Lecture 13: Trace and Replay of Task Parallel Programs

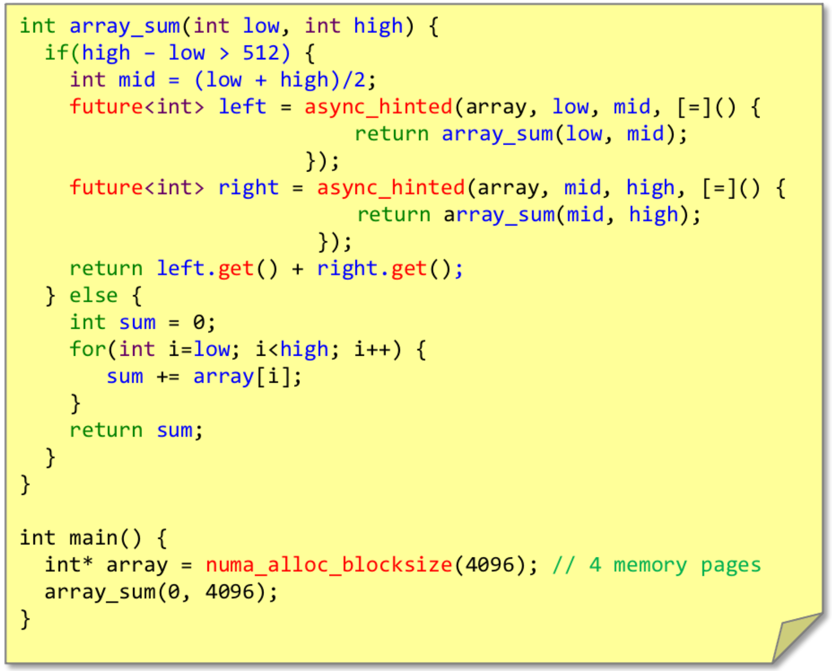
Vivek Kumar

Computer Science and Engineering

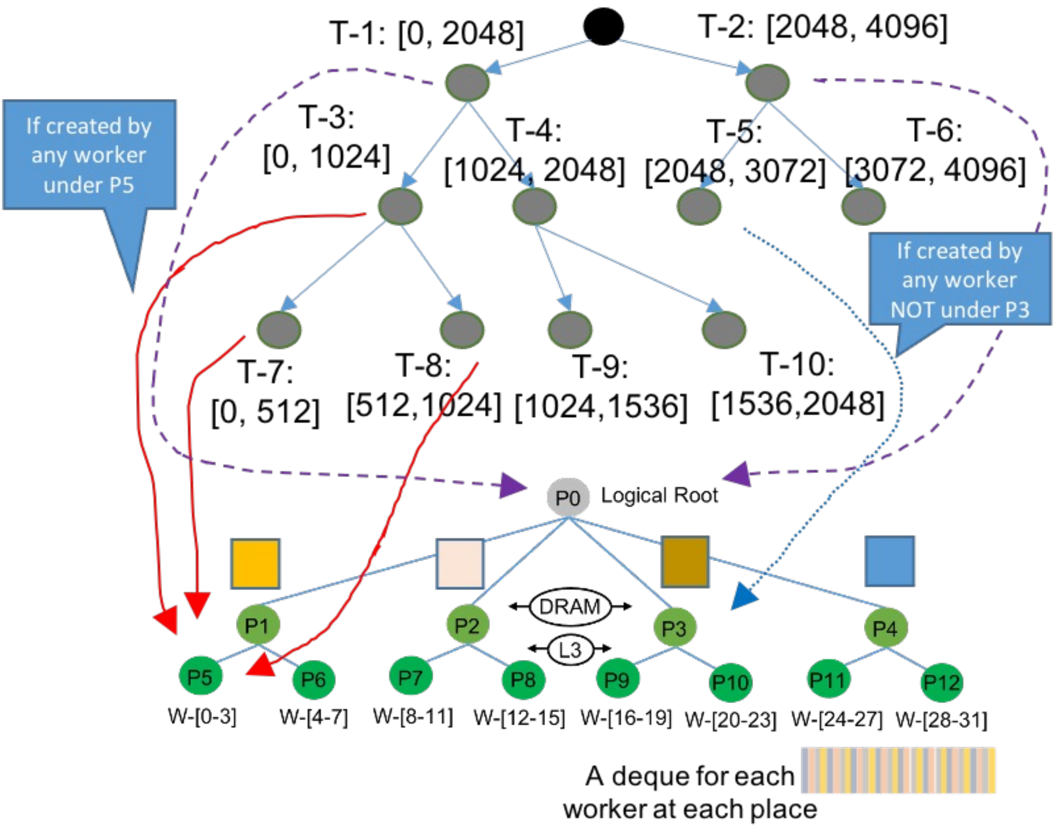
IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture (Recap)



```
int array_sum(int low, int high) {
  if(high - low > 512) {
    int mid = (low + high)/2;
    future<int> left = async_hinted(array, low, mid, [=]() {
                          return array_sum(low, mid);
                       });
    future<int> right = async_hinted(array, mid, high, [=]() {
                          return array_sum(mid, high);
                        });
    return left.get() + right.get();
  } else {
    int sum = 0;
    for(int i=low; i<high; i++) {
       sum += array[i];
    }
    return sum;
  }
}

int main() {
  int* array = numa_alloc_blocksize(4096); // 4 memory pages
  array_sum(0, 4096);
}
```



- Three rules to **push** a task for preserving locality
  1. Task whose data spans to multiple NUMA nodes must be pushed to at root place
  2. Task whose data is not local to the worker creating the task, then the task should be pushed at appropriate remote DRAM place
  3. Worker creating a task with the local data must push it to its cache place

- Four rules to **steal** a task for preserving locality
  1. Attempt to steal from local leaf place
  2. Attempt to steal from the local DRAM place
  3. Attempt to steal from sibling cache places under same DRAM
  4. Attempt to steal from the root place

Today’s Class

- Trace and replay of asynchronous tasks
- Quiz-2

Runtime Profiling

- It is a technique for understanding the behavior of the parallel runtime / program during the execution
  - High-level details
    - Total number of tasks created
    - Total number of tasks stolen
    - Total number of tasks migrated across NUMA domains
    - Total number of failed steals
    - Task execution time, etc.
  - Low-level details
    - Tracing the program execution (computation graph)
    - Computation type (compute-bound / memory-bound)
    - Power usage
    - Instructions retired for each task
    - Total CPU stalls, etc.

Runtime Profiling

- It is a technique for understanding the behavior of the parallel runtime / program during the execution
  - High-level details
    - Total number of tasks created
    - Total number of tasks stolen
    - Total number of tasks migrated across NUMA domains
    - Total number of failed steals
    - Task execution time, etc.
  - Low-level details
    - Tracing the program execution (computation graph)
    - Computation type (compute-bound / memory-bound)
    - Power usage
    - Instructions retired for each task
    - Total CPU stalls, etc.

Easily obtained using thread local counters

Runtime Profiling

- It is a technique for understanding the behavior of the parallel runtime / program during the execution
  - High-level details
    - Total number of tasks created
    - Total number of tasks stolen
    - Total number of tasks migrated across NUMA domains
    - Total number of failed steals
    - Task execution time, etc.
  - Low-level details
    - Tracing the program execution (computation graph)
    - Computation type (compute-bound / memory-bound)
    - Power usage
    - Instructions retired for each task
    - Total CPU stalls, etc.

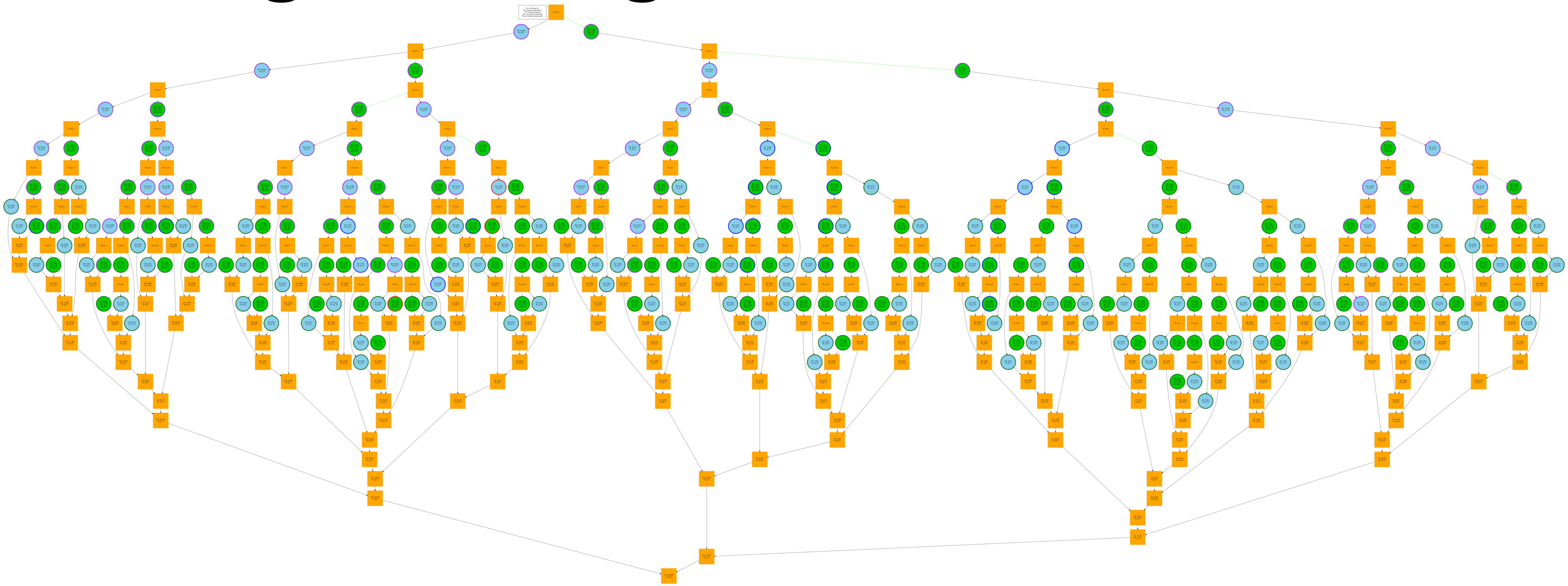
Requires special support

Runtime Profiling

- It is a technique for understanding the behavior of the parallel runtime / program during the execution
  - High-level details
    - Total number of tasks created
    - Total number of tasks stolen
    - Total number of tasks migrated across NUMA domains
    - Total number of failed steals
    - Task execution time, etc.
  - Low-level details
    - Tracing the program execution (computation graph)
    - Computation type (compute-bound / memory-bound)
    - Power usage
    - Instructions retired for each task
    - Total CPU stalls, etc.

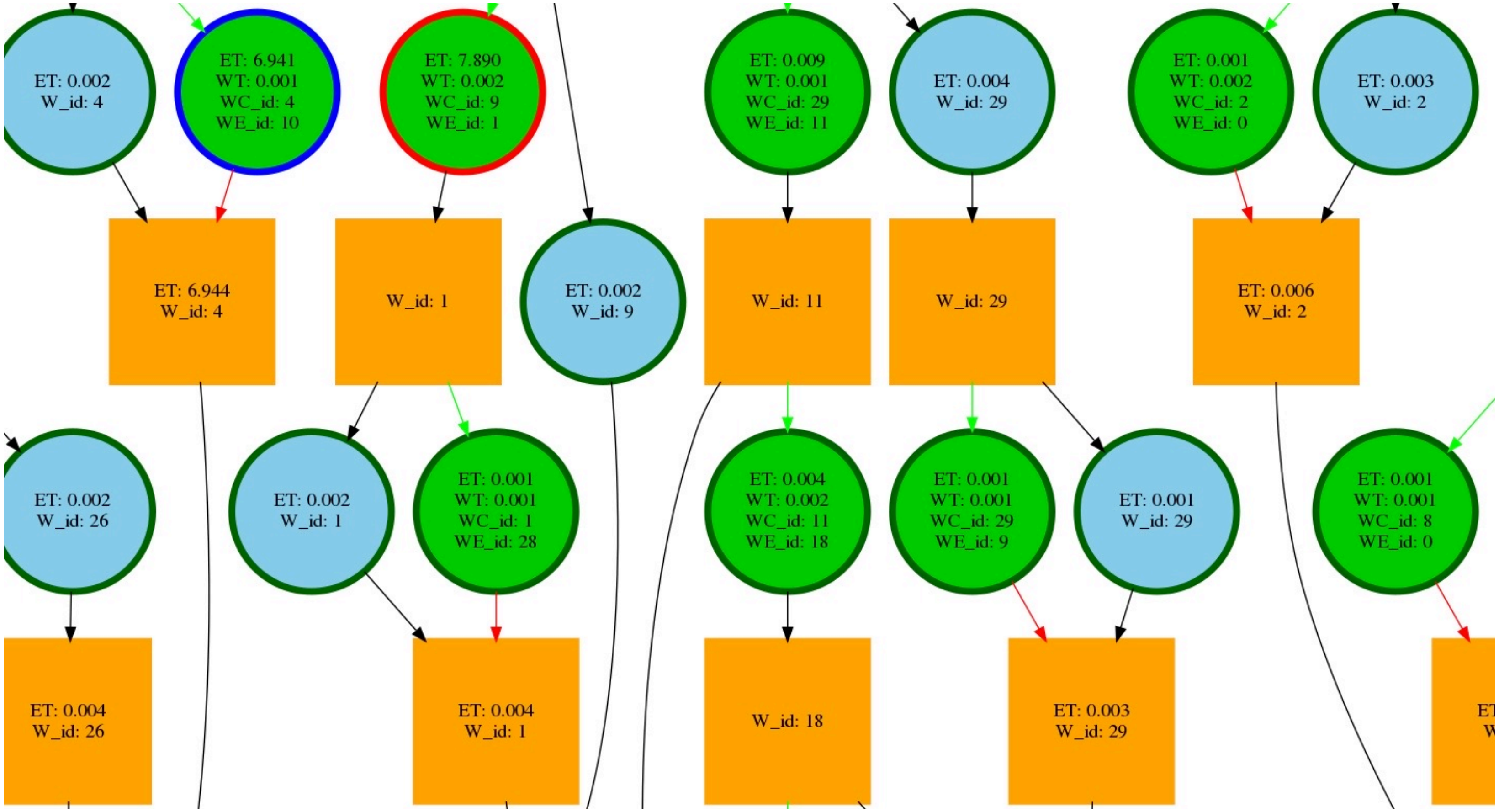
Overall goal: profiling with minimal overheads!

Tracing the Program Execution



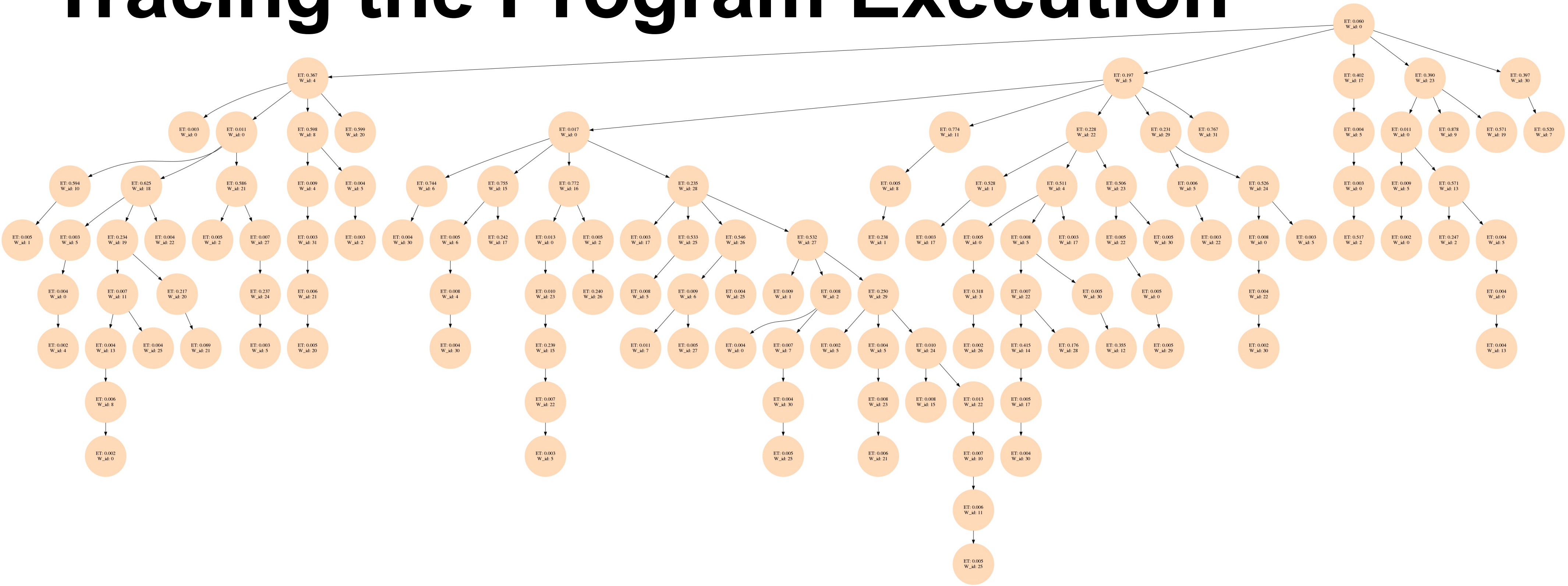
- Recursive task parallel Fibonacci number calculation (N=20, threshold=10)
  - Graph will be too big to fit in the slide for large N, hence small value chosen
- Blue node represents fib(n-2), green node represents async fib(n-1), and orange rectangular boxes are the synchronization scope for tasks created in that scope

Tracing the Program Execution



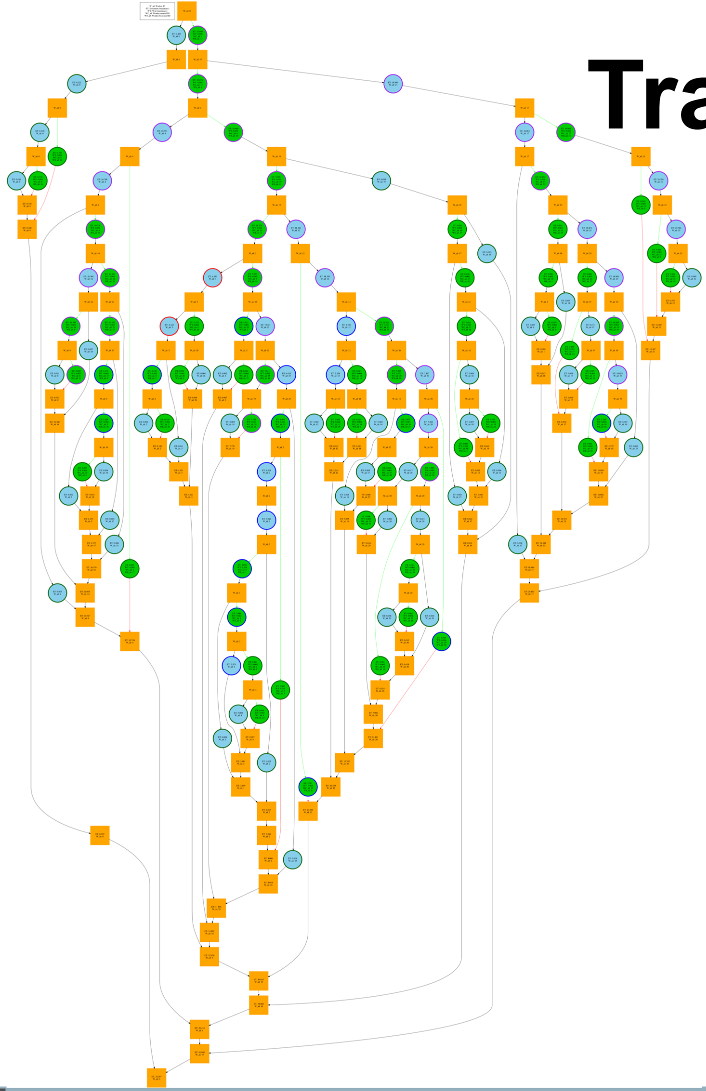
- Enlarging the nodes from the computation graph of Fibonacci
- ET=Execution time
- WT: Wait time
- WC_id: Id of worker who created this task
- WE_id: Id of the worker who executed this task (due to stealing)

Tracing the Program Execution



- Steal tree of the same Fibonacci execution using 32 workers

Tracing the Program Execution



- Recursive task parallel QuickSort
  - Array size 1024, and chunk threshold 32
- Blue node represents sequential task, green node represents an async, and orange rectangular boxes are the synchronization scope for tasks created in that scope

Tracing the Program Execution

- Advantages
  - Offline analysis can help in reducing/increasing the task threshold if its not done automatically by the runtime
  - Reducing task management overheads in iterative applications
    - How?

Iterative Averaging

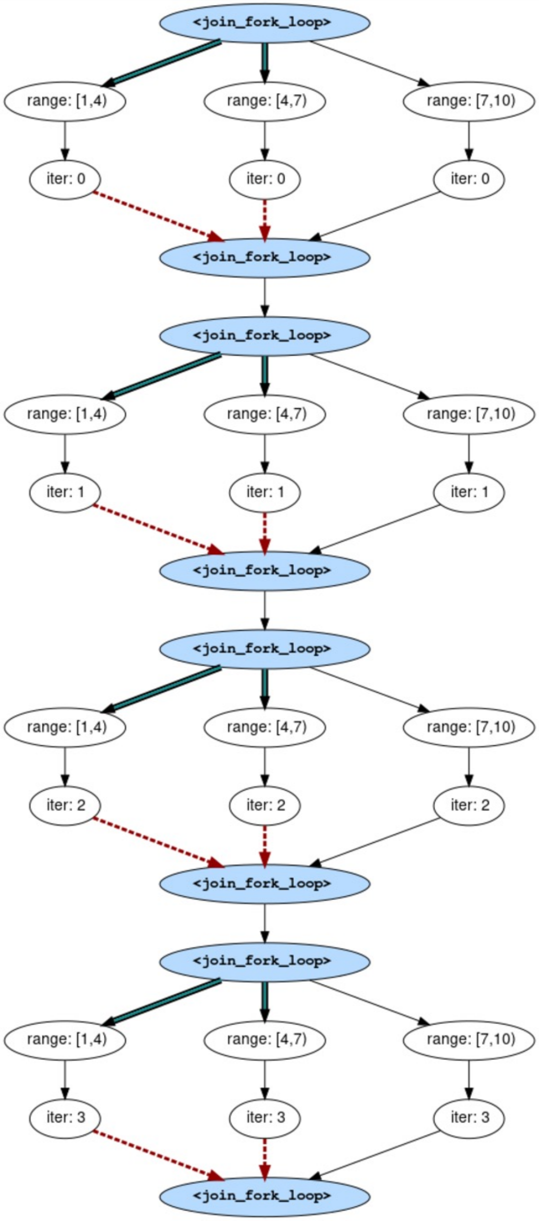
```
double A[SIZE+2], A_shadow[SIZE+2];

void recurse(int low, int high) {
  if((high – low) > THRESHOLD) {
    int mid = (high+low)/2;
    future<void> f1 = async([=]() { recurse(low, mid); });
    recurse(mid, high);
    f1.get();
  } else {
    for(int j=low; j<high; j++) {
      A_shadow[j] = (A[j-1] + A[j+1])/2.0;
    }
  }
}

void compute(int MAX_ITERS) {
  for(int i=0; i<MAX_ITERS; i++) {
    recurse(1, SIZE+1);
    double* temp = A_shadow;
    A_shadow = A;
    A = temp;
  }
}
```

- Initialize a one-dimensional array of (SIZE+2) double's with boundary conditions, A[0] = 0 and A[SIZE+1] = 1
- In each iteration, each interior element A[j] in 1…SIZE is replaced by the average of its left and right neighbours
  - ○ Two separate arrays are used in each iteration, one for old values and the other for the new values
- After a sufficient number of iterations, we expect each element of the array to converge to A[j] = (A[j-1]+A[j+1])/2, for all j in 1…SIZE

Details: https://classes.engineering.wustl.edu/cse231/core/index.php/Iterative_Averaging

Iterative Averaging

- Observations
  - Exact same computation graph in each for loop iteration in compute()
- Optimization
  - Improved locality if each workers executes the exact same set of tasks in each for loop iteration of compute
  - Random work-stealing
    - It would result in poor locality as each worker could get different set of tasks in each for loop iteration of compute
  - Trace/Replay for improving locality
    - Trace (i.e., record) the tasks executed by each worker during the first iteration of for loop inside compute
    - For the rest of iterations of the above for loop of compute, disable random work-stealing and use the information gathered during the Trace (i.e., record) phase to replay the exact set of tasks at each worker



Details: https://classes.engineering.wustl.edu/cse231/core/index.php/Iterative_Averaging

Tracing the Program Execution

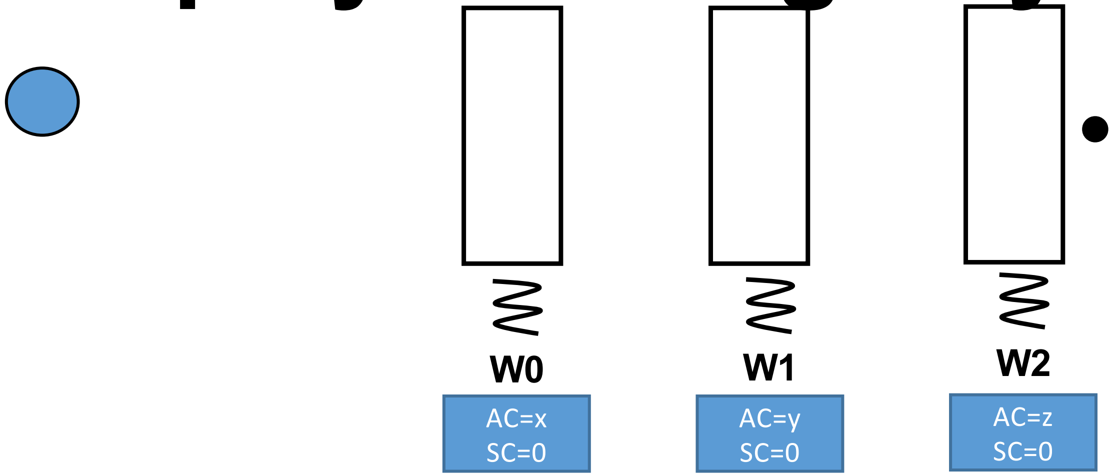
- Advantages
  - Offline analysis can help in reducing/increasing the task threshold if its not done automatically by the runtime
  - Reducing task management overheads in iterative applications
    - How?
  - Data-race detection
    - If there is NO path to connect between two nodes (i.e., they may execute in parallel), and if they perform read/write or write/write operation on a shared memory location then it's a data race
      - More on this later when we will cover data race detection (post mid-sem)
- Drawbacks
  - Recording details for each and every task will consume too much memory (e.g., millions of tasks in Fibonacci 40)
  - Profiling overheads as each worker has to do some extra work

Tracing the Program Execution

- Advantages
  - Offline analysis can help in reducing/increasing the task threshold if its not done automatically by the runtime
  - Reducing task management overheads in iterative applications
    - How?
  - Data-race detection
    - If there is NO path to connect between two nodes (i.e., they may execute in parallel), and if they perform read/write or write/writ shared memory location then it's a data race
      - More on this later when we will cover data race dete
- Drawbacks
  - Recording details for each and every task will consume too much memory (e.g., millions of tasks in Fibonacci 40)
  - Profiling overheads as each worker has to do some extra work

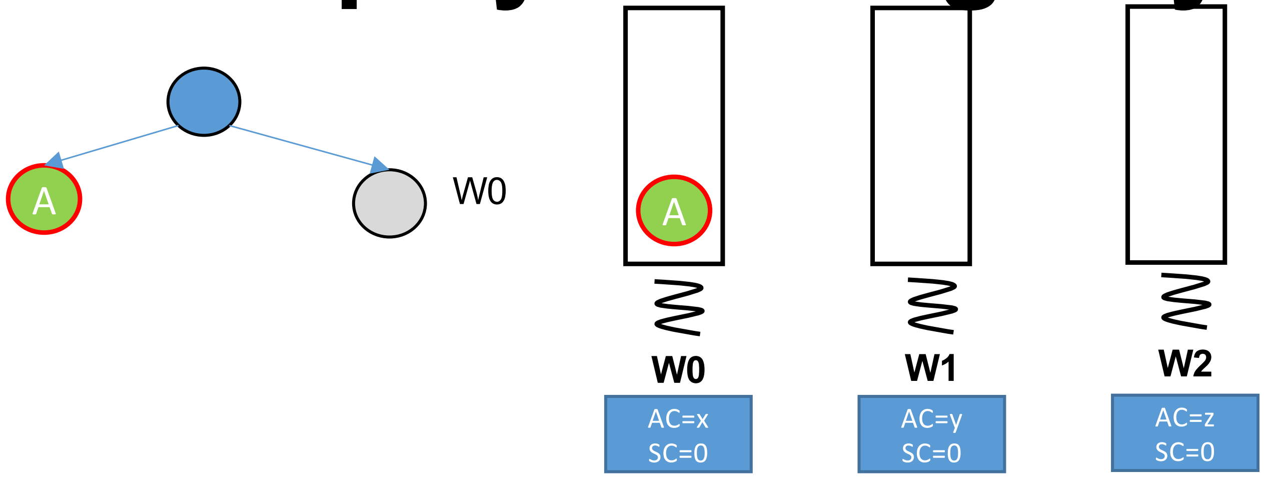
How to avoid these overheads?

Trace & Replay: Tracing Async



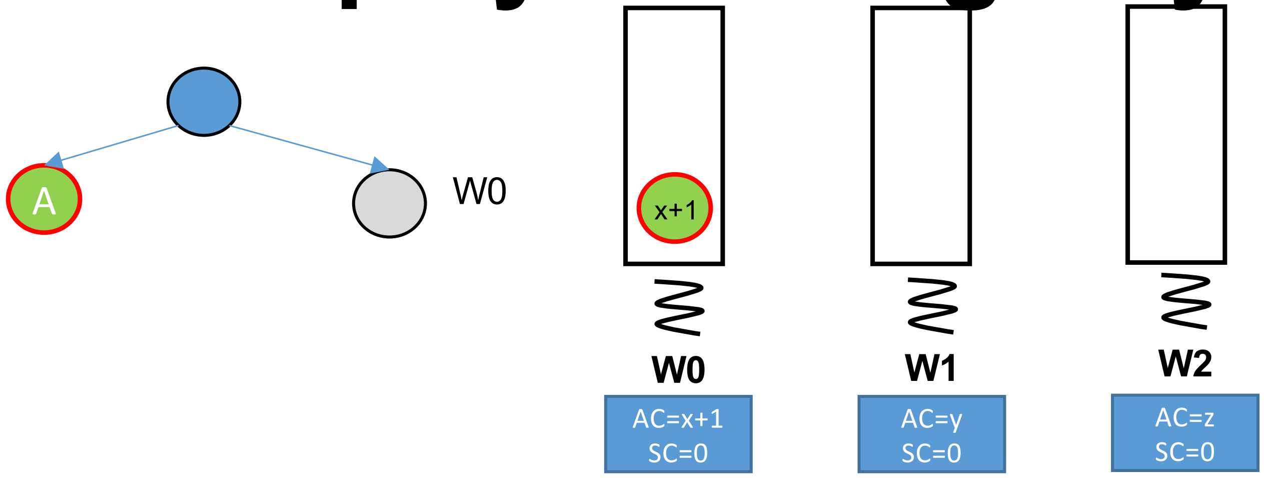
- Let there be three workers in a work-stealing based parallel runtime
  - Worker encountering an async will push that task into its deque, and would start working on the statement after the async
- Each worker has two counters
  - Async Counter (AC)
  - Each worker initializes its AC value = workerID * INT_MAX/numWorkers
  - Steal Counter (SC) initialized to zero

Trace & Replay: Tracing Async



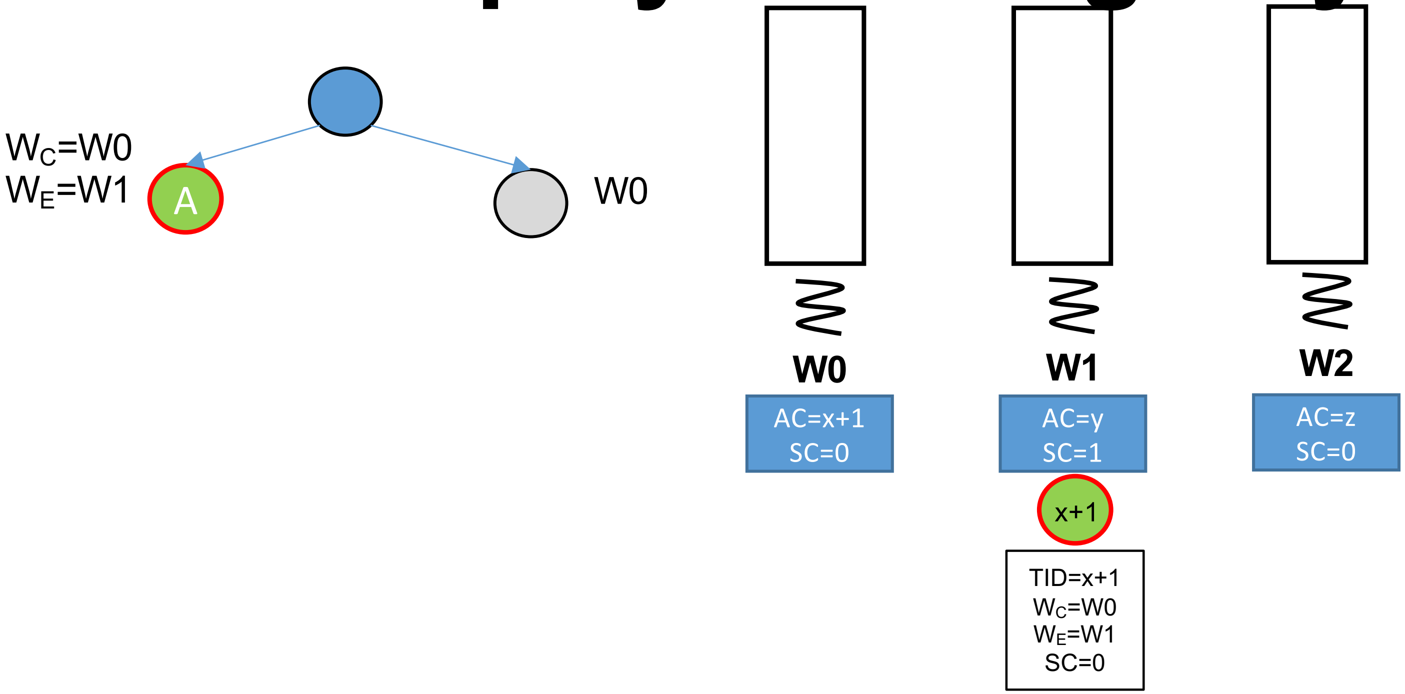
- Worker W0 starts a recursive task parallel application
- W0 creates an async A that is pushed into its deque

Trace & Replay: Tracing Async



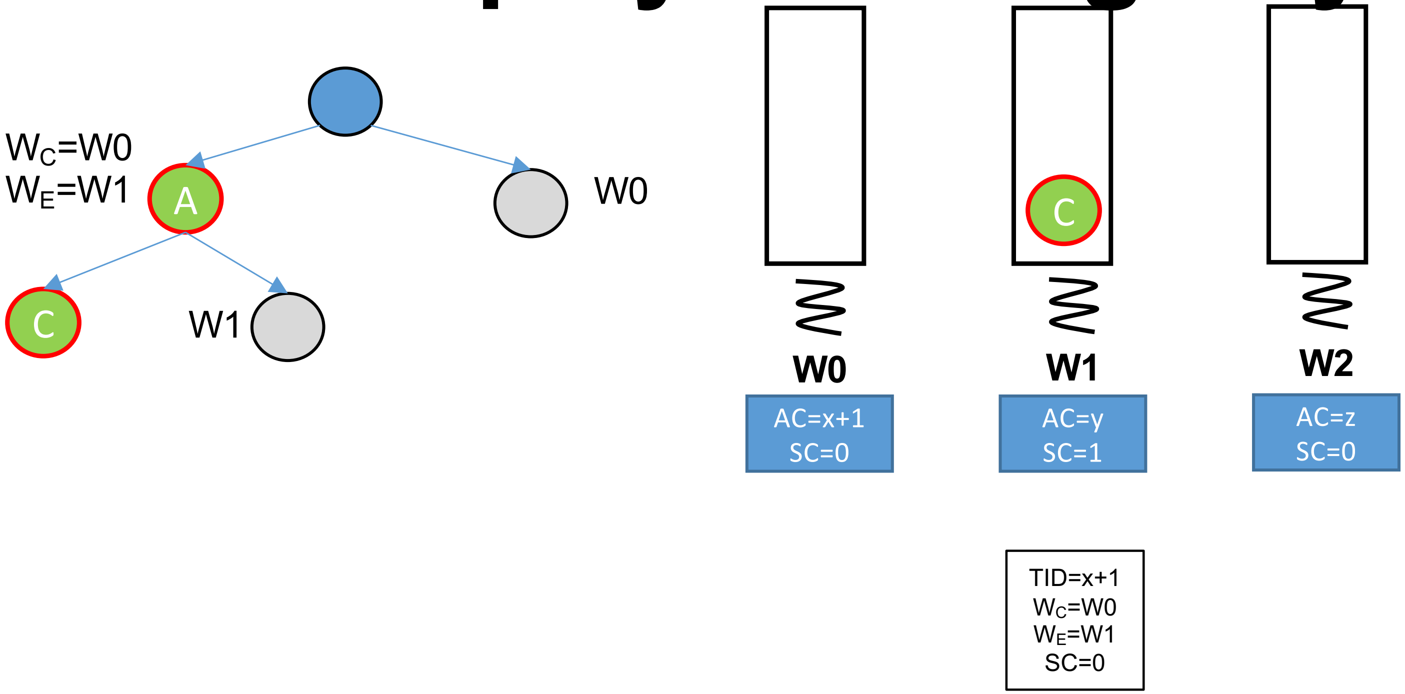
- AC at W0 is incremented and is assigned as the ID of the Task A **before** its pushed into W0's deque

Trace & Replay: Tracing Async



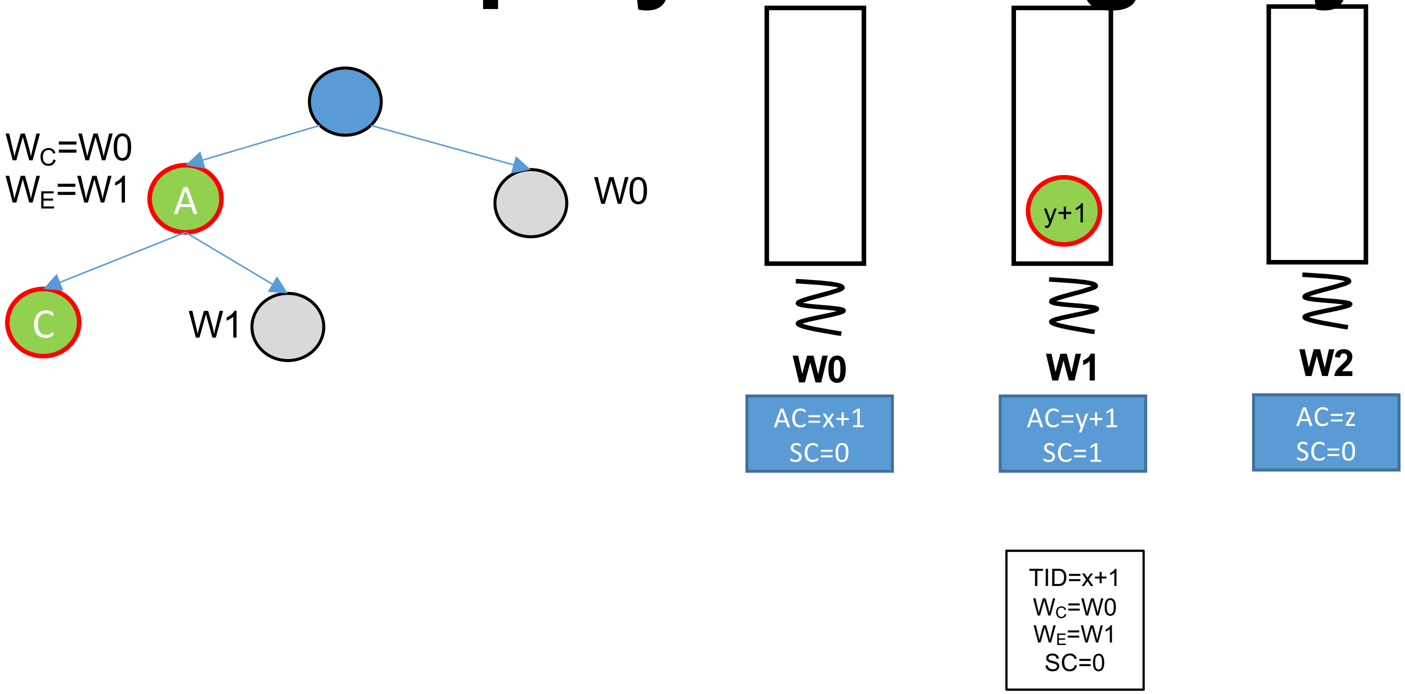
- W1 steals the task A from W0
- It appends a node in a private linked list containing info about this stolen task A
  - ID of the task (TID=x+1)
  - Worker who created this task (W_C=W0)
  - Worker who executed (stolen) this async (W_E=W1)
  - Current Steal Counter at W1 (SC=0)
- W1 then increment its Steal Counter (SC) before executing this stolen task

Trace & Replay: Tracing Async



- W1 creates an async C

Trace & Replay: Tracing Async



AC=x+1
SC=0

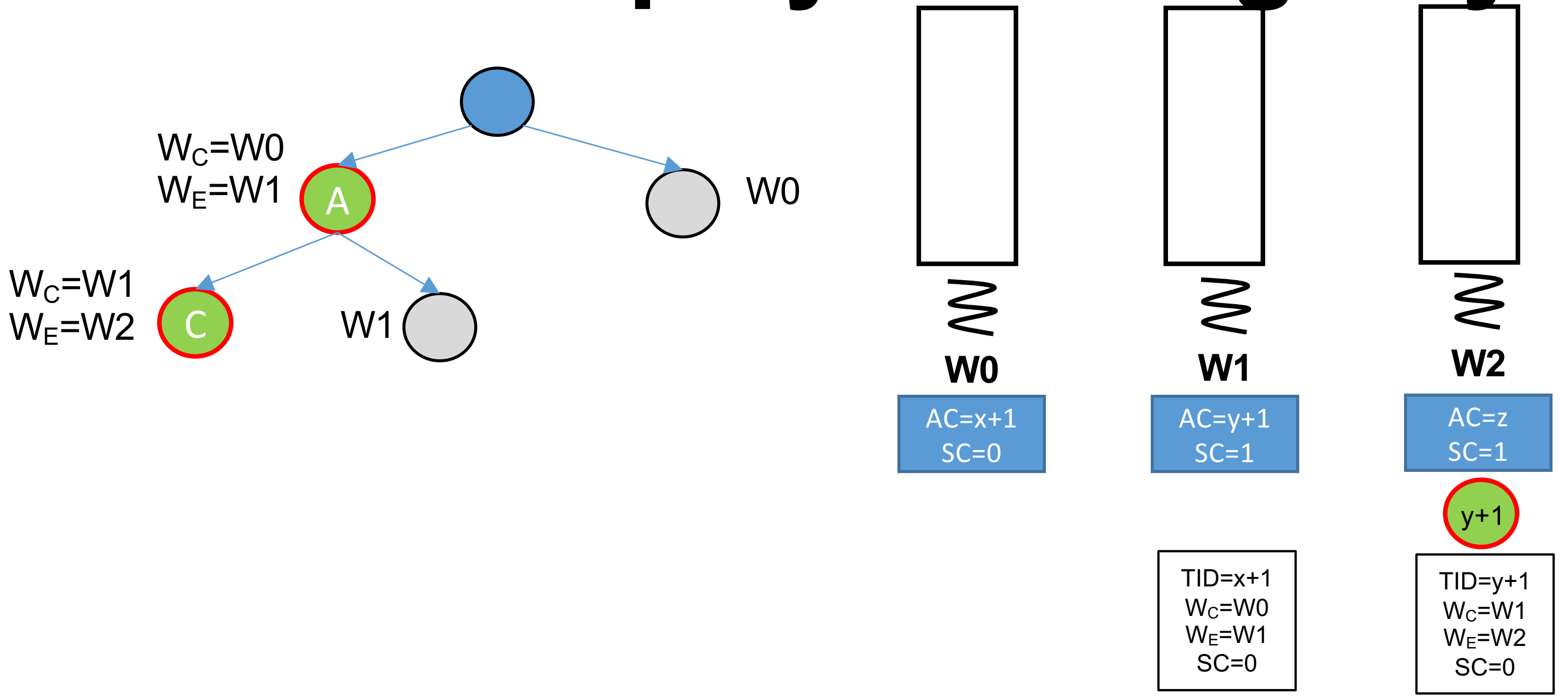
AC=y+1
SC=1

AC=z
SC=0

TID=x+1
W_C=W0
W_E=W1
SC=0

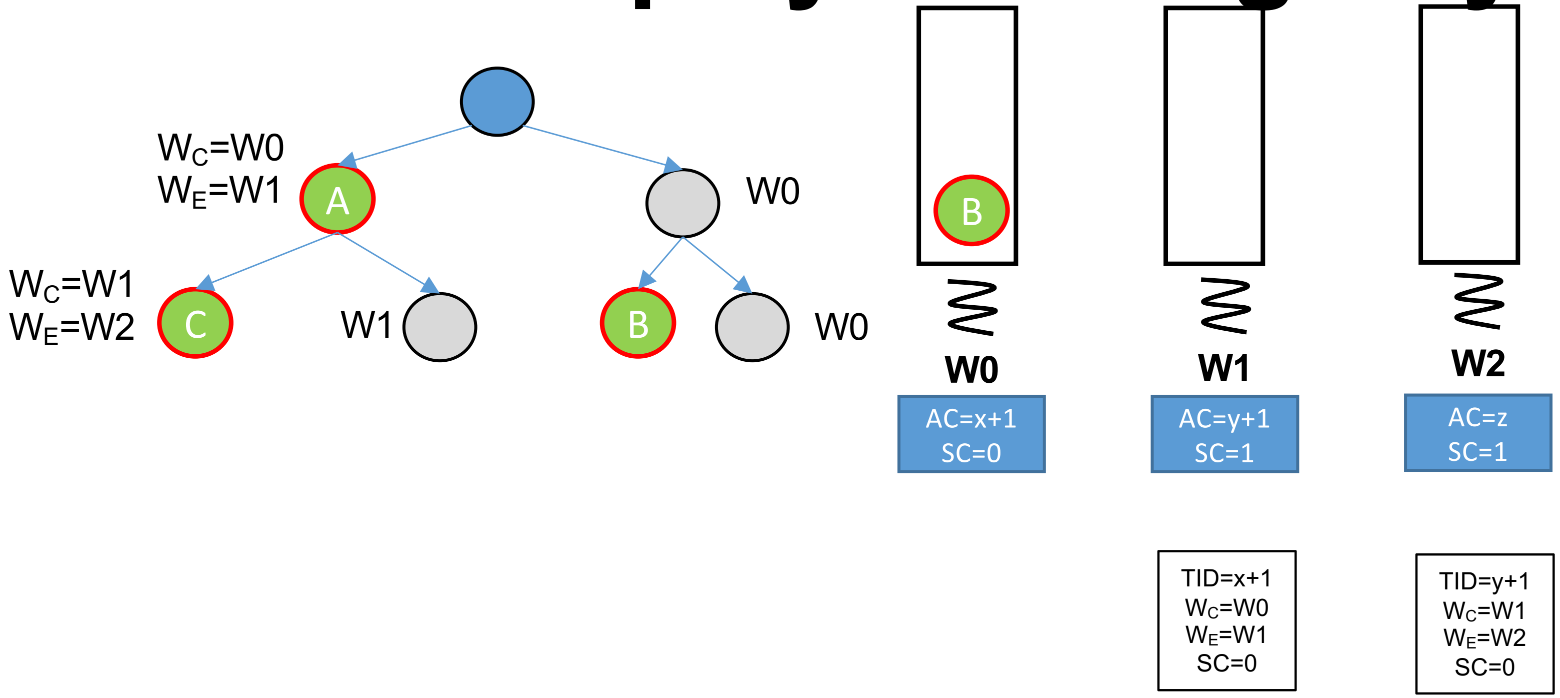
- AC at W1 is incremented and is assigned as the ID of the Task C **before** its pushed into W1’s deque

Trace & Replay: Tracing Async



- W2 steals the task C from W1
- It appends a node in a private linked list containing info about this stolen task C
    - ○ ID of the task (TID=y+1)
    - ○ Worker who created this task (W_C=W1)
    - ○ Worker who executed (stolen) this async (W_E=W2)
    - ○ Current Steal Counter at W2 (SC=0)
- W2 then increment its Steal Counter (SC) before executing this stolen task

Trace & Replay: Tracing Async



W_C=W0
W_E=W1

A

W0

W_C=W1
W_E=W2

C

W1

B

W0

B

W0

AC=x+1
SC=0

W1

AC=y+1
SC=1

TID=x+1
W_C=W0
W_E=W1
SC=0

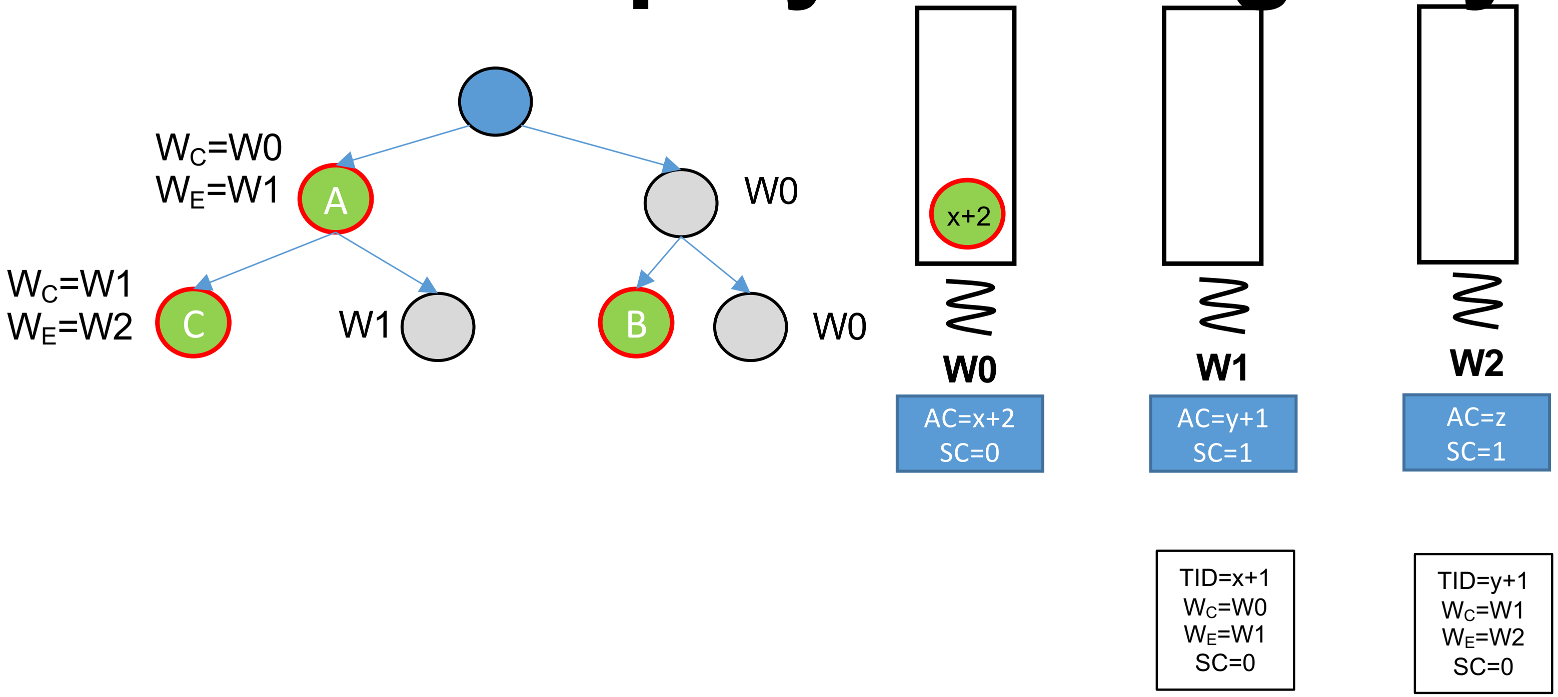
W2

AC=z
SC=1

TID=y+1
W_C=W1
W_E=W2
SC=0

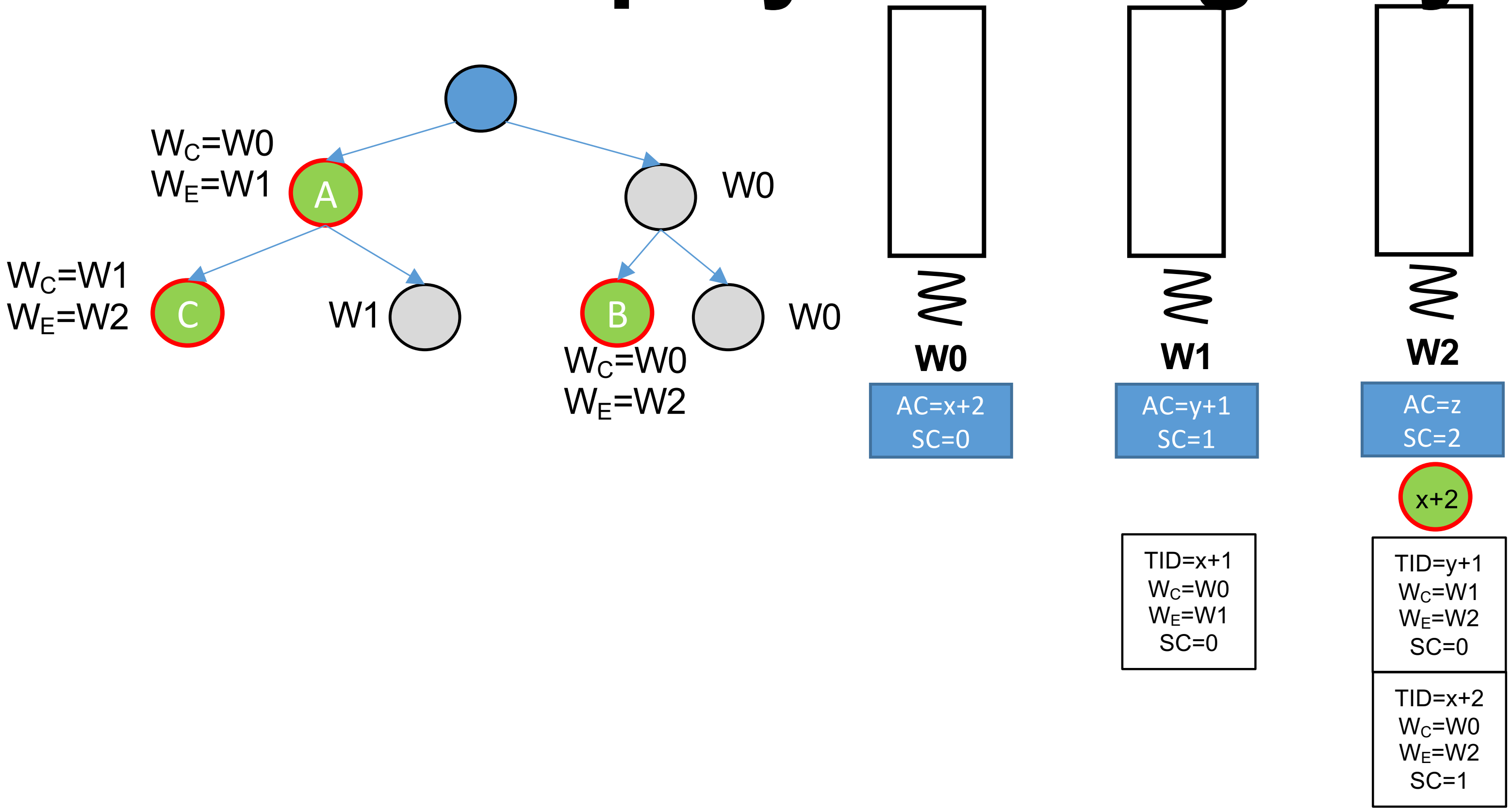
- W0 creates an async B

Trace & Replay: Tracing Async



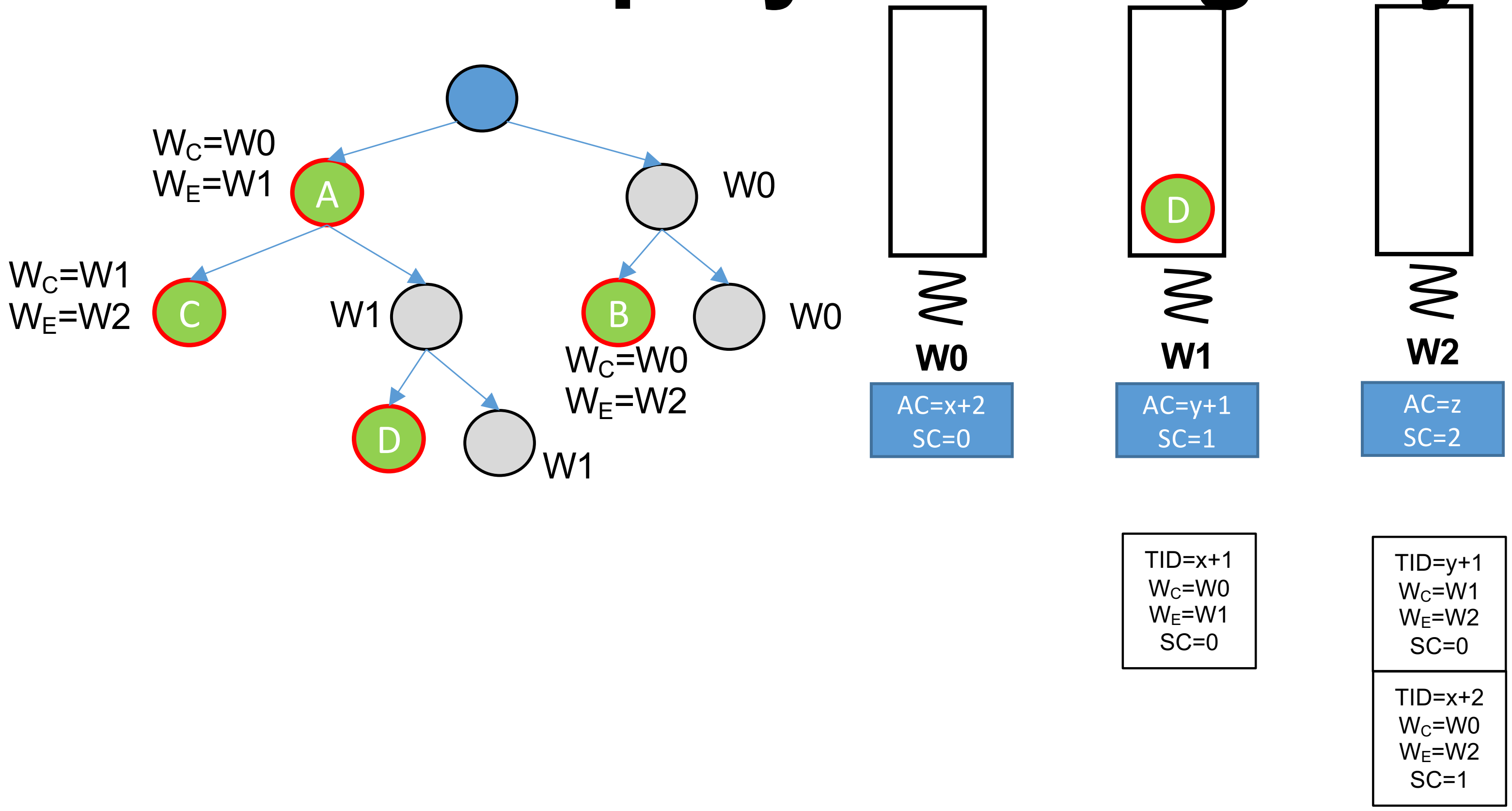
- AC at W0 is incremented and is assigned as the ID of the Task B **before** its pushed into W0’s deque

Trace & Replay: Tracing Async



- W2 steals the task B from W0
- It append a node in a private linked list containing info about this stolen task B
    - ○ ID of the task (TID=x+2)
    - ○ Worker who created this task (W_C=W0)
    - ○ Worker who executed (stolen) this async (W_E=W2)
    - ○ Current Steal Counter at W2 (SC=1)
- W2 then increment its Steal Counter (SC) before executing this stolen task

Trace & Replay: Tracing Async



W_C=W0
W_E=W1
A

W0

W_C=W1
W_E=W2
C

W1

B
W_C=W0
W_E=W2

W0

D

W1

D

W0
AC=x+2
SC=0

W1
AC=y+1
SC=1

W2
AC=z
SC=2

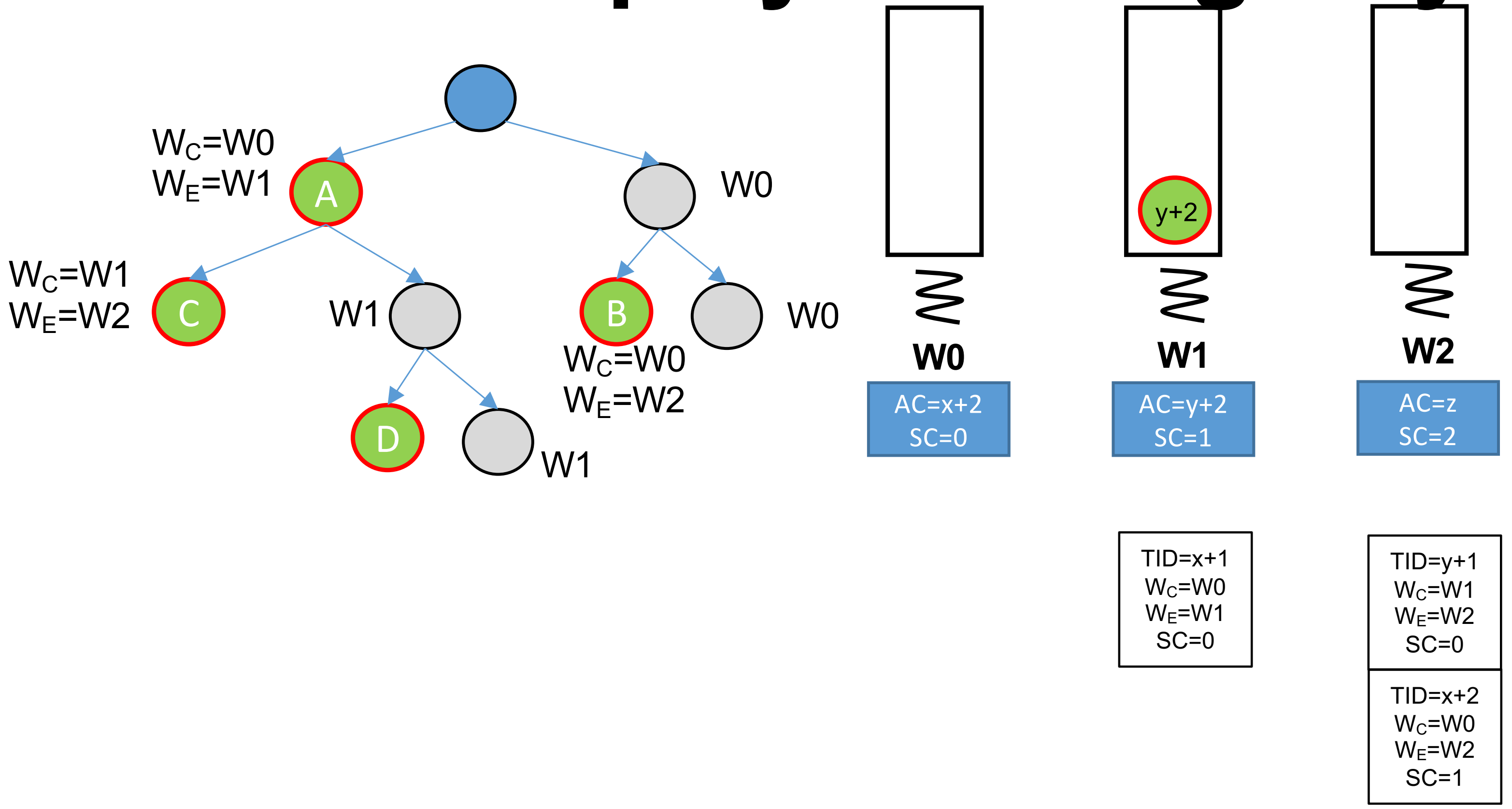
TID=x+1
W_C=W0
W_E=W1
SC=0

TID=y+1
W_C=W1
W_E=W2
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

- W1 creates an async D

Trace & Replay: Tracing Async



AC=x+2
SC=0

AC=y+2
SC=1

AC=z
SC=2

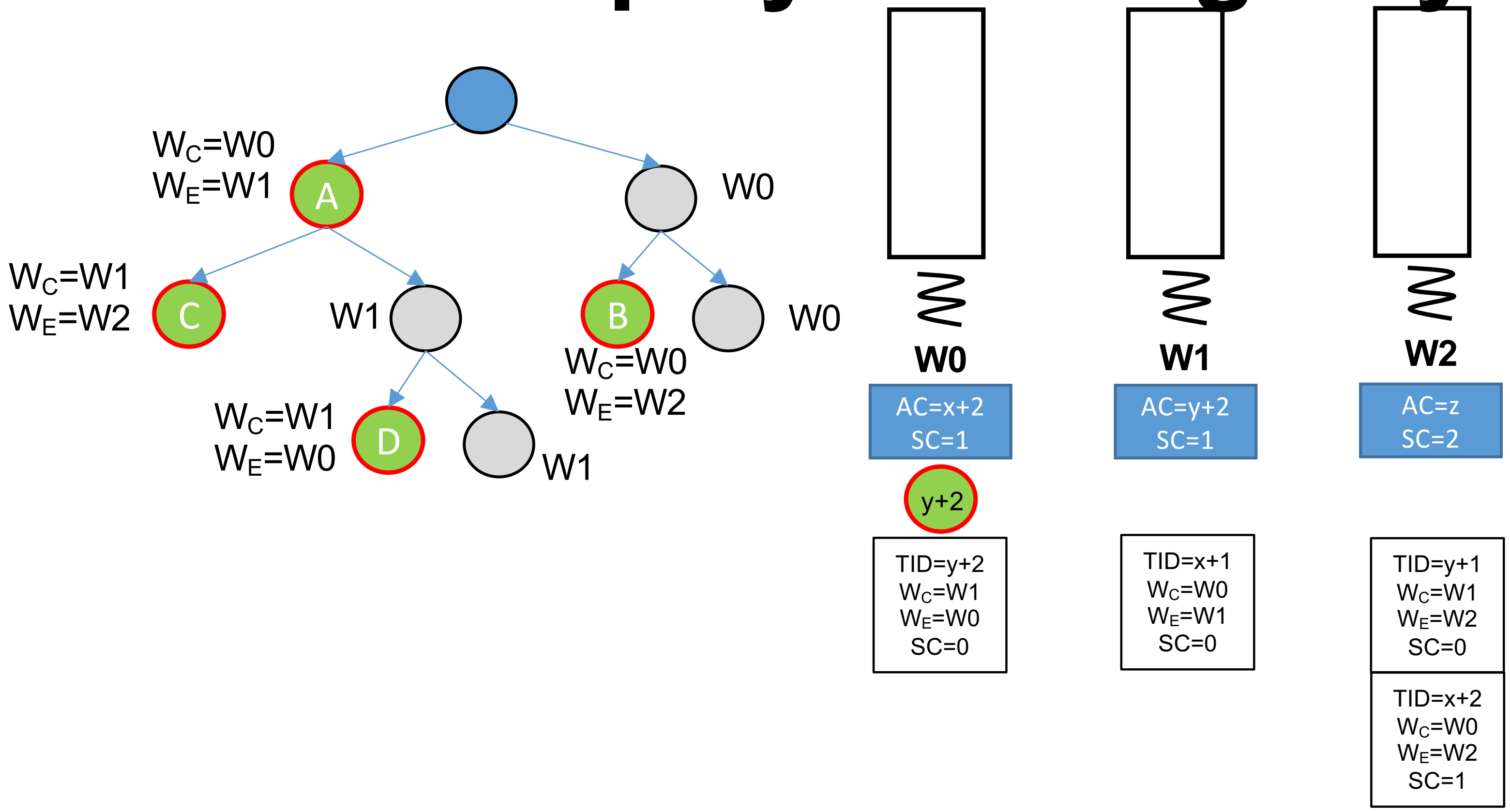
TID=x+1
W_C=W0
W_E=W1
SC=0

TID=y+1
W_C=W1
W_E=W2
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

- AC at W1 is incremented and is assigned as the ID of the Task D **before** its pushed into W1’s deque

Trace & Replay: Tracing Async



TID=y+2
W_C=W1
W_E=W0
SC=0

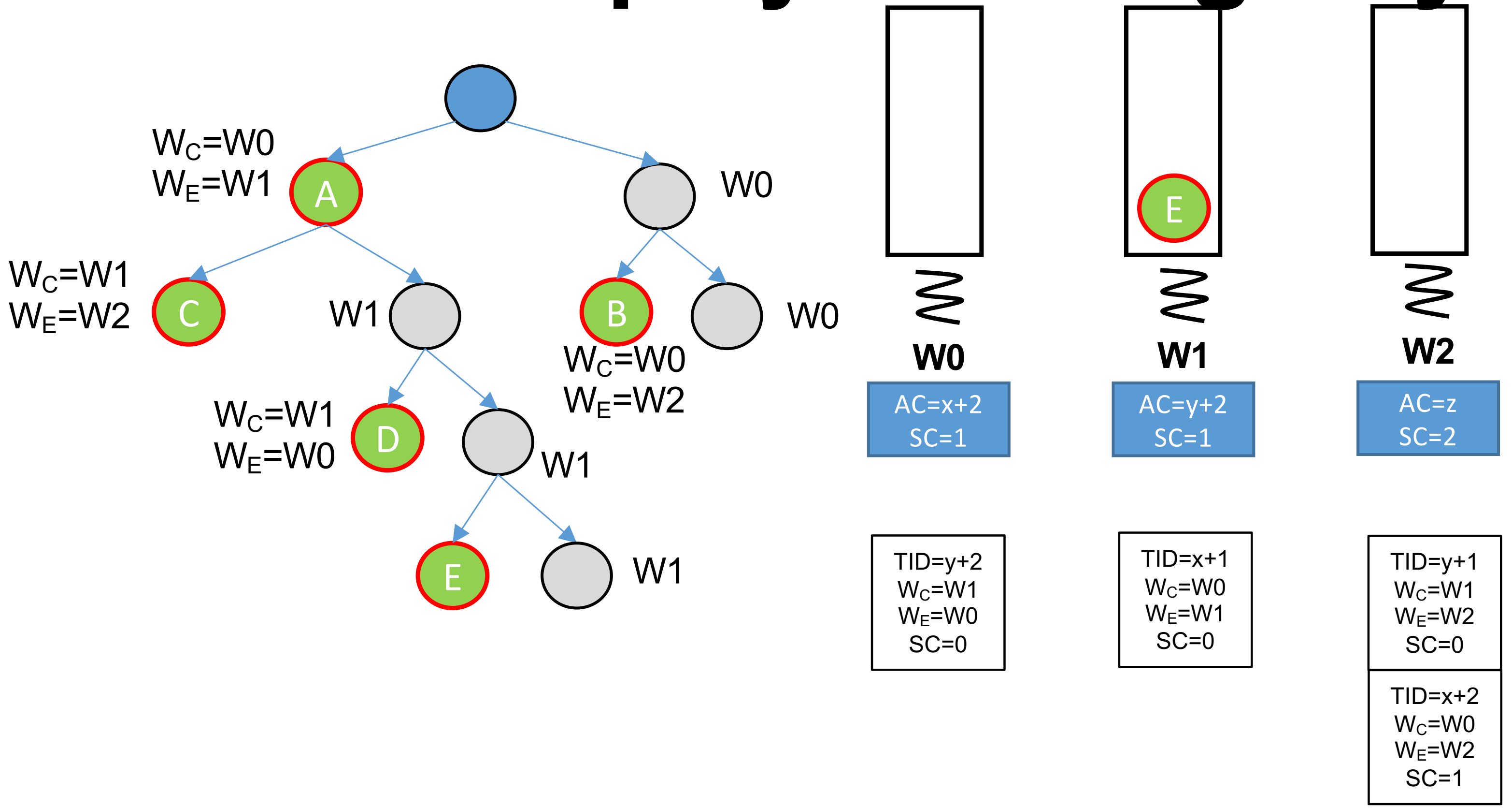
TID=x+1
W_C=W0
W_E=W1
SC=0

TID=y+1
W_C=W1
W_E=W2
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

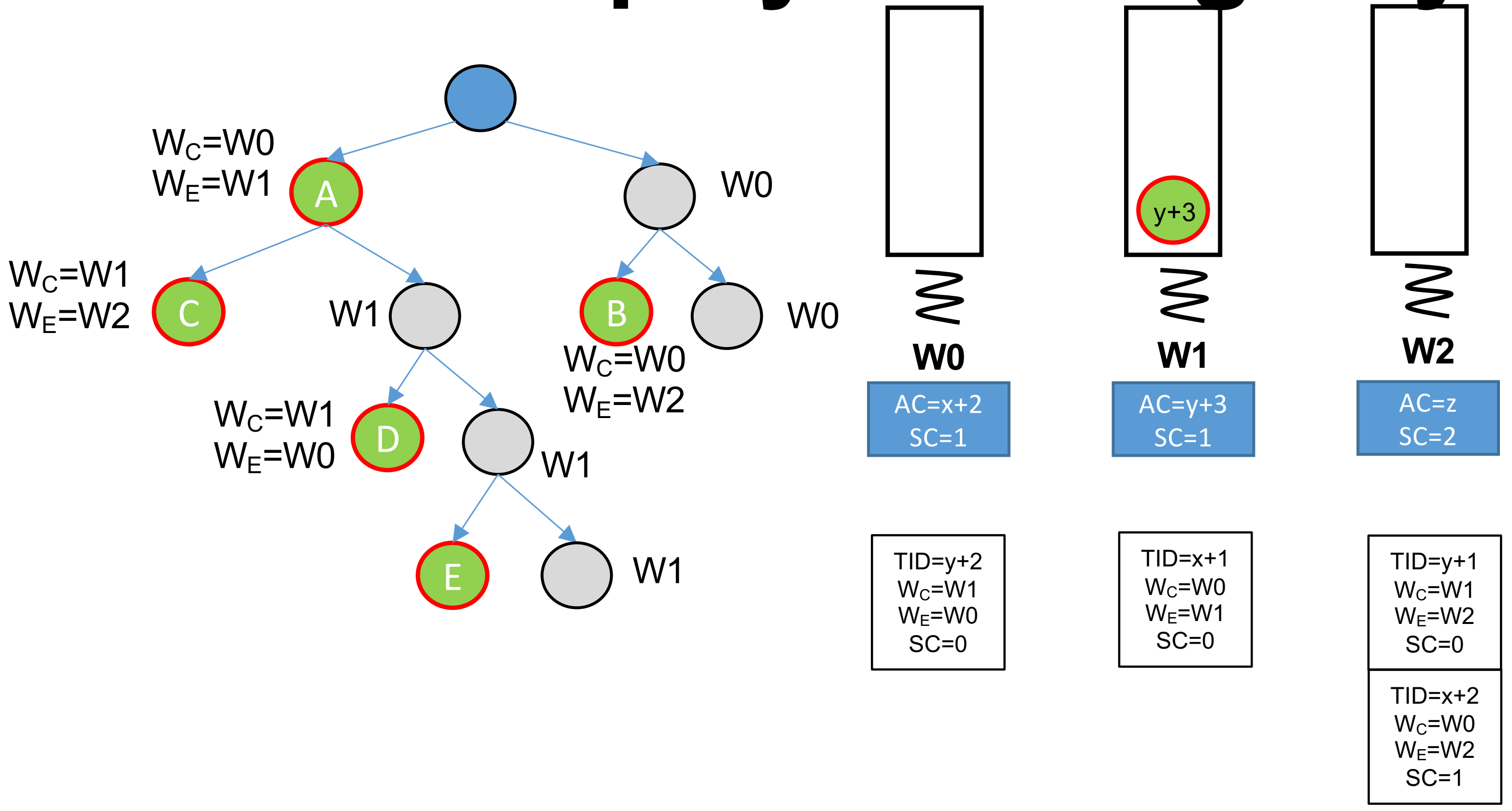
- W0 steals the task D from W1
- It append a node in a private linked list containing info about this stolen task D
  - ID of the task (TID=y+2)
  - Worker who created this task (W_C=W1)
  - Worker who executed (stolen) this async (W_E=W0)
  - Current Steal Counter at W0 (SC=0)
- W0 then increment its Steal Counter (SC) before executing this stolen task

Trace & Replay: Tracing Async



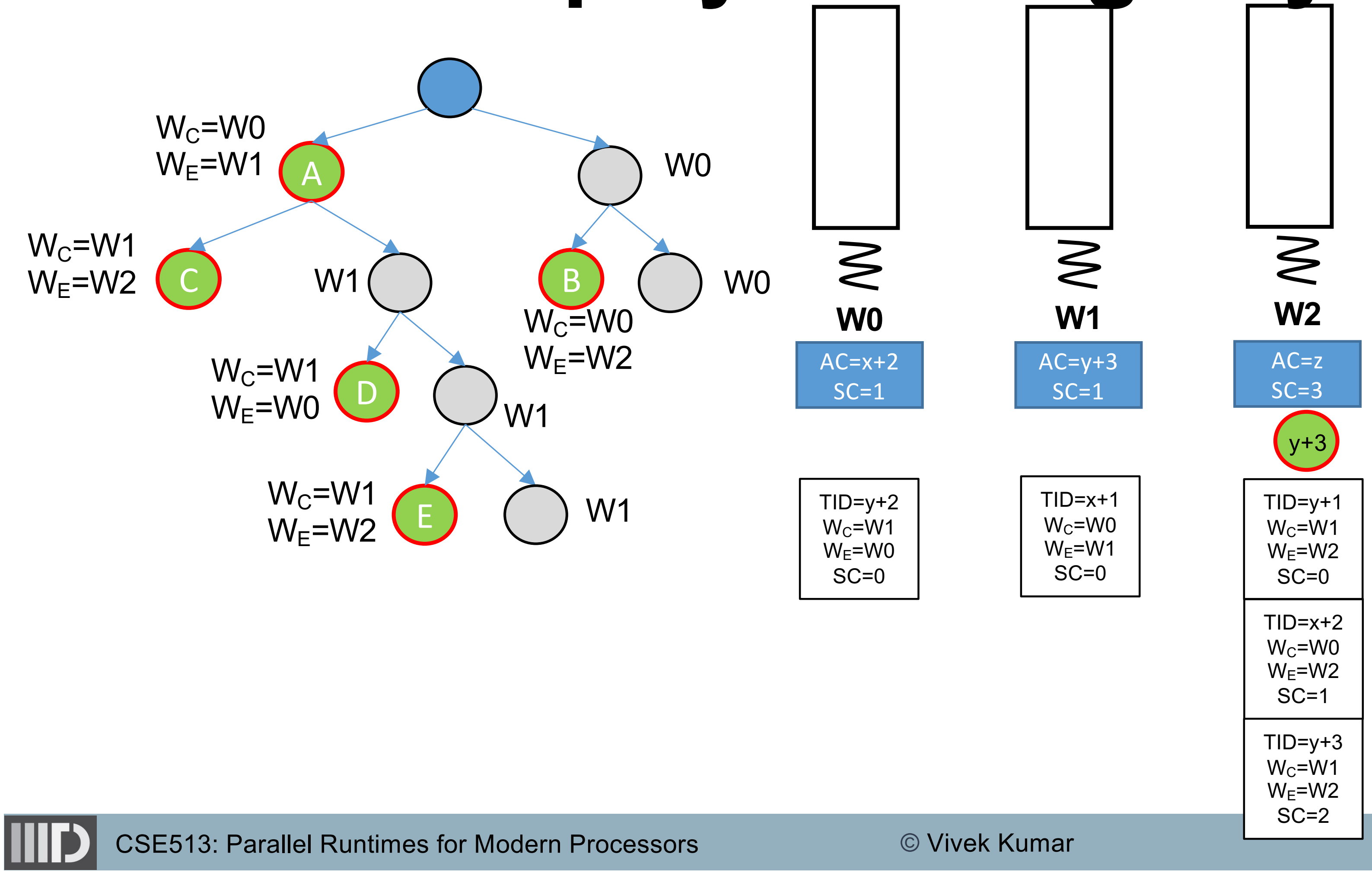
- W1 creates an async E

Trace & Replay: Tracing Async



- AC at W1 is incremented and is assigned as the ID of the Task E **before** its pushed into W1's deque

Trace & Replay: Tracing Async



W0: AC=x+2, SC=1

W1: AC=y+3, SC=1

W2: AC=z, SC=3

TID=y+2, W_C=W1, W_E=W0, SC=0

TID=x+1, W_C=W0, W_E=W1, SC=0

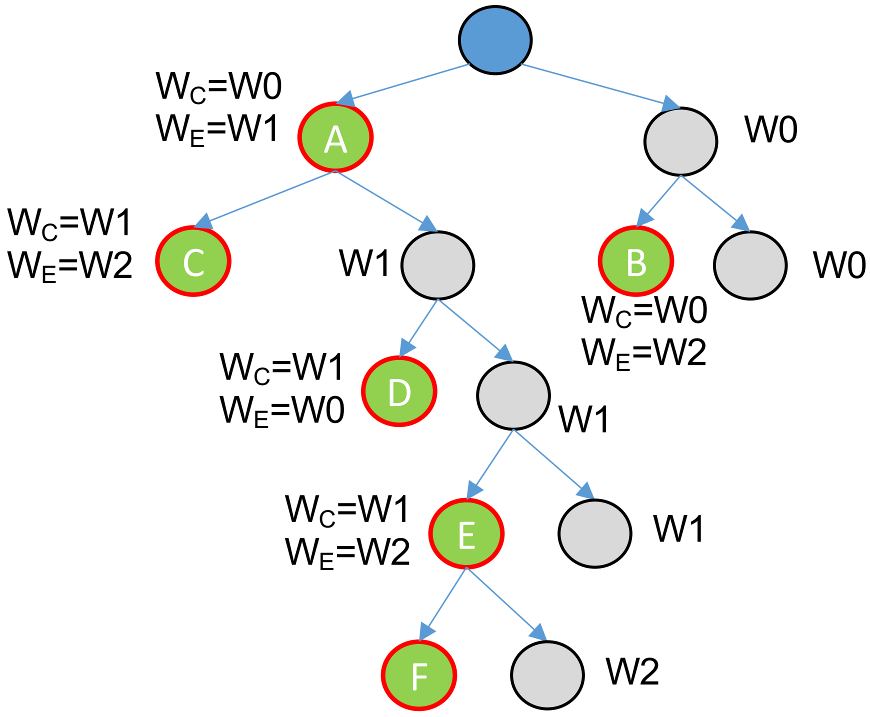
TID=y+1, W_C=W1, W_E=W2, SC=0

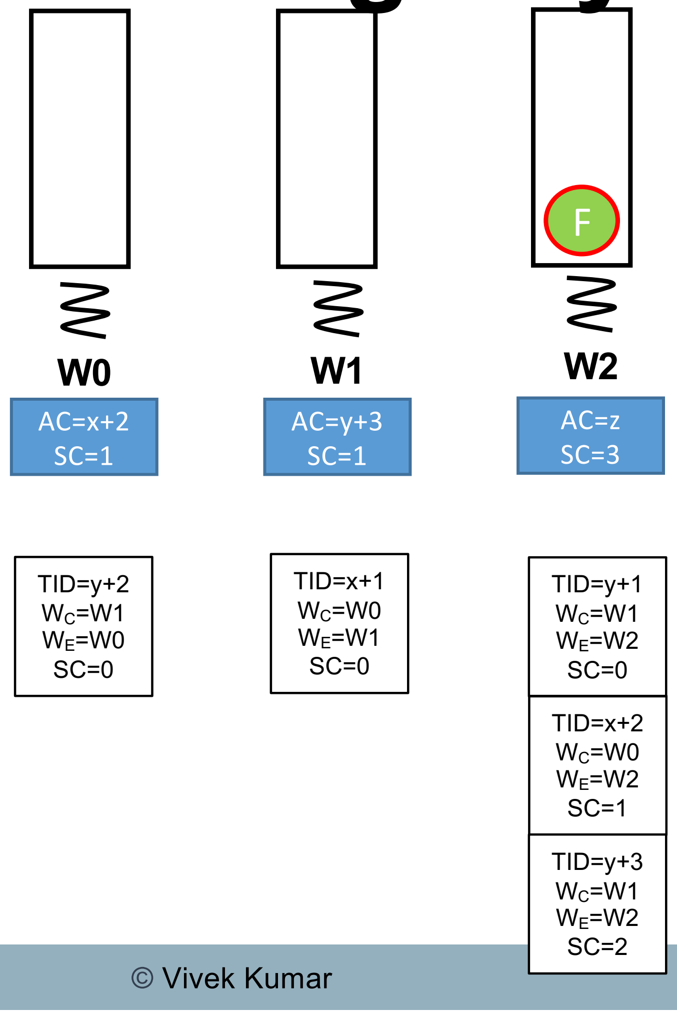
TID=x+2, W_C=W0, W_E=W2, SC=1

TID=y+3, W_C=W1, W_E=W2, SC=2

- W2 steals the task E from W1
- It append a node in a private linked list containing info about this stolen task E
  - ○ ID of the task (TID=y+3)
  - ○ Worker who created this task (W_C=W1)
  - ○ Worker who executed (stolen) this async (W_E=W2)
  - ○ Current Steal Counter at W2 (SC=2)
- W2 then increment its Steal Counter (SC) before executing this stolen task

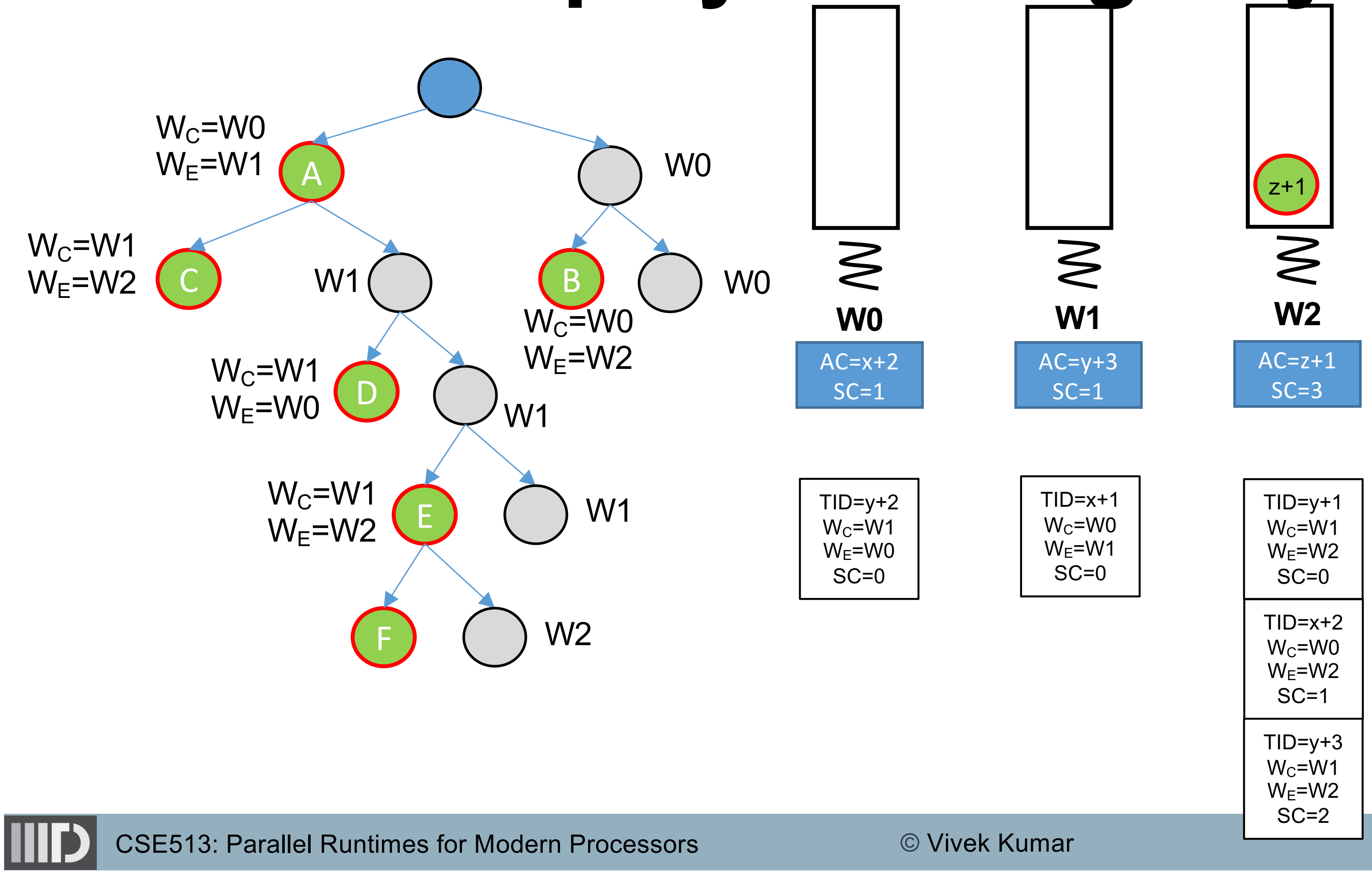
Trace & Replay: Tracing Async





- W2 creates an async F

Trace & Replay: Tracing Async



W0: AC=x+2, SC=1

TID=y+2
W_C=W1
W_E=W0
SC=0

W1: AC=y+3, SC=1

TID=x+1
W_C=W0
W_E=W1
SC=0

W2: AC=z+1, SC=3

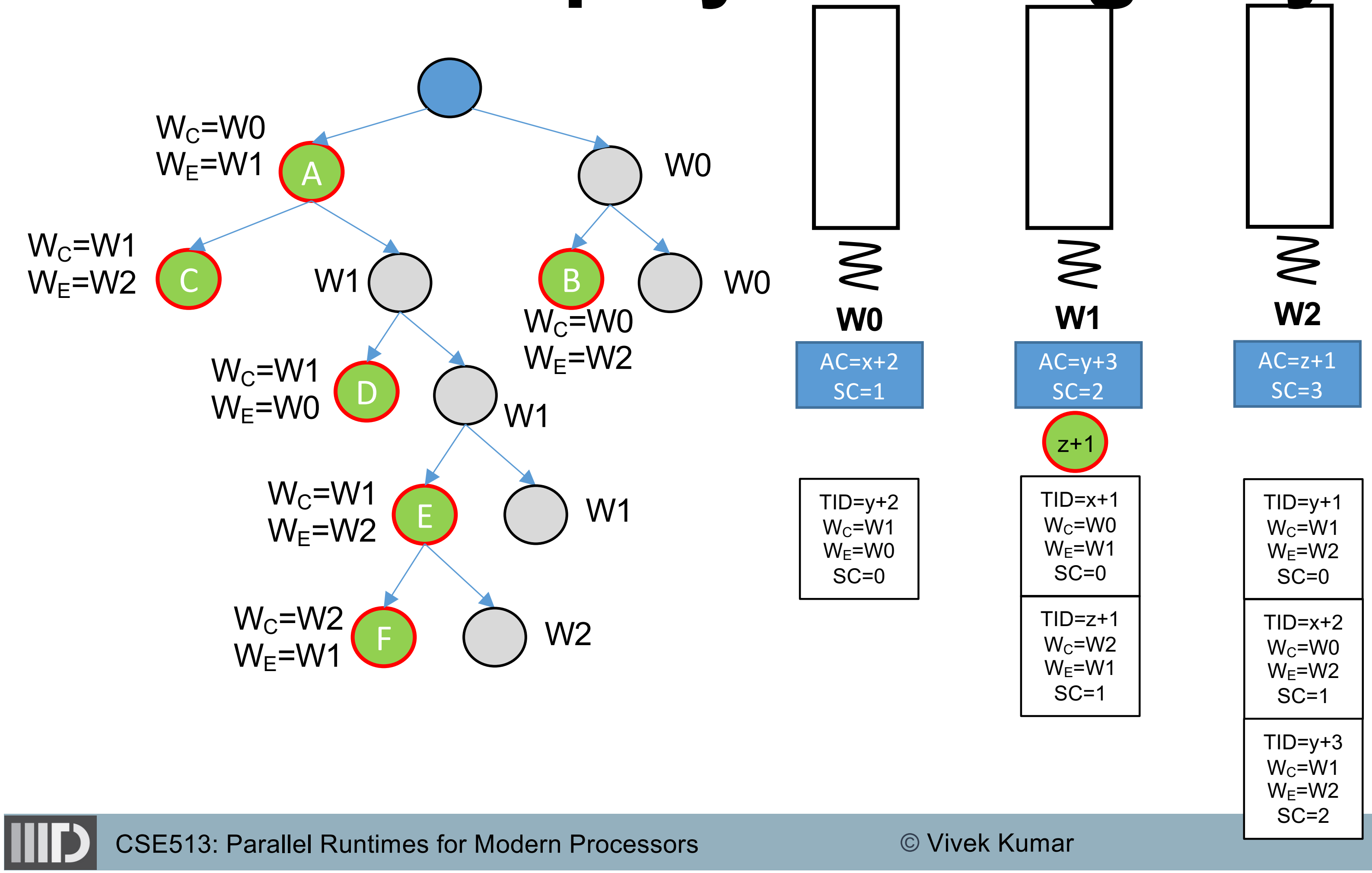
TID=y+1
W_C=W1
W_E=W2
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

TID=y+3
W_C=W1
W_E=W2
SC=2

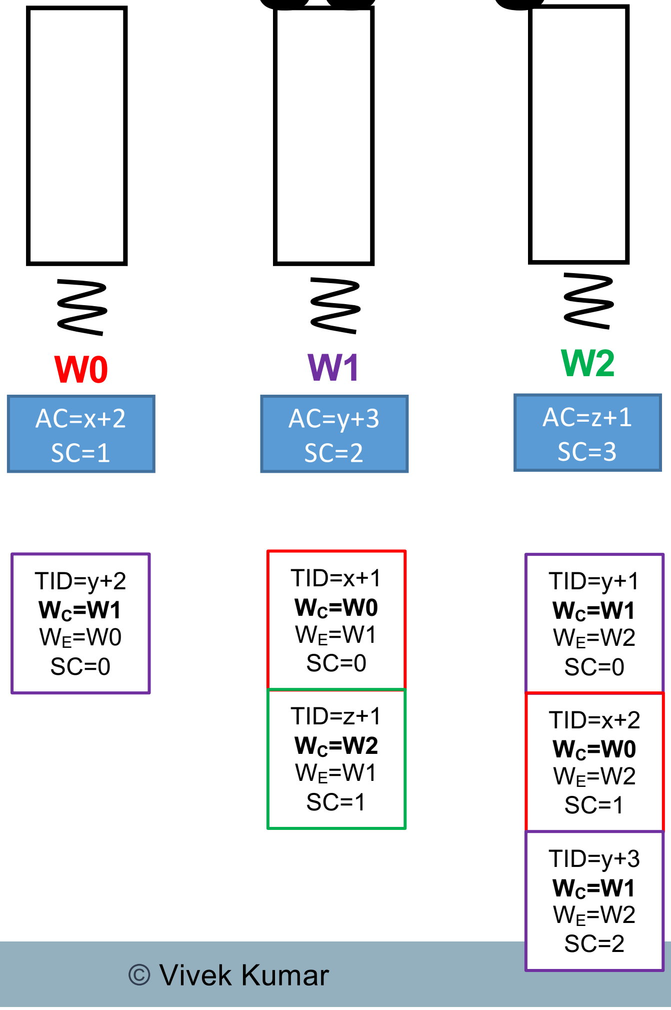
- AC at W2 is incremented and is assigned as the ID of the Task F **before** its pushed into W2's deque

Trace & Replay: Tracing Async



- W1 steals the task F from W2
- It append a node in a private linked list containing info about this stolen task F
  - ID of the task (TID=z+1)
  - Worker who created this task (W_C=W2)
  - Worker who executed (stolen) this async (W_E=W1)
  - Current Steal Counter at W1 (SC=1)
- W1 then increment its Steal Counter (SC) before executing this stolen task

Trace & Replay: List Aggregation



W0

AC=x+2
SC=1

TID=y+2
W_C=W1
W_E=W0
SC=0

W1

AC=y+3
SC=2

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=z+1
W_C=W2
W_E=W1
SC=1

W2

AC=z+1
SC=3

TID=y+1
W_C=W1
W_E=W2
SC=0

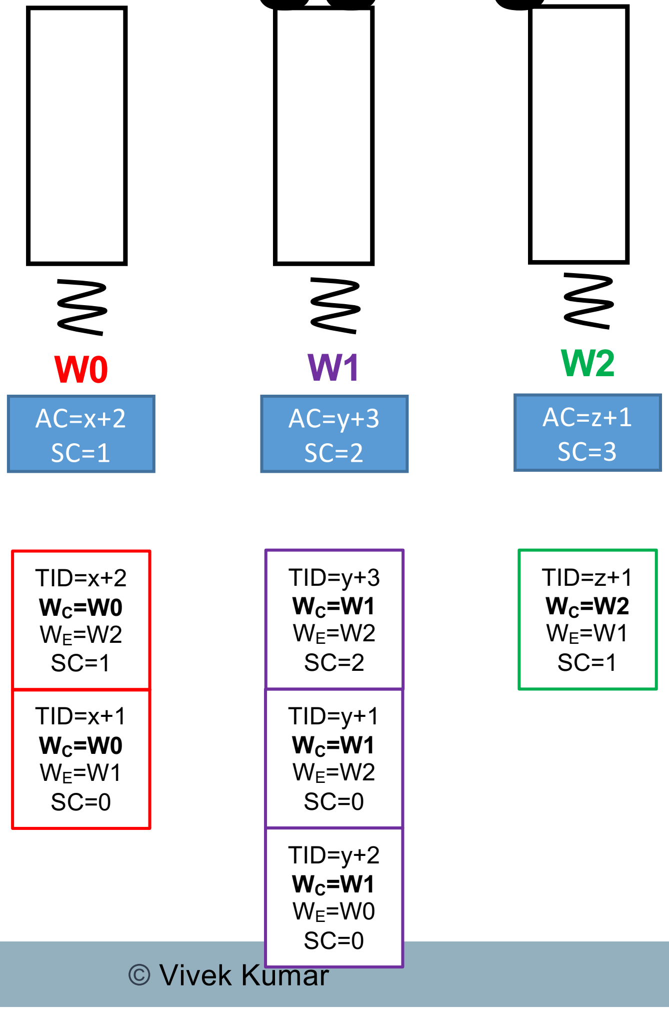
TID=x+2
W_C=W0
W_E=W2
SC=1

TID=y+3
W_C=W1
W_E=W2
SC=2

- Recursive task parallel computation has now completed
- W0 now iterates over the linked list stored at each worker
- W0 aggregates each of the linked list nodes based on the worker who actually created the task corresponding to that node (value of W_C)
  - Hence, there would be numWorker number of linked lists
  - Each worker would finally have nodes with W_C value corresponding to itself

Trace & Replay: List Aggregation

Each worker has the list of nodes corresponding to tasks stolen from itself



TID=x+2
W_C=W0
W_E=W2
SC=1

TID=x+1
W_C=W0
W_E=W1
SC=0

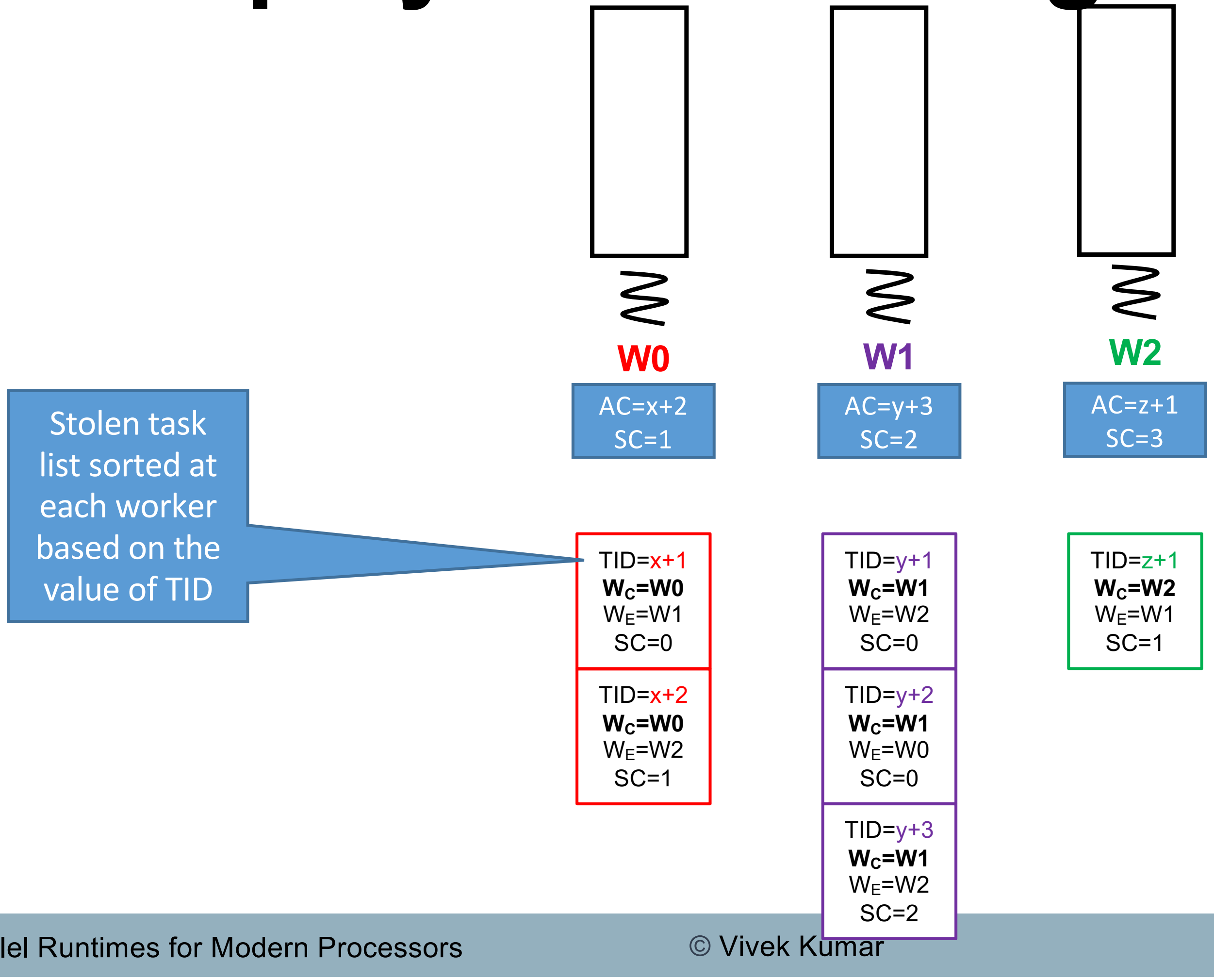
TID=y+3
W_C=W1
W_E=W2
SC=2

TID=y+1
W_C=W1
W_E=W2
SC=0

TID=y+2
W_C=W1
W_E=W0
SC=0

TID=z+1
W_C=W2
W_E=W1
SC=1

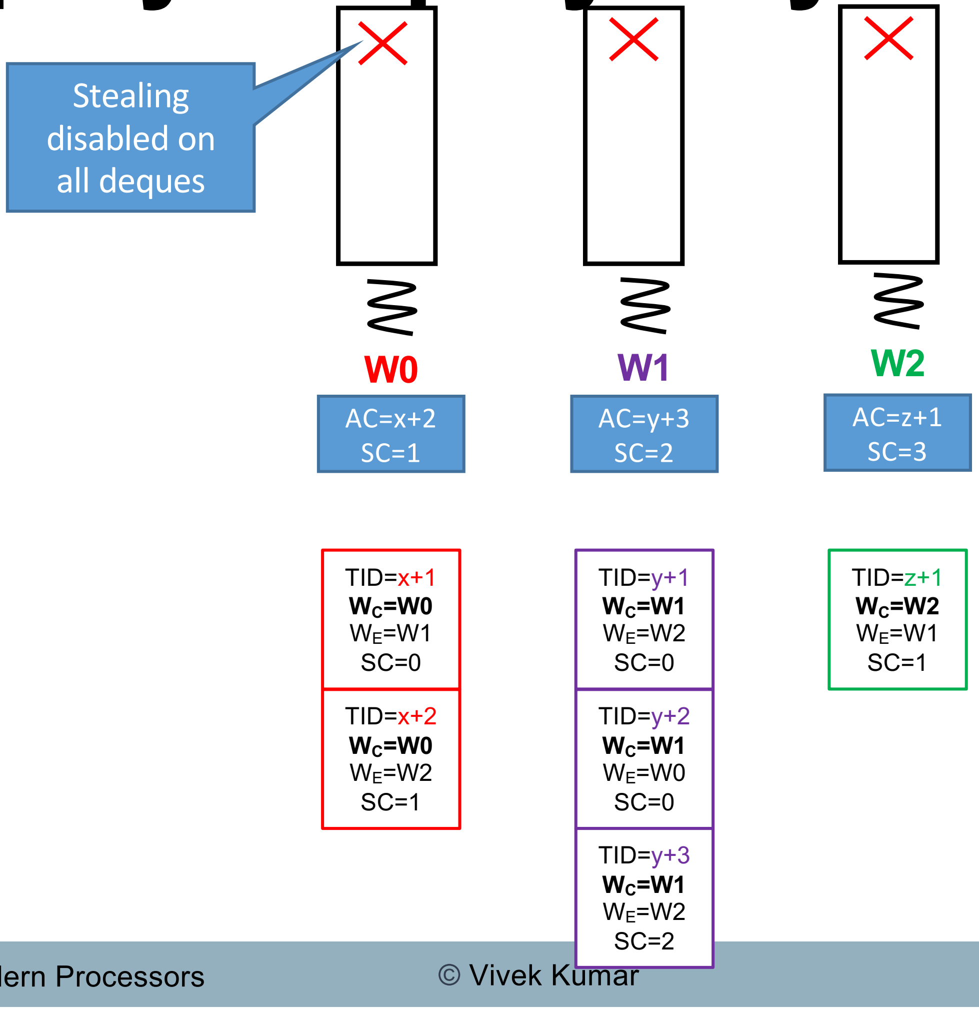
Trace & Replay: List Sorting



Stolen task list sorted at each worker based on the value of TID

- W0 will sort each of these lists (at each worker) based on the TID stored inside the nodes

Trace & Replay: Replay Async



Stealing disabled on all deques

AC=x+2
SC=1

AC=y+3
SC=2

AC=z+1
SC=3

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

TID=y+1
W_C=W1
W_E=W2
SC=0

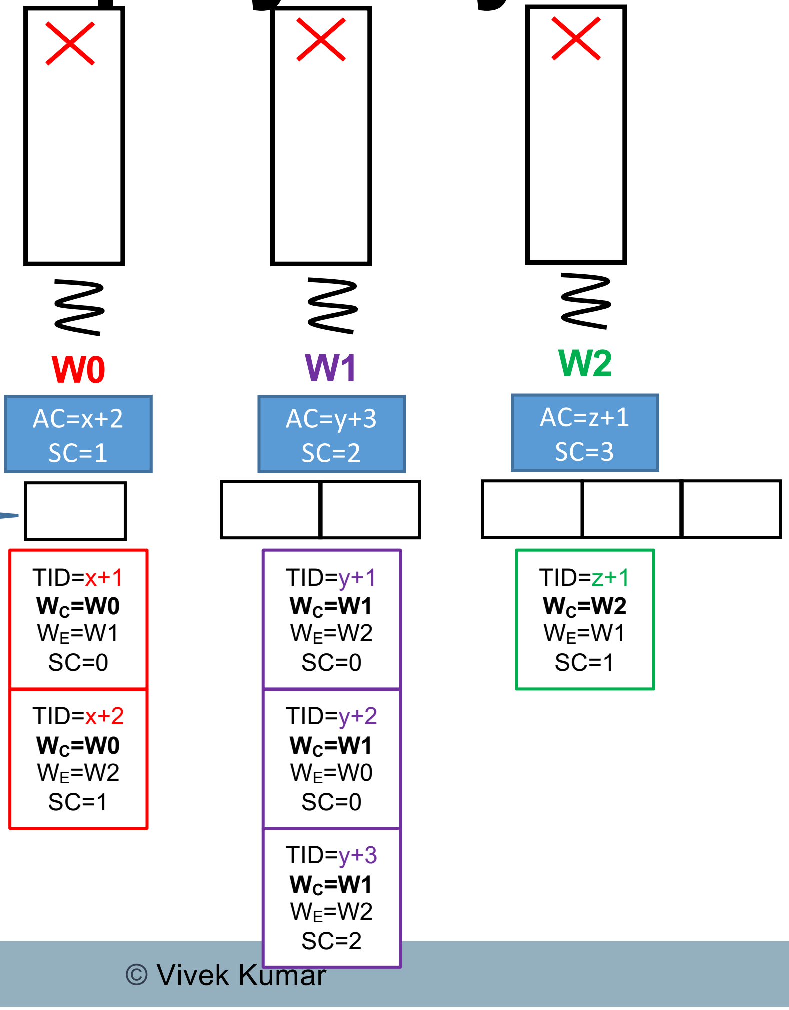
TID=y+2
W_C=W1
W_E=W0
SC=0

TID=y+3
W_C=W1
W_E=W2
SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

- Replay phase is essentially executing the same recursive task parallel program, but by using the steal information stored at each worker during the tracing phase
- During the replay phase, each worker would disable the direct steal operation on its deque

Trace & Replay: Replay Async

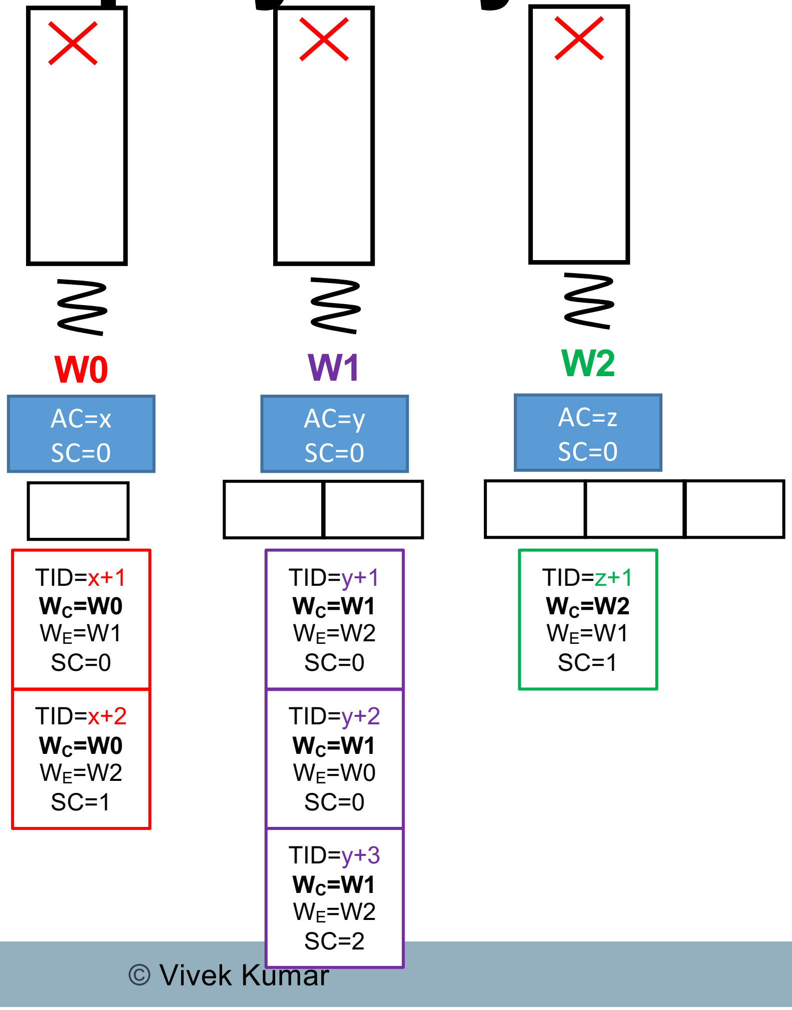


Arrays created at each worker to hold stolen tasks. Size of the array is the value of Steal Counter (SC) at that worker

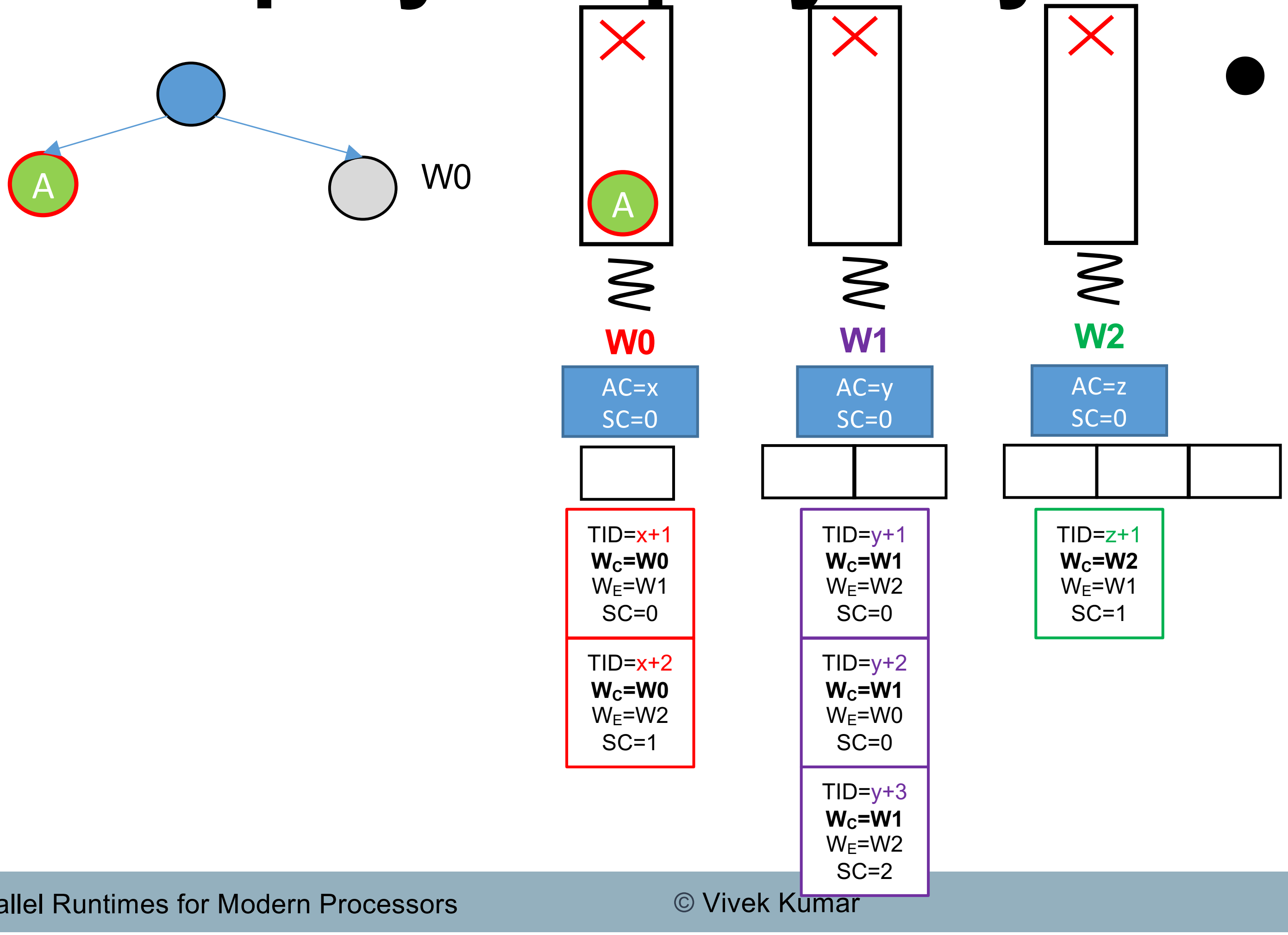
Trace & Replay: Replay Async

Async Counter (AC) reinitialized at each workers

Steal Counter (SC) reinitialized to 0 at each workers



Trace & Replay: Replay Async



AC=x
SC=0

AC=y
SC=0

AC=z
SC=0

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

TID=y+1
W_C=W1
W_E=W2
SC=0

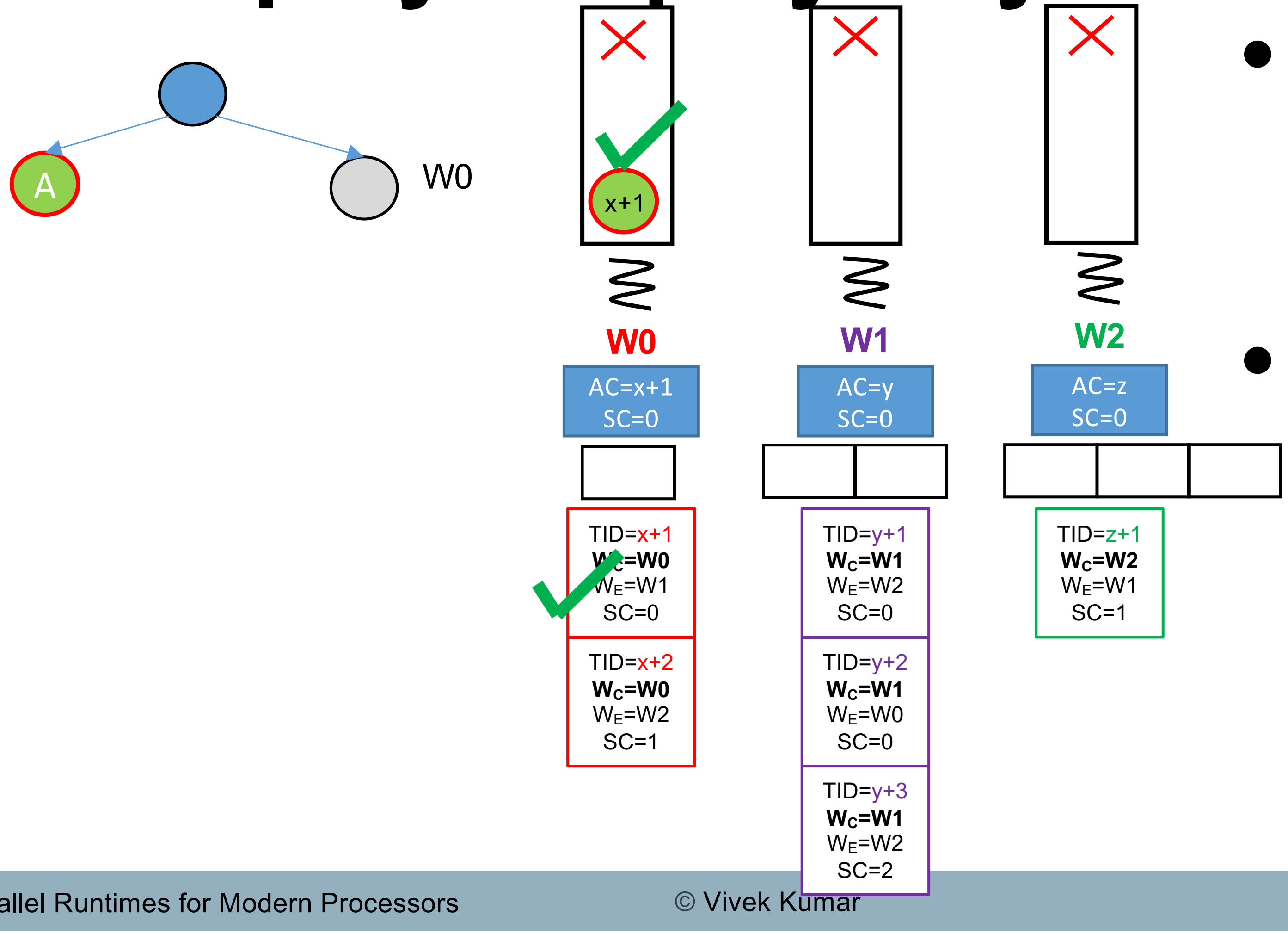
TID=y+2
W_C=W1
W_E=W0
SC=0

TID=y+3
W_C=W1
W_E=W2
SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

- W0 starts the computation and creates an async A

Trace & Replay: Replay Async



AC=x+1
SC=0

AC=y
SC=0

AC=z
SC=0

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

TID=y+1
W_C=W1
W_E=W2
SC=0

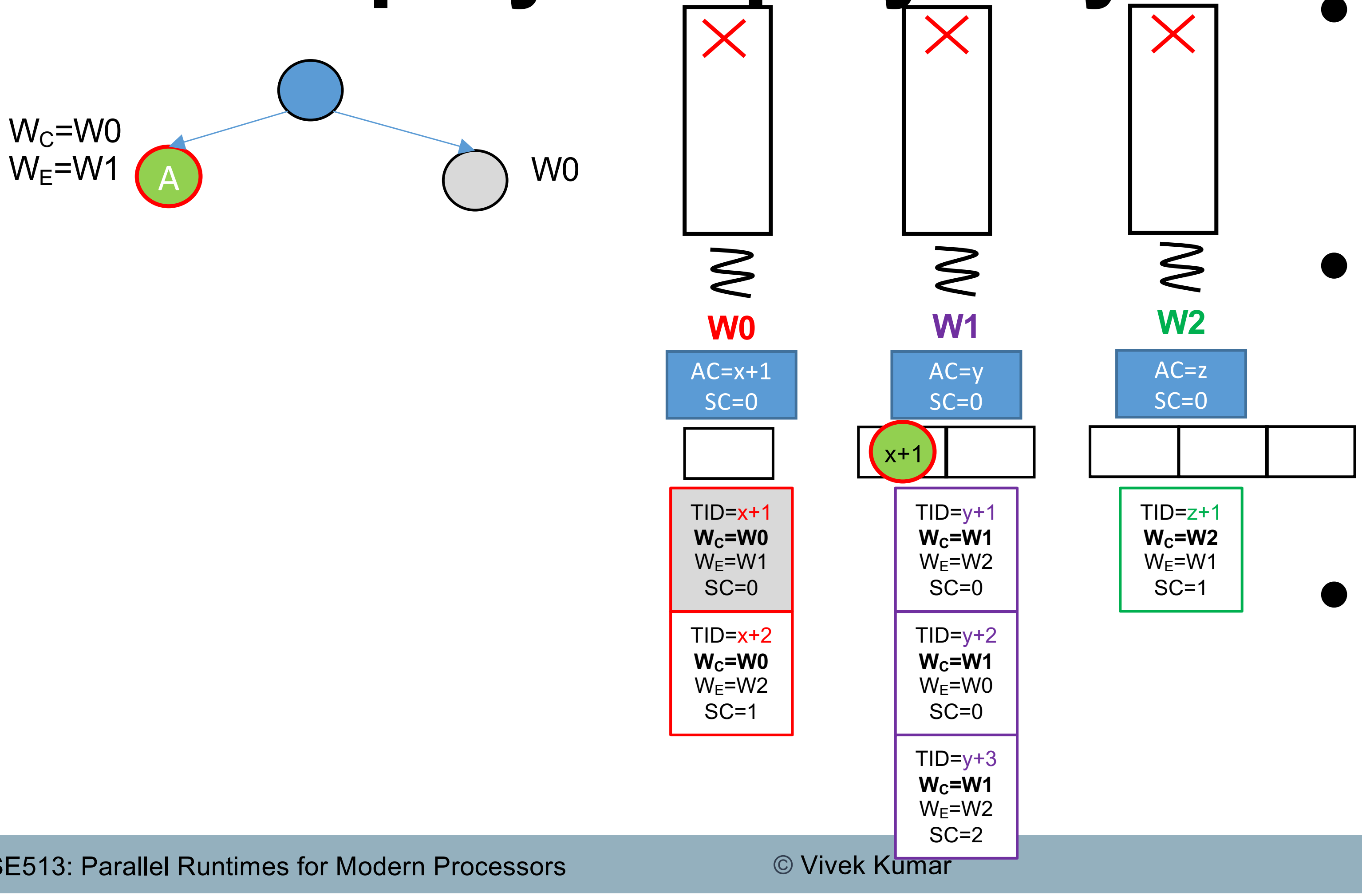
TID=y+2
W_C=W1
W_E=W0
SC=0

TID=y+3
W_C=W1
W_E=W2
SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

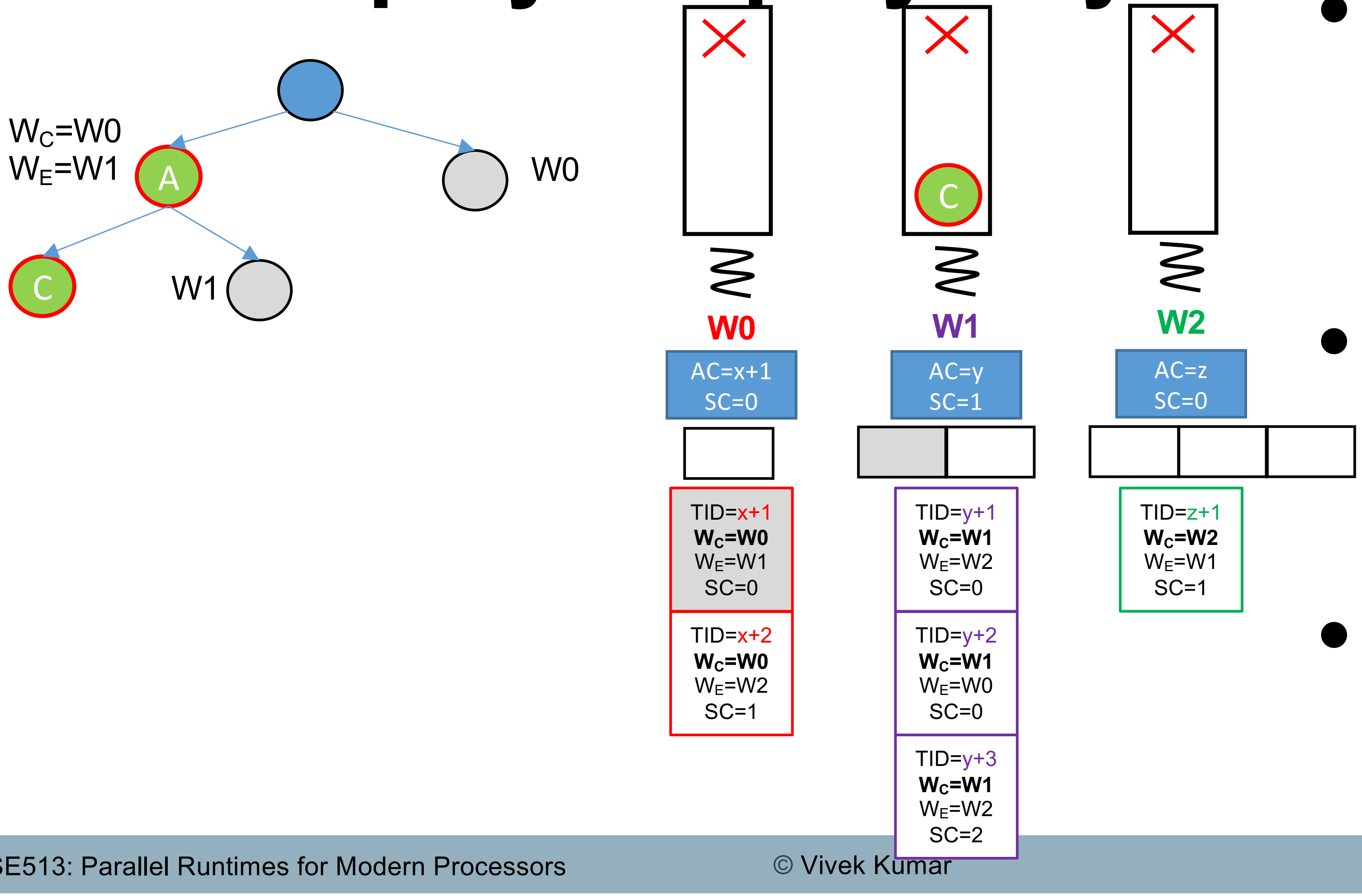
- AC at W0 is incremented and is assigned as the ID of the Task A **before** its pushed into W0’s deque
- When W0 attempts to push task A into its deque, it would observe that the TID of A matches with the currently active steal node on its linked list

Trace & Replay: Replay Async



- W0 does not push task A into its deque, but directly copies it into the array at W1
- A is copied into an index value corresponding to SC counter stored inside the steal info node of task A at W0 (i.e., 0)
- W0 remove the currently pointing steal node from its linked list

Trace & Replay: Replay Async



TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

TID=y+1
W_C=W1
W_E=W2
SC=0

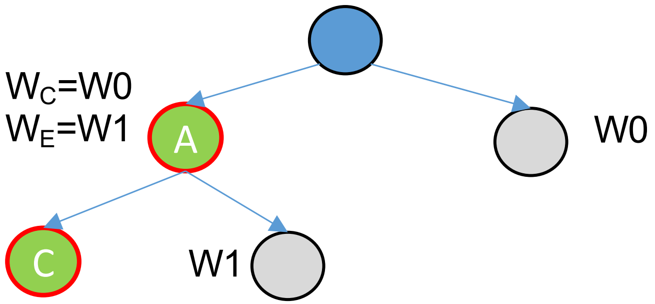
TID=y+2
W_C=W1
W_E=W0
SC=0

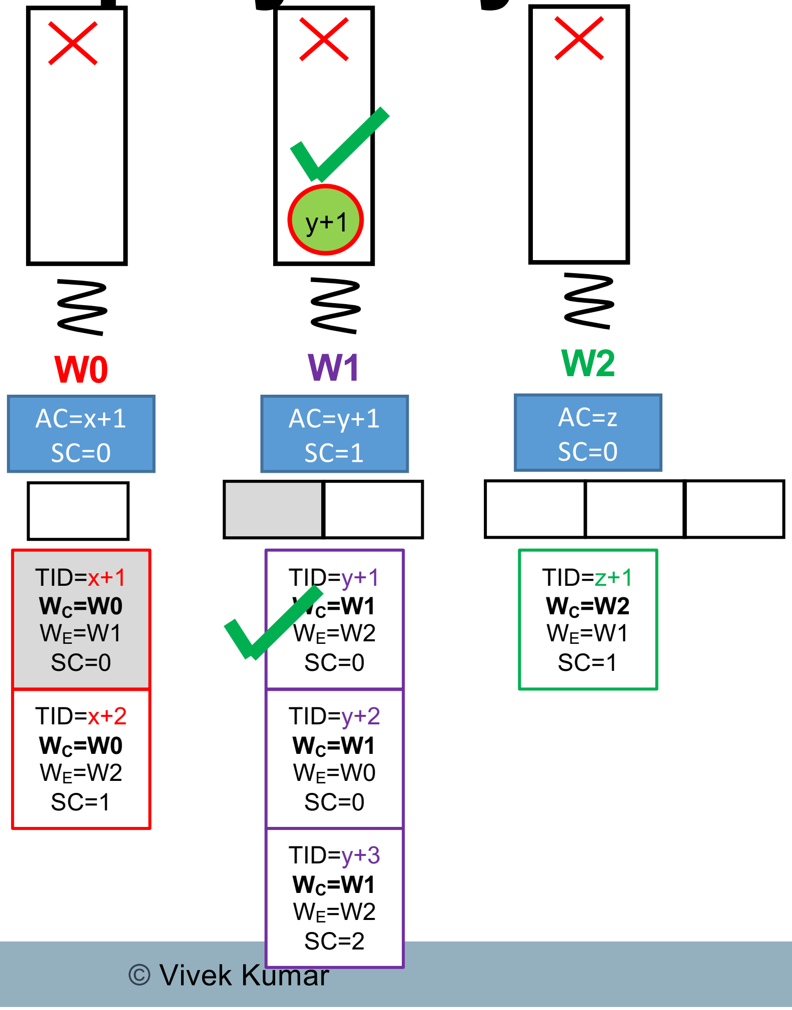
TID=y+3
W_C=W1
W_E=W2
SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

- Until now, W1 was waiting for a task to be available in its task array at an index of its current SC value (i.e., 0)
- After receiving the task, W1 will increment its SC value and will start executing the transferred task
- W1 generates an async C once it starts the execution of the transferred task

Trace & Replay: Replay Async





W0	W1	W2

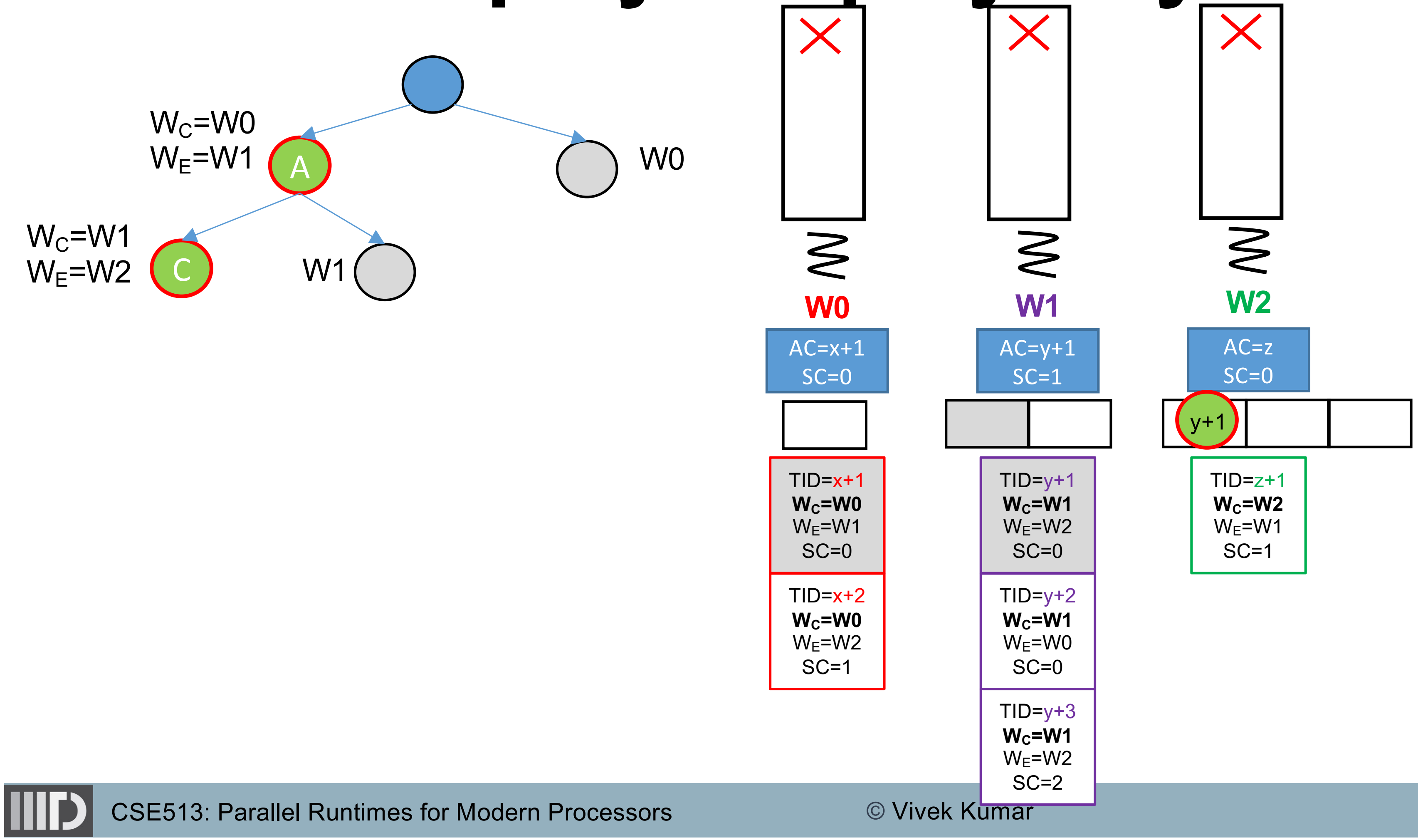
AC=x+1
SC=0
	AC=y+1
	SC=1
		AC=z
		SC=0

TID=x+1
W_C=W0
W_E=W1
SC=0
	TID=y+1
	W_C=W1
	W_E=W2
	SC=0
		TID=z+1
		W_C=W2
		W_E=W1
		SC=1

TID=x+2
W_C=W0
W_E=W2
SC=1
	TID=y+2
	W_C=W1
	W_E=W0
	SC=0	

	TID=y+3
	W_C=W1
	W_E=W2
	SC=2	

Trace & Replay: Replay Async



W_C=W0
W_E=W1

A

W0

W_C=W1
W_E=W2

C

W1

W0

AC=x+1
SC=0

TID=x+1
$\mathbf{W_C=W0}$
W_E=W1
SC=0

TID=x+2
$\mathbf{W_C=W0}$
W_E=W2
SC=1

W1

AC=y+1
SC=1

TID=y+1
$\mathbf{W_C=W1}$
W_E=W2
SC=0

TID=y+2
$\mathbf{W_C=W1}$
W_E=W0
SC=0

TID=y+3
$\mathbf{W_C=W1}$
W_E=W2
SC=2

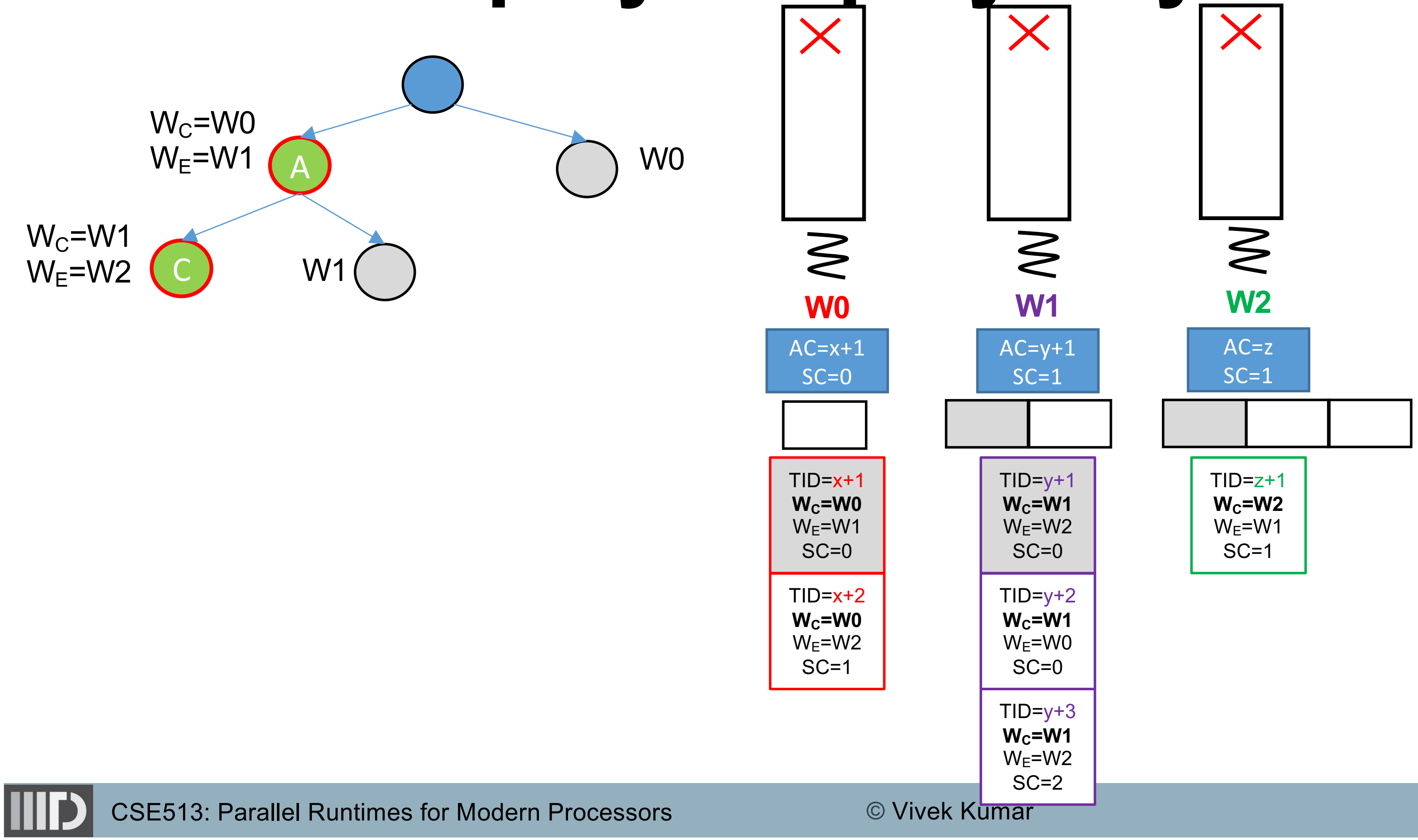
W2

AC=z
SC=0

y+1

TID=z+1
$\mathbf{W_C=W2}$
W_E=W1
SC=1

Trace & Replay: Replay Async



W0
AC=x+1
SC=0

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

W1
AC=y+1
SC=1

TID=y+1
W_C=W1
W_E=W2
SC=0

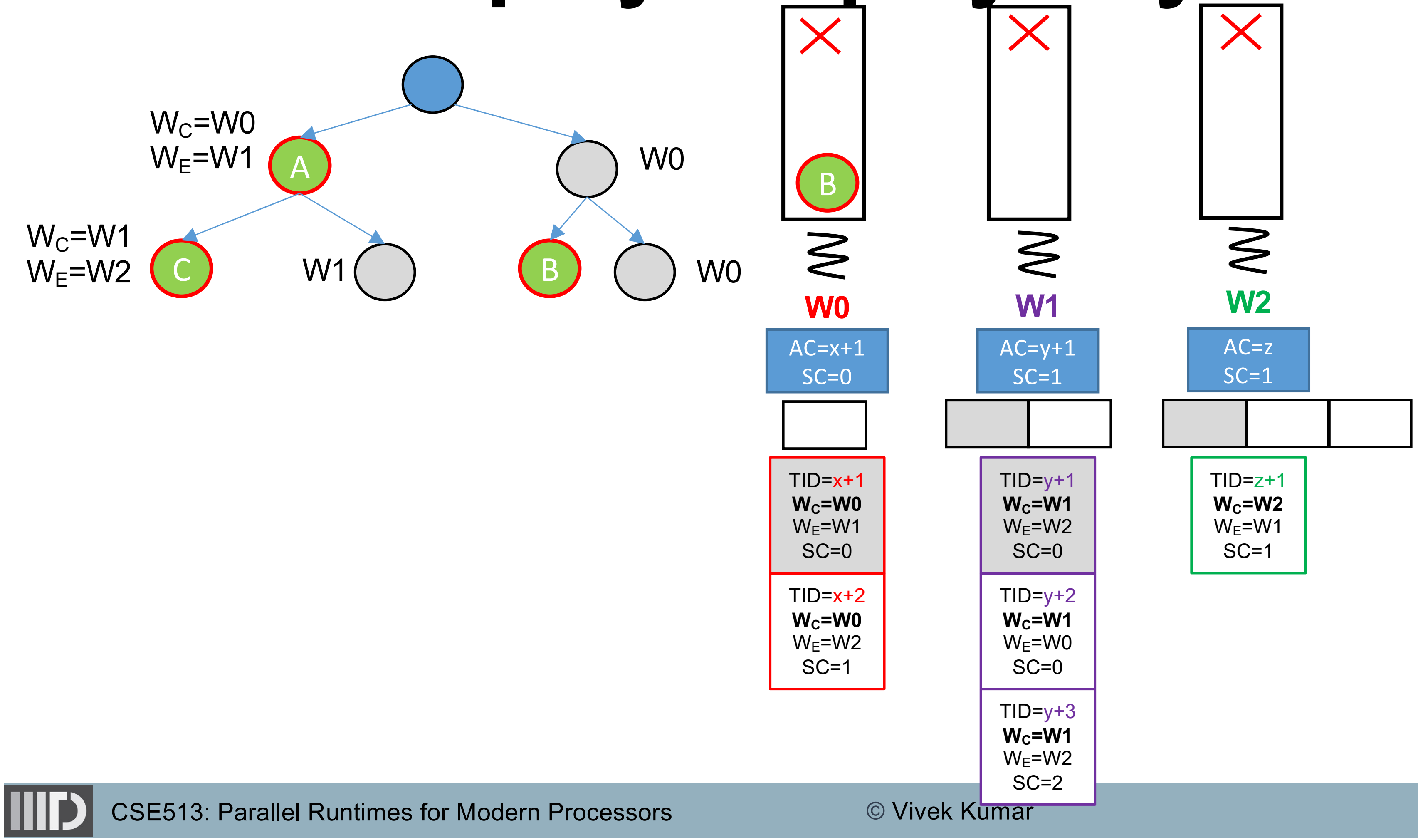
TID=y+2
W_C=W1
W_E=W0
SC=0

TID=y+3
W_C=W1
W_E=W2
SC=2

W2
AC=z
SC=1

TID=z+1
W_C=W2
W_E=W1
SC=1

Trace & Replay: Replay Async



TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

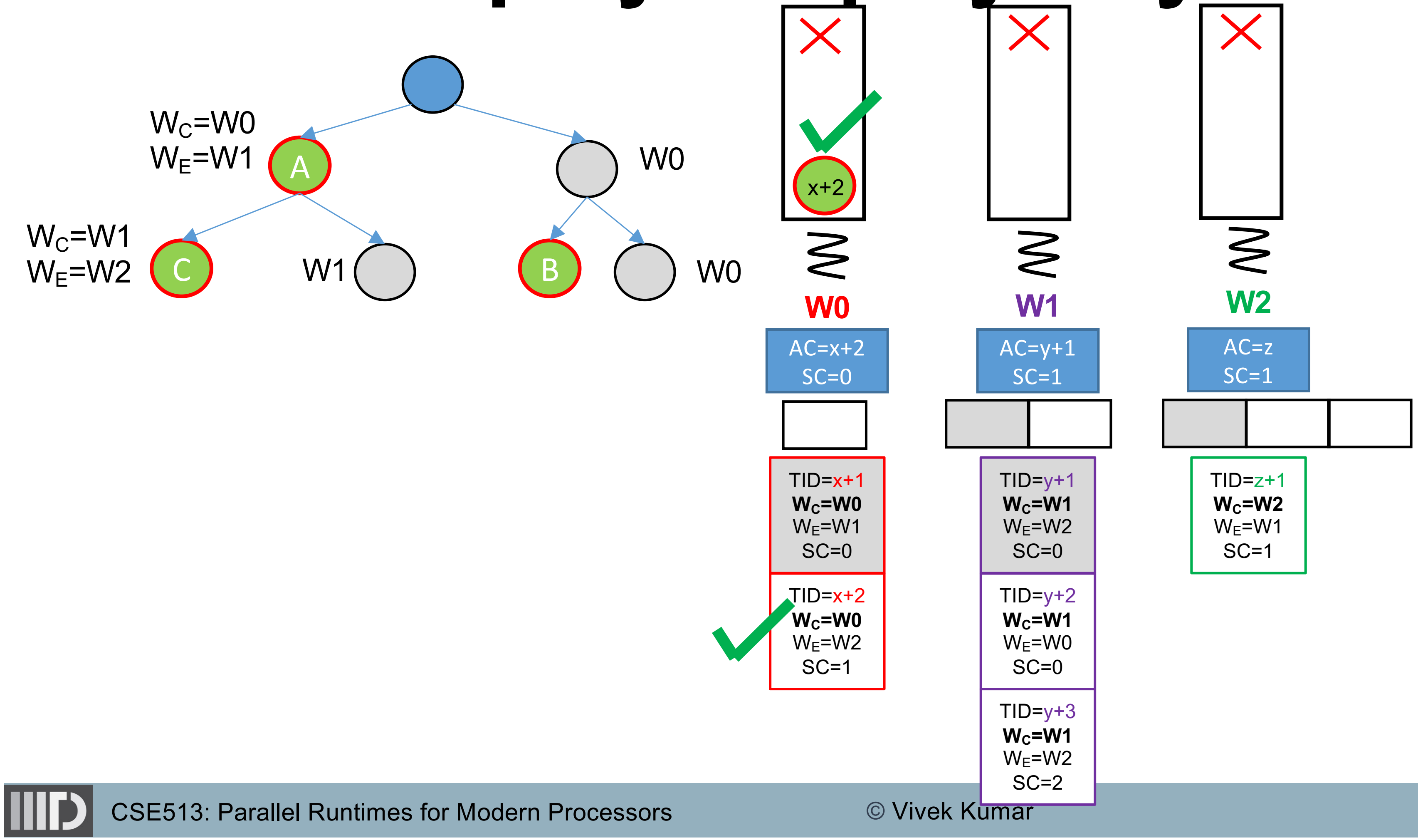
TID=y+1
W_C=W1
W_E=W2
SC=0

TID=y+2
W_C=W1
W_E=W0
SC=0

TID=y+3
W_C=W1
W_E=W2
SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

Trace & Replay: Replay Async



W0	W1	W2

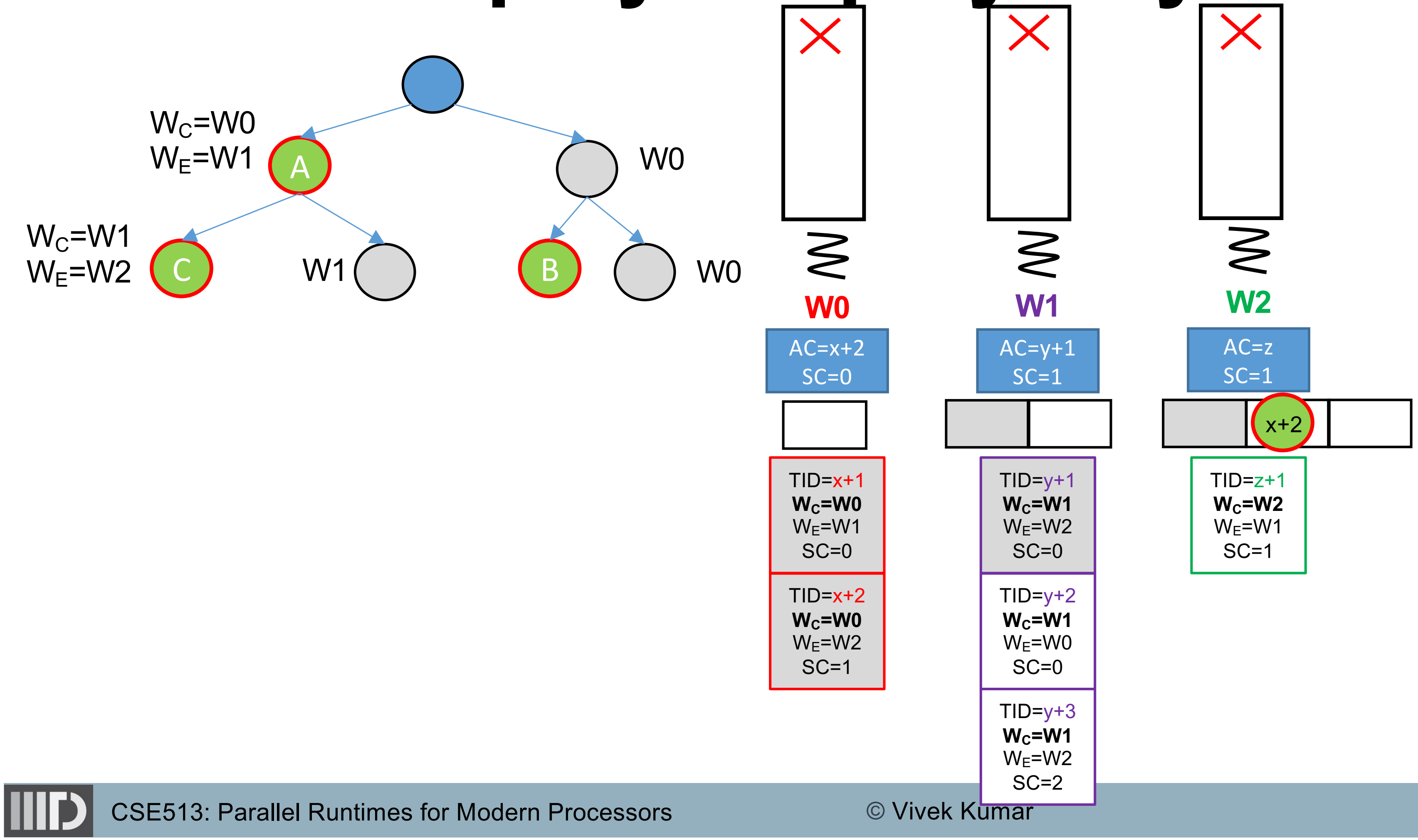
AC=x+2
SC=0
	AC=y+1
	SC=1
		AC=z
		SC=1

TID=x+1
W_C=W0
W_E=W1
SC=0
	TID=y+1
	W_C=W1
	W_E=W2
	SC=0
		TID=z+1
		W_C=W2
		W_E=W1
		SC=1

✓ TID=x+2
W_C=W0
W_E=W2
SC=1
	TID=y+2
	W_C=W1
	W_E=W0
	SC=0	

	TID=y+3
	W_C=W1
	W_E=W2
	SC=2	

Trace & Replay: Replay Async



W0: AC=x+2, SC=0

W1: AC=y+1, SC=1

W2: AC=z, SC=1

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

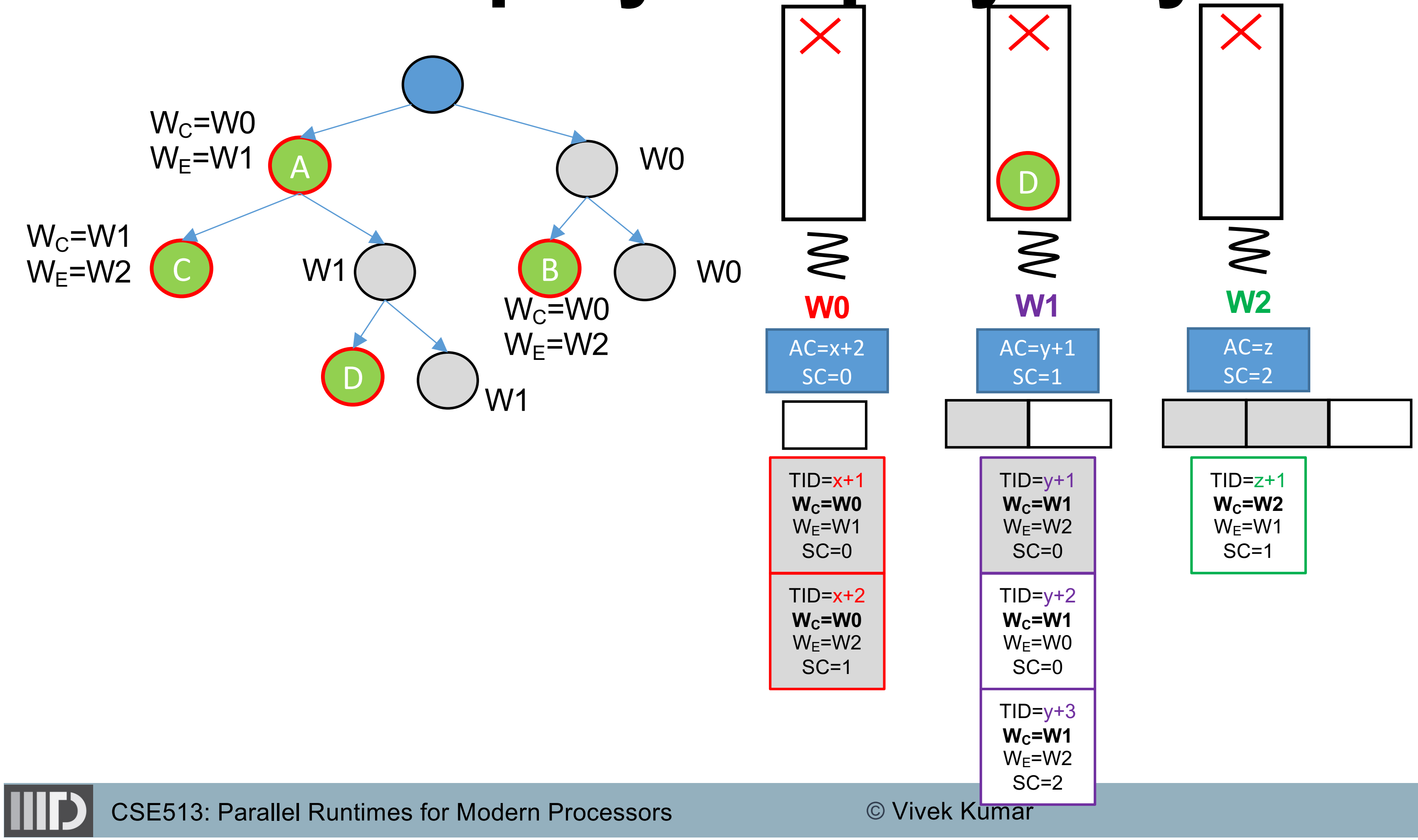
TID=y+1
W_C=W1
W_E=W2
SC=0

TID=y+2
W_C=W1
W_E=W0
SC=0

TID=y+3
W_C=W1
W_E=W2
SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

Trace & Replay: Replay Async



W0

AC=x+2
SC=0

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

W1

AC=y+1
SC=1

TID=y+1
W_C=W1
W_E=W2
SC=0

TID=y+2
W_C=W1
W_E=W0
SC=0

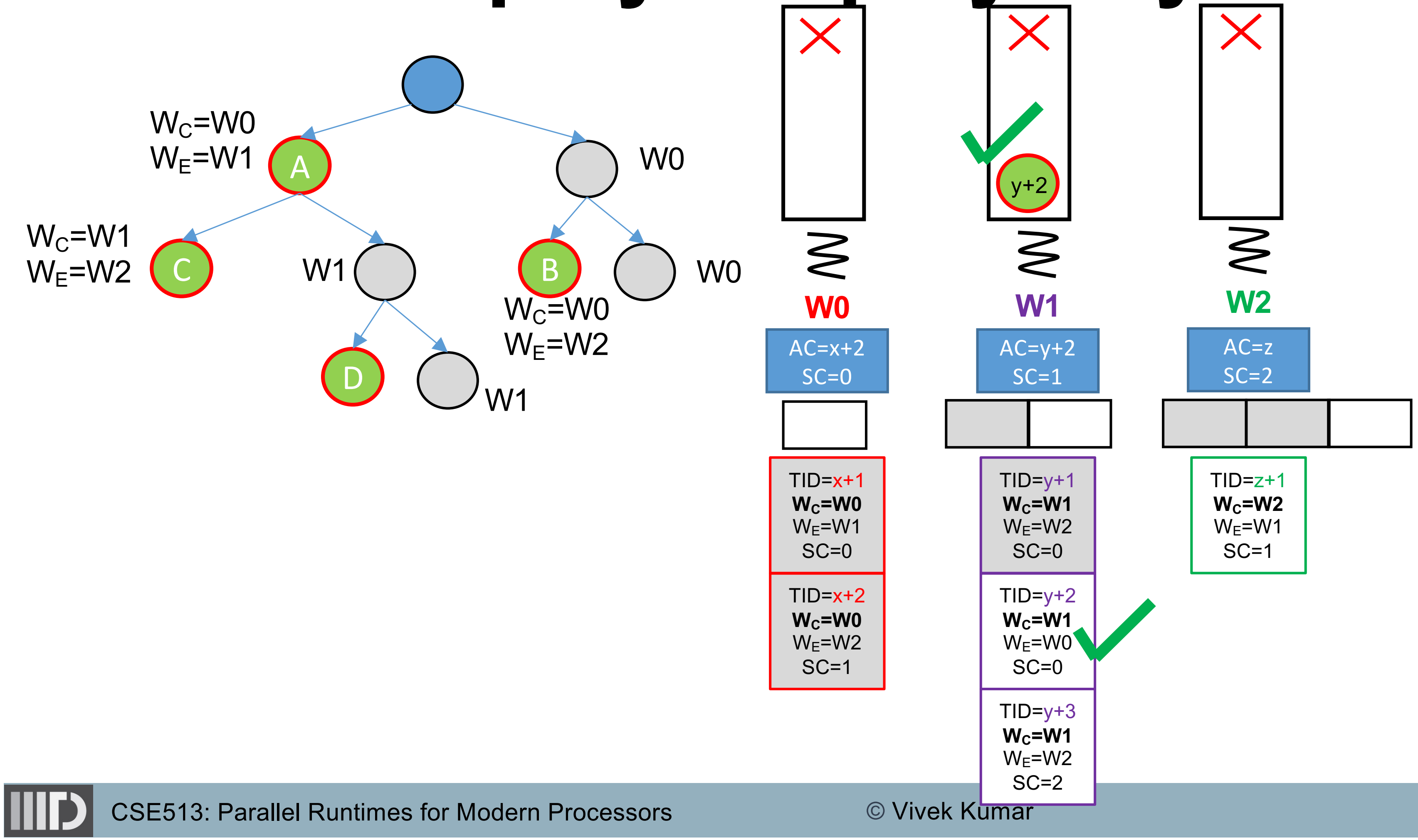
TID=y+3
W_C=W1
W_E=W2
SC=2

W2

AC=z
SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

Trace & Replay: Replay Async



AC=x+2
SC=0

AC=y+2
SC=1

AC=z
SC=2

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

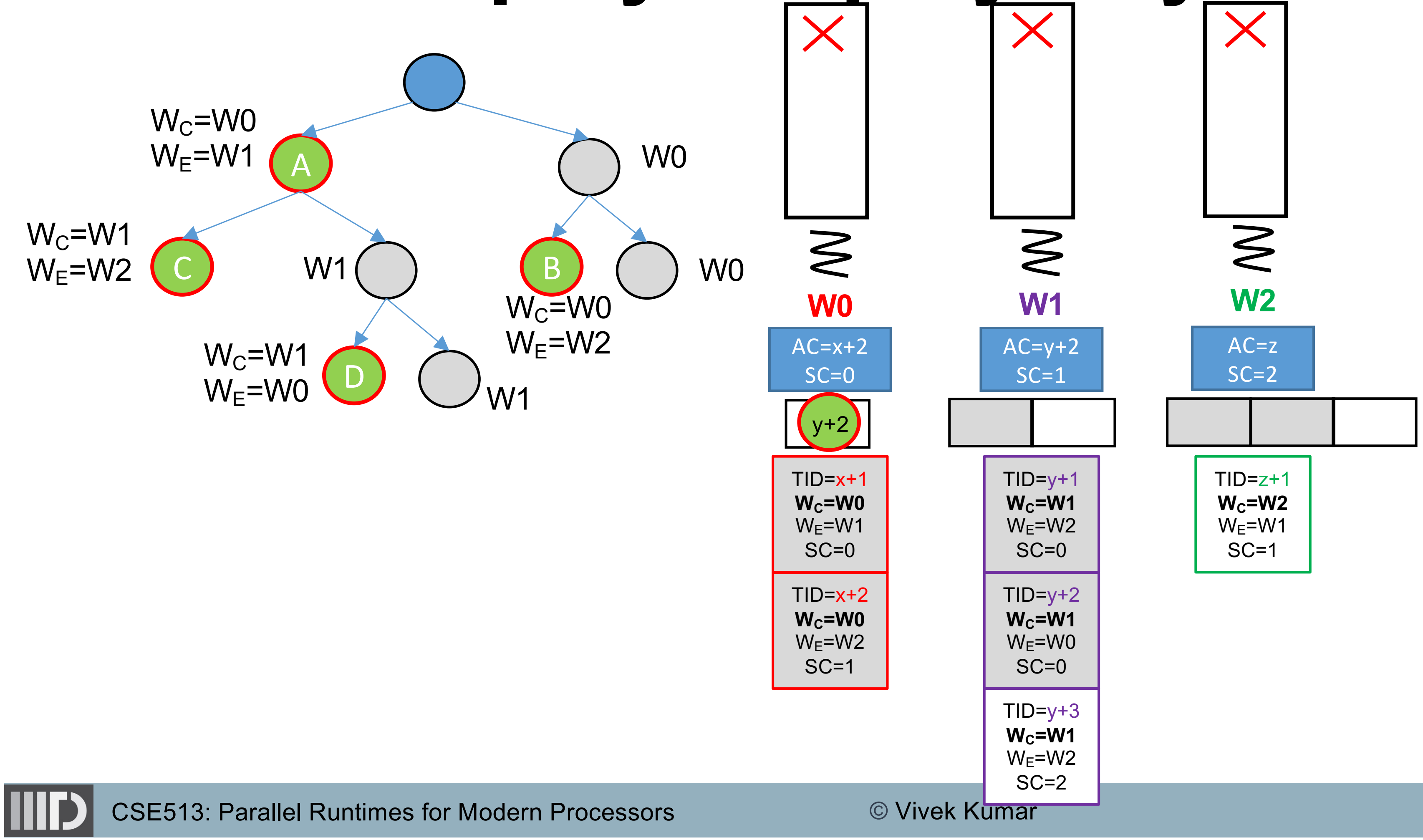
TID=y+1
W_C=W1
W_E=W2
SC=0

TID=y+2
W_C=W1
W_E=W0
SC=0

TID=y+3
W_C=W1
W_E=W2
SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

Trace & Replay: Replay Async



W0: AC=x+2, SC=0

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

W1: AC=y+2, SC=1

TID=y+1
W_C=W1
W_E=W2
SC=0

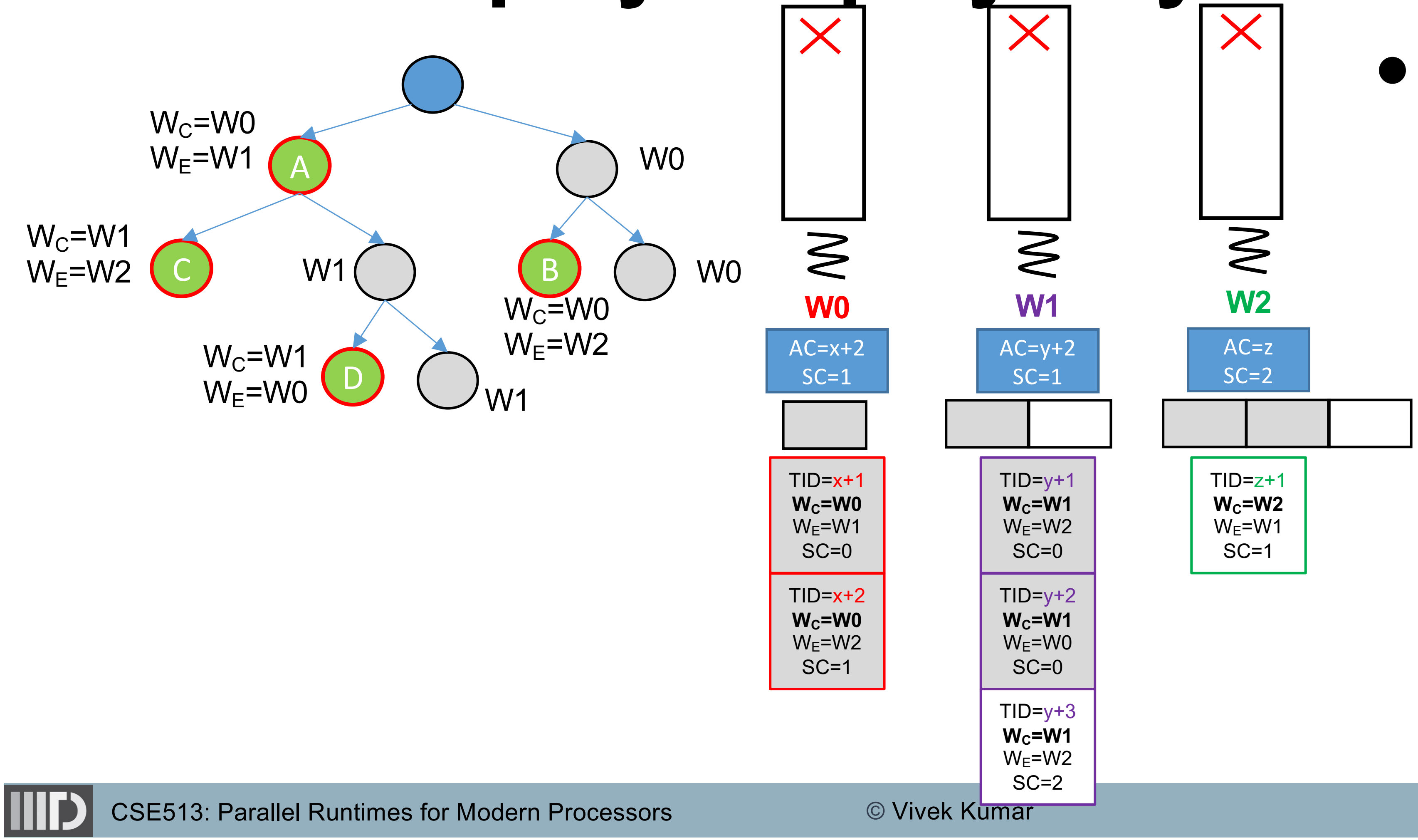
TID=y+2
W_C=W1
W_E=W0
SC=0

TID=y+3
W_C=W1
W_E=W2
SC=2

W2: AC=z, SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

Trace & Replay: Replay Async



W0
AC=x+2
SC=1

TID=x+1
W_C=W0
W_E=W1
SC=0

TID=x+2
W_C=W0
W_E=W2
SC=1

W1
AC=y+2
SC=1

TID=y+1
W_C=W1
W_E=W2
SC=0

TID=y+2
W_C=W1
W_E=W0
SC=0

TID=y+3
W_C=W1
W_E=W2
SC=2

W2
AC=z
SC=2

TID=z+1
W_C=W2
W_E=W1
SC=1

- Each worker would continue its execution until completion by using the tasks transferred by the victim instead of they themselves performing the steal operations

Reading Materials

- I am not providing any reading material on this topic, as the lecture slides should be sufficient

Next Lecture

- Mid semester review