)
12        for  $i \in M$ 
13            RELEASE( $lock[i]$ )
14            ACQUIRE( $lock[h(x)]$ ) // blocking
15        for all  $i \in M$  in increasing order
16            ACQUIRE( $lock[i]$ ) // blocking
17        restart transaction // does not return
18    access location  $x$ 
```

The Algorithm, in Words

- When trying to access memory, first try to acquire its lock x .
 - If you already have it or immediately get it, obviously just continue.
- If someone else is currently holding x , do the following:
 - For all locks y in L , if $h(y) > h(x)$, release it (but don't forget it!).
 - Block on x
 - Re-acquire all locks previously dropped, in sorted order, blocking if conflicted
 - Restart transaction

The Algorithm, in Words

- When trying to access memory, first try to acquire its lock x .
- If someone else is currently holding x , do the following:
 - Abort (without releasing any locks in L)
 - For any lock y in L , if $h(y) > h(x)$, release it.
 - Block on x
 - Re-acquire all locks previously dropped, in sorted order, blocking if conflicted
 - Restart transaction

Transactions abort themselves here, rather than being aborted at random by conflict. This simplifies transaction implementation.

At every restart, at least one more lock is added to L so there must be a finite number of restarts

Lemma: *Transactions do not Deadlock*

- A transaction only blocks on a lock if that lock has a higher h value than any other lock it holds.
- There is thus no cycle of blocking.

Lemma: *Every Transaction Makes Forward Progress*

- Every time a transaction restarts, it will hold **at least one more** lock than it did before. If there is a finite number of locks needed per transaction, then there is a finite number of restarts required to acquire all necessary locks.
- That is, L_{prev} is a strict subset of L_{next}
- Before a restart:
 - All greater locks are dropped.
 - Original conflict is obtained.
 - All previously dropped locks are re-obtained.
 - The lesser locks were never dropped.


Not So Fast: Real-World Complications

- How big should the ownership array be?
 - Want to reduce chances of owner function collisions (birthday paradox!)
 - Don't want to take up too much space
 - **Experiments have been done empirically, but theoretical analysis remains an open problem**
- Not all transactions are reversible.
 - If the algorithm knows all memory locations needed to be accessed in an *irrevocable* transaction, then it can ensure all locks are held before ever starting and ensure a *commit*.

More Related Open Problems

- Ownership array might be able to be cached for performance, owner function writing addresses to cache lines – **empirical evidence needed**
- Compilers might be able to optimize for groups of locks acquired in transactions
- Lock ordering might be dynamic rather than static, which might enable a faster algorithm

Questions?



```
void main() {  
    with_transaction {  
        askForQuestions();  
        answerQuestions();  
    }  
    thankTheAudience();  
    endPresentation();  
    return;  
}
```