Early Release: Friend or Foe?

Travis Skare
Christos Kozyrakis

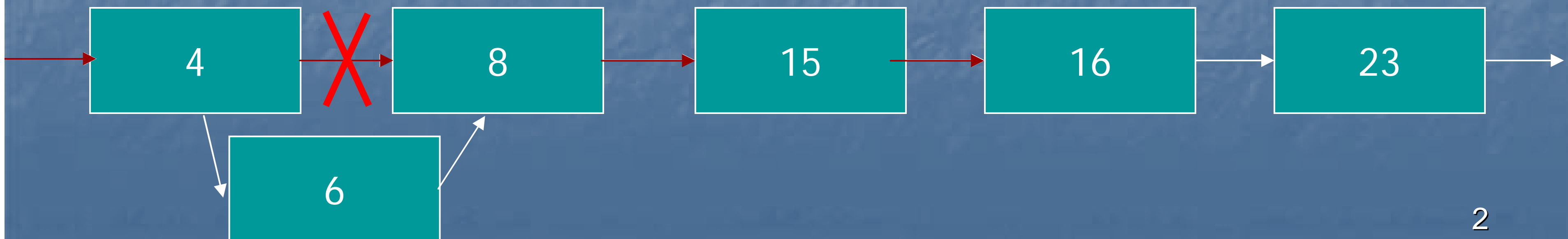
WTW 2006

Computer Systems Lab
Stanford University
http://tcc.stanford.edu

Motivation for Early Release

- Consider multiple transactions operating on a long singly-linked list
- A write to the head of the list can cause a transaction working towards the end to abort.

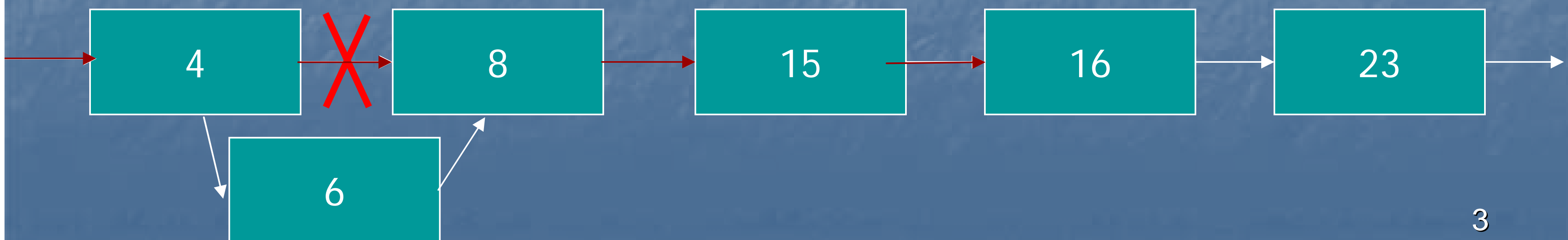
Violation



Early Release

- Early release enables transactions to clear individual elements from their read sets at any time before commit.
- Other transactions writing to the address in the future will not cause a violation.

No Violation



Early Release: Benefits

- Reduces violations
- Missing an opportunity for release does not affect original program semantics
  - A programmer can write an entire program, then go back and implement ER later to increase performance
  - Similar to the process of fine-tuning transactions for speed increases

Early Release: Drawbacks

- Added programming complexity—similar in many cases to fine-grained locking
  - Missing a Release() doesn't cause race conditions…
  - But adding too many can break correctness
  - Could be used as a compiler optimization
- Possible implementation overhead
  - Further complications if release granularity differs from word length (e.g., cache-line granularity)

This Study

- Will implementing Early Release be beneficial inside a collection of data structures?
  - Linked List
  - Hashtable
  - AVL Tree
  - B-Tree
  - Array-based heap (priority queue)
- Variety of work sizes, benchmark settings
- Code chosen to be beneficial to ER
  - If no gain here, unlikely to find gain in “real” apps.

Methodology

- Stanford TCC System
- 1-32 Processors
- Variable workload in between transactions
  - Kept small in results
- 30% read, 35% add, 35% remove
- 6,000 element pre-population
  - Some affinity for keys that will be added/removed
- 8-byte random keys, integer data

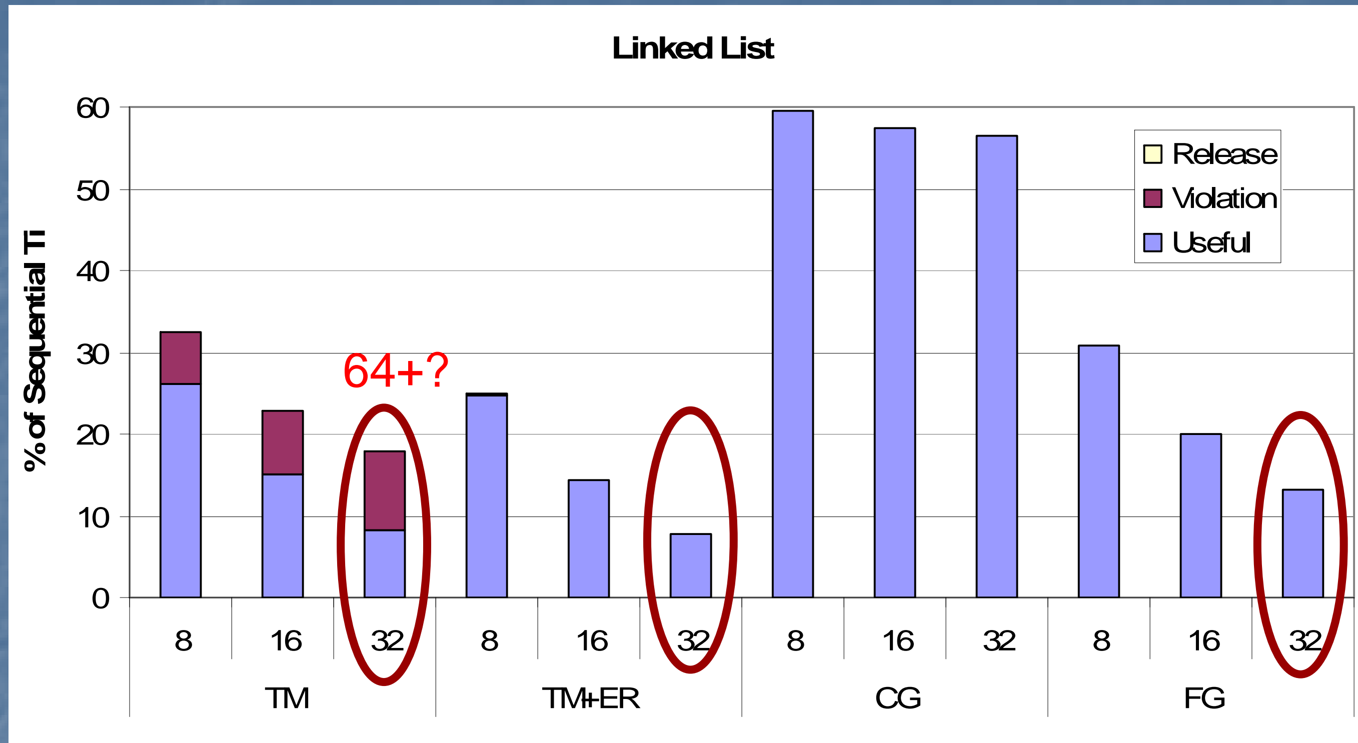
Linked-List Example (FG)

```
int List_Insert_FineGrain(LinkedList *list, string searchKey, int data){
        ListNode *insert = CreateNode(search, data);
        ListNode *prev = list->head, *cur=prev->next;

        Lock(list->head->lock);
        while(cur!=NULL){
                Lock(cur->lock);     // hand-over-hand locking (1)
                if(searchKey<=cur->key){
                        insert->next=cur;
                        prev->next=insert;
                        Unlock(prev->lock);
                        Unlock(cur->lock);
                        return 1;
                }
                Unlock(prev->lock); // hand-over-hand locking (2)
                prev=cur;
                cur=cur->next;
        }
        insert->next=NULL;
        prev->next=insert;
        Unlock(prev->lock);     // release last lock held
        return 1;
}
```

Linked-List Examples (TM/ER)

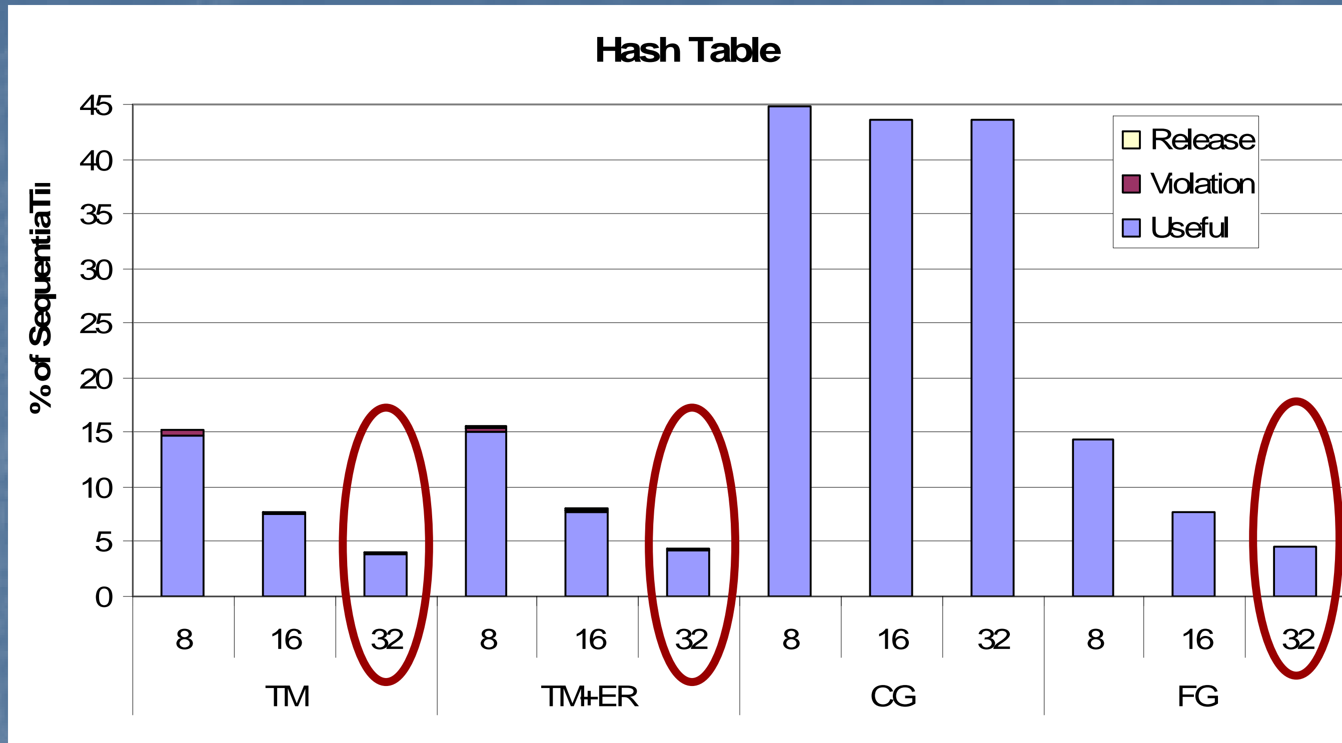
```
int List_Insert_EarlyR(LinkedList *list, string searchKey, int data){
        ListNode *insert = CreateNode(search, data);
        ListNode *prev=list->head, *cur=prev->next;

        while(cur!=NULL){
                // &prev->next has RS bits set by access ("Lock")
                if(searchKey<=cur->key){
                        insert->next=cur;
                        prev->next=insert;
                        TCC_Release(&prev->next);
                        TCC_Release(&insert->next);
                        return 1;
                }
                TCC_Release(&prev->next); // release unused element
                                          // compare to Unlock()
                prev=cur;
                cur=cur->next;
        }
        insert->next=NULL;
        prev->next=insert;
        TCC_Release(&prev->next);       // compare to Unlock
        TCC_Release(&insert->next);     // but would be correct without ER
        return 1;
}
```

Results: Linked List



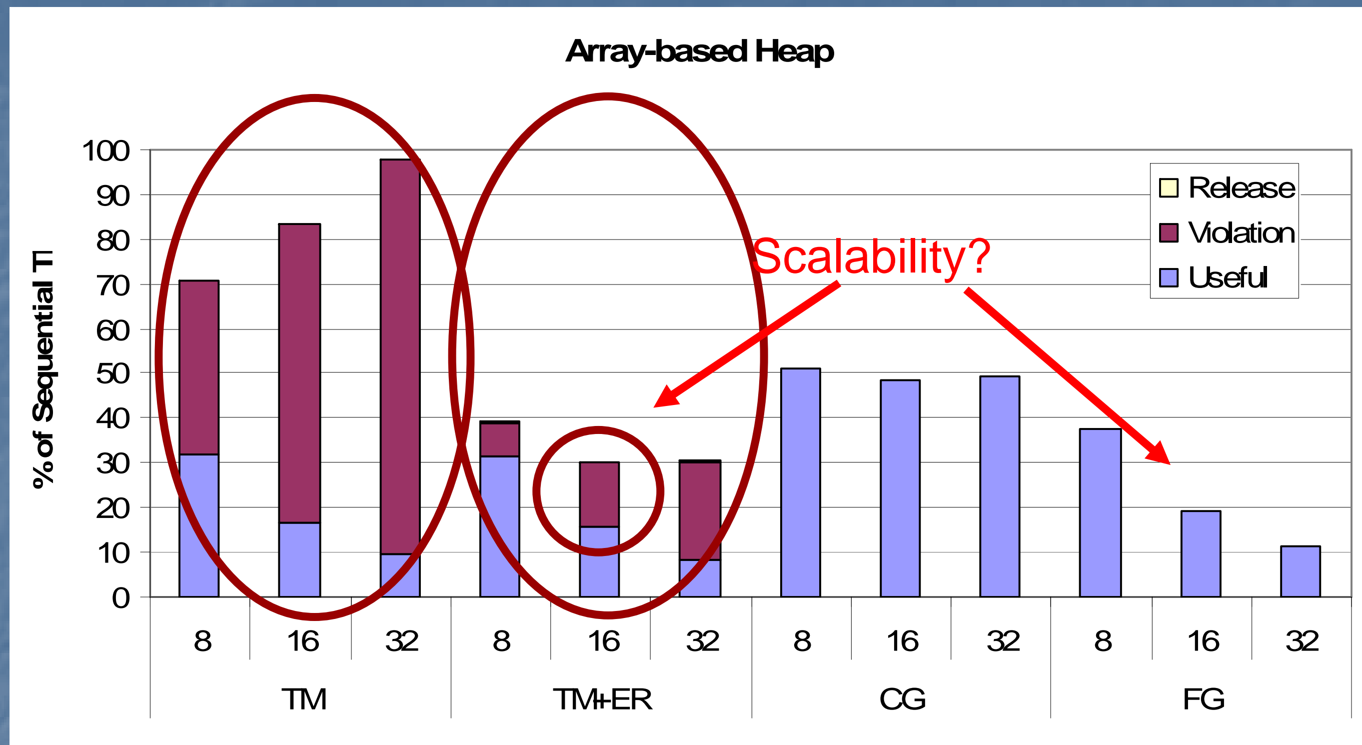
- "Sequential" data structure – single point of entry, single path through data
- ER does help here – beats out even FG locking due to lock/unlock overheads

Results: Hashtable



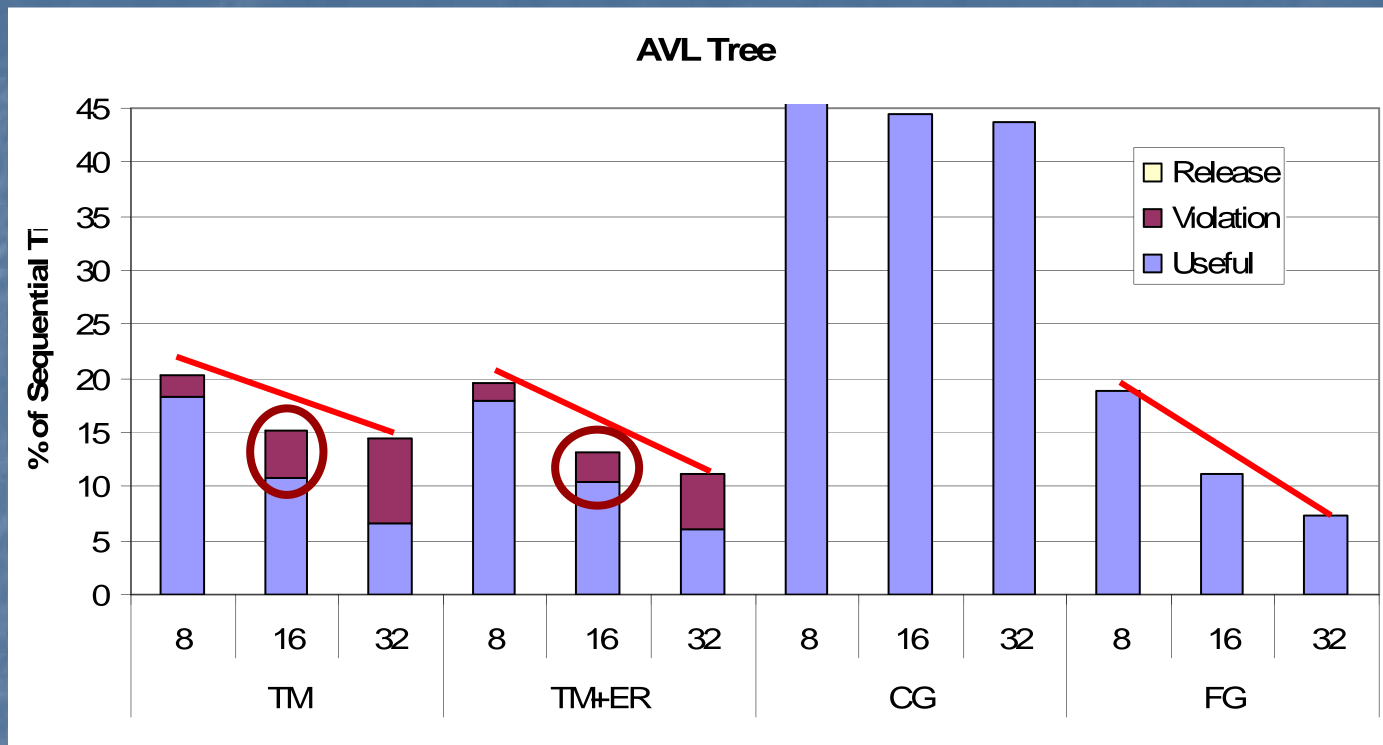
- Most parallelizable data structure – statistically transactions operate on different buckets
  - 256 Buckets used in trials
- ER rarely helps here: "naïve" TM approach is even ~1% faster and rivals FG code

Results: Array-based heap



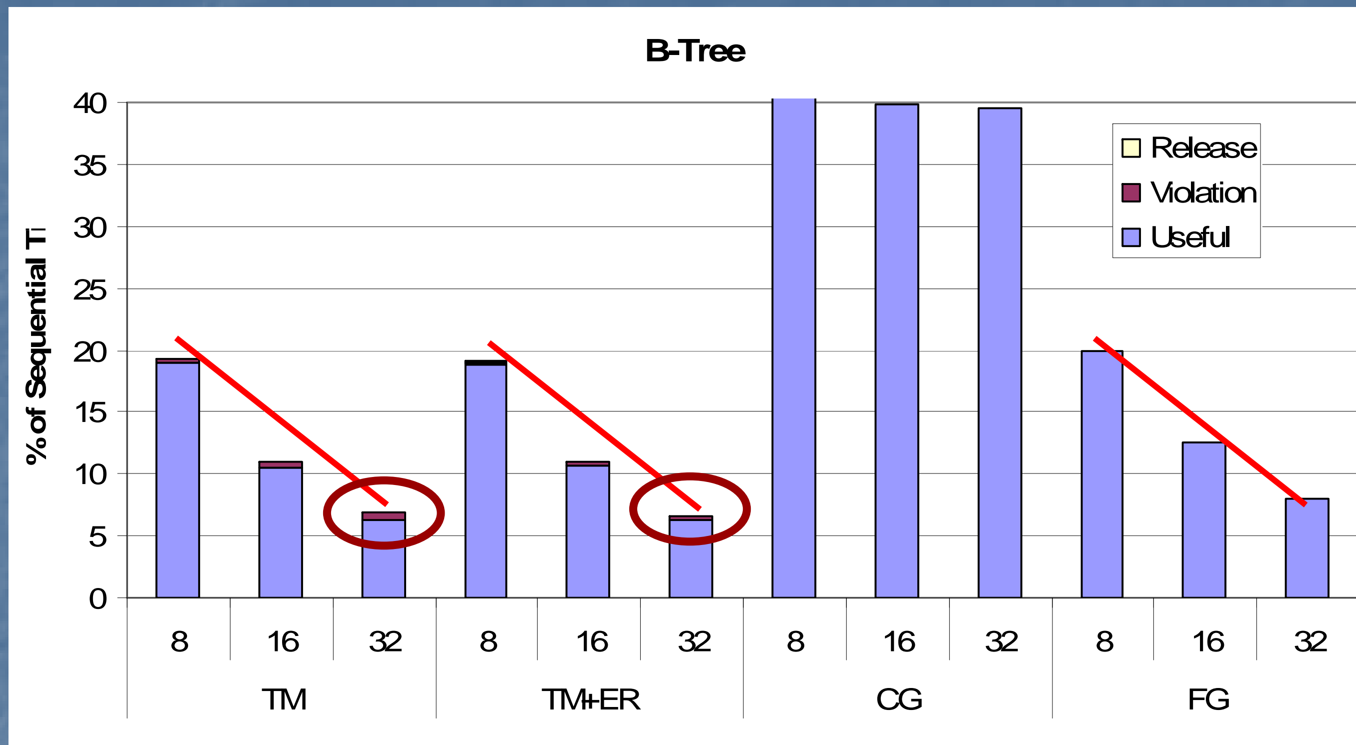
- Naïve implementation (though concurrent ones exist)
- High contention over a few elements
- Early release enhances system scalability
- Violations still occur (bubble-up, etc)

Results: AVL Tree



- There are still violations in the ER case
  - Cannot use ER when balancing the tree, etc.
- ER does show some benefit, especially in scalability
- For less stressful workloads, ER not so beneficial.

Results: B-Tree



- Very Parallelizable
- We still see some violations with splitting, rotations
  - Cannot use ER in these cases
  - Not many violations to reduce from the TM case

Conclusions

- Studied effects of Early Release on five structures
- No performance boost for parallel structures
  - Hashtable, Trees
  - Should generalize to most user-level application code
- There are applications where ER has advantages
  - Heap, rough performance counters, etc
  - Scalability
  - But programmer could use better structures, nesting, etc.
- Also consider programming complexity