Early Release: Friend or Foe?

Travis Skare and Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{travissk, kozyraki}@stanford.edu

1 Introduction

Transactional Memory (TM) [9] has the potential to simplify concurrency management by supporting parallel tasks (transactions) that appear to execute atomically and in isolation. There is already a significant body of work on programming language constructs for transactional memory [7, 6, 5, 3, 1, 2]. Nevertheless, there still exists little consensus on several constructs, particularly those motivated by performance optimizations.

In this paper, we study a set of data structure algorithms to evaluate the ease-of-use and performance benefits of the *early release (ER)* construct [8, 5]. Early release allows a transaction to remove a data address from its transactional read-set long before it commits. Once an address has been released, other transactions can write to this address without generating a conflict with the releasing transaction. The programmer or a compiler must guarantee that early release of an address is safe: removing the address from the read-set should not violate the overall application atomicity and consistency. The tradeoff with early release is obvious: on one hand, removing addresses from the read-set reduces the probability of conflicts that incur expensive long stalls or rollbacks. On the other hand, there is an additional burden to guarantee that early release is safe for a particular address, regardless of any other code that may be executing in parallel. Therefore, if a programmer manually applies early release, she must be extremely careful about when, where, and with which address early release is used.

Previous studies have used a single data structure algorithm (linked list) to conclude that various forms of early release can be a particularly useful programming construct for performance optimization of transactional programs [8, 4]. In this paper, we study five data structure algorithms and rewrite their transactional memory versions to use early release. Our observations are as follows: from the point of view of performance, early release provides a significant benefit only for highly sequential data structures such as a linked list or array-based heap. For concurrent data structures such as hash tables and trees (AVL trees and B-trees), there is no significant performance advantage from early release, even for write-intensive workloads with significant rollback penalties. From the ease-of-use point of view, we demonstrate with code examples that the complexity of manually using early release is often similar to that of using fine-grain locks. Therefore, unlike previous work, our results suggest that early release is not a particularly useful construct for user-level programming with transactional memory systems. Of course, early release may still be useful to an optimizing compiler that is able to automatically identify when it is safe and profitable to use in transactional code.

2 Methodology

We ran transaction-based code on an execution-driven simulator for a CMP that follows the TCC architecture [11]. TCC provides transactional memory using lazy conflict detection to provide non-blocking guarantees. Read-set and write-set tracking is at word granularity. The CMP includes up to 32 cores with private L1 caches (32 KBytes, 1-cycle access) and private L2 caches (256 KBytes, 12-cycle access). The read-set and write-set of the transactions in the studied algorithms fit in the processor caches, hence there is no overhead for overflows and virtualization. The processors communicate over a 16-byte, split-transaction bus. All non-memory instructions in our simulator have a CPI of one, but we model all details in the memory hierarchy for loads and stores, including inter-processor communication.

We added an early release instruction to the base TCC model. The instruction takes a word address and removes it from the transaction read-set. If the word is in the L1 cache, the early release instruction executes in one clock cycle, otherwise it

takes 12 cycles. TCC tracks read-sets at word granularity which helps with the ER implementation. However, early release is difficult to implement consistently in other hardware TM systems that use cache line granularity. In such systems, since an early release instruction provides a word address, it is not safe to release the entire cache line. Alternatively, we can define an early release instruction that releases an address range. Since the range may not always be aligned to cache line boundaries, TM systems that track state at cache line granularity would still experience difficulties.

We used a number of data structure algorithms for our evaluation (linked list, array-based heap, hash table, AVL tree, and B-tree of degree 5). Such data structures are interesting because they include significant parallelism, they are mostly pointer-based, they are regular enough for a programmer to master, and they can be used to create workloads with varying conflict frequencies or transaction sizes. While transactional versions for most of these data structures have been presented before, we contribute new versions with early release. For each data structure, we constructed the following benchmark code. First, we prepopulate the data structure with 6,000 elements [1]. For all structures but the heap, the key for the elements is a randomly-generated 8-character string. For the heap, the keys are randomly generated integers. Then, we perform a series of data structure accesses. To create workloads with frequent conflicts that favor early release, we use the following mix of accesses: 35% insertions of new elements, 35% deletions of existing elements (after searching for them), and 30% reads of elements (after searching for them). A transaction includes one access followed by 400 cycles of work on the retrieved data. The amount of work per transaction makes for a high rollback penalty, further benefiting early release.

We coded our benchmarks in C using a simple inlined API to define transaction boundaries and early release of addresses. We also coded the same benchmarks using both coarse-grain (CG) and fine-grain (FG) locks for comparison purposes. The coarse-grain case uses a single lock during the data structure access, but releases the lock during the additional work. The fine-grained case uses per-node locks to expose more concurrency in data structure accesses. The lock-based code ran on a version of the simulator that has the same resources as the TCC version but uses regular cache coherence and multiprocessor synchronization. For both transactional and non-transactional code, we ran simulations using 1 to 32 processors. For brevity, we present the results for 8, 16, and 32 processors. Larger processor counts generate the highest conflict frequency, but including some smaller counts provides insights into scaling. Results are presented in Figure 1 as execution time, normalized to sequential execution (represented by 100%). Lower bars are better. We break down execution time to useful (regular instructions and cache stalls), violation (time wasted on transactions that rollback), and overhead due to early release (additional instructions). We found that the overhead of early release instructions is insignificant for all benchmarks (less than 1%), hence the corresponding bar is practically invisible in all cases.

Even though our results are measured on a particular hardware TM system, we believe that the conclusions can be generalized. While other hardware or software systems may have higher rollback overheads compared to TCC, we have artificially increased the cost of rollbacks by introducing a significant amount of work per data structure access. In a software TM system, one could call early release once per object instead of once per word, hence reducing the overhead due to additional instructions [8]. However, we found that no workload exhibits significant overhead due to the early release instructions. The coding complexity arguments are equally applicable for all TM systems regardless of the exact API used. On the other hand, if early release is applied automatically by a compiler, there is no coding complexity visible to the programmer.

3 Linked List

The linked list data-structure is not optimal for parallel accesses, as it arranges elements in a single list and search takes $O(N)$ steps to access. The coarse-grain lock (CG) code for linked list accesses (not shown) simply adds a set of `Lock` and `Unlock` statements around the sequential code. Similarly, the original TM code (not shown) simply adds `begin_xaction` and `end_xaction` around the sequential access code and the additional work per access. The TM and CG code is easy to write given the sequential code.

Figures 2 and 3 show the code for linked list insertion with fine-grain lock (FG) and TM with early release respectively (TM+ER). The code for deletions is similar. The FG code holds a lock on the current node as it scans the list to ensure correct insertion. Locks are acquired and released in a careful *hand-over-hand* process to make sure that there is no point at which no lock is held (current and next locks overlap). The TM+ER code is similar, but uses `Release` instead of `Unlock` and there is no need for `Lock` statements (addresses are inserted in the read-set and write-set automatically on loads and stores)[2]. Overall, the complexity of developing FG and TM+ER code is similar. Misplacing or misusing a `Release` statement is

[1] We also performed experiments with larger pre-populations (e.g., 60,000 elements). The results show similar trends, so we omit them for brevity.

[2] The `begin_xaction` and `end_xaction` statements are outside of the insert function. They also enclose the code for the extra work per element.

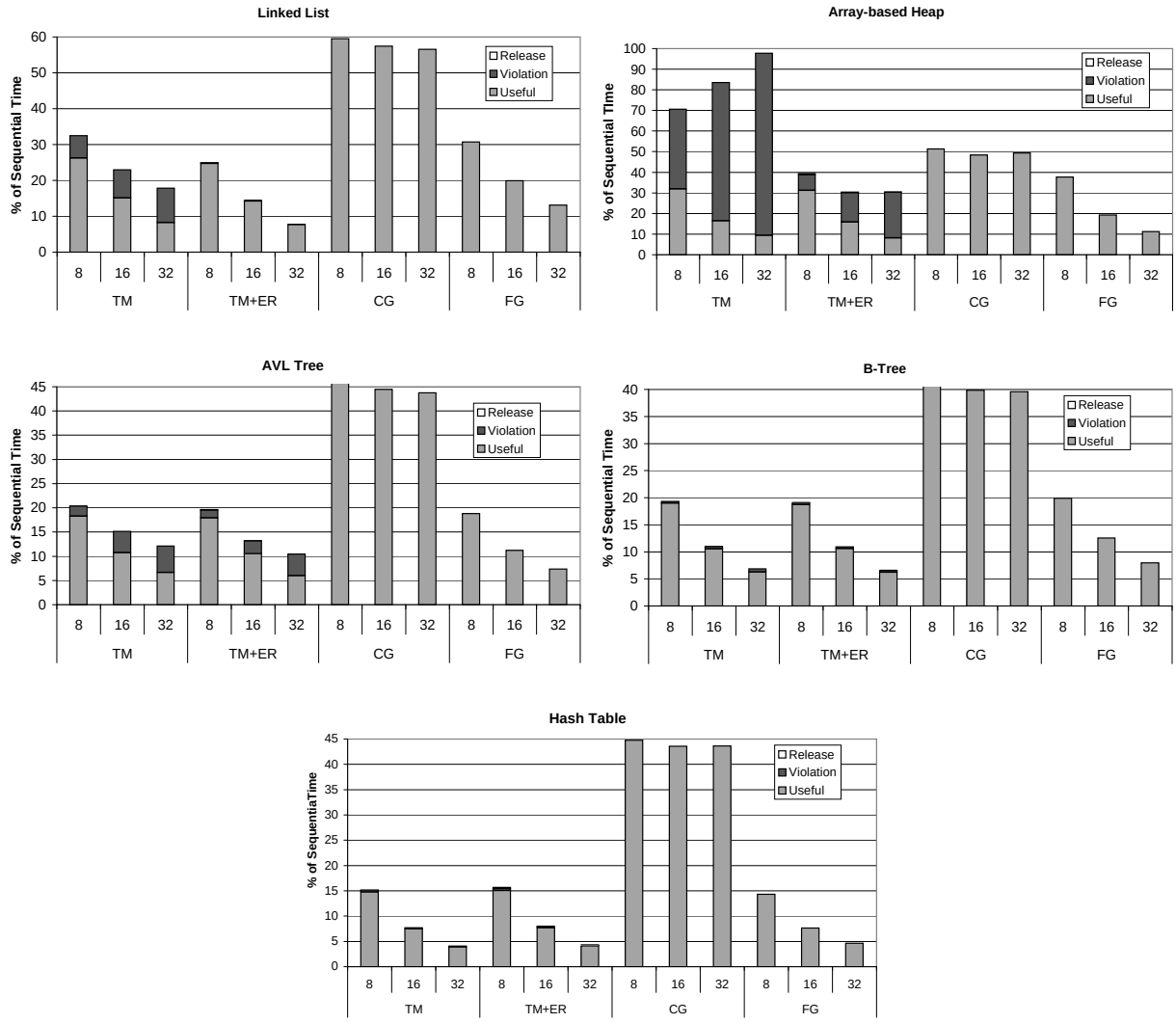

Figure 1: Execution time for the five data structures workloads for different approaches: transactions (TM), early release (TM+ER), coarse-grain locks (CG), and fine-grain locks (FG). The configuration is 8/16/32 processors, 400 cycles of work per access, and 6,000 elements per data structure before timing starts.

as easy as misplacing or misusing an `Unlock`. Unlike with locks, forgetting an early release statement has no correctness implications. However, having an extra one, using the wrong address, or using it too early can lead to incorrect code.

As pointed out by an anonymous reviewer, "the early release code does not maintain transactional consistency if a search is performed on a linked list and the key is not found, since this leaves the entire list absent from the read set. Non-serializability can result if there are concurrent insertions. For example, thread A checks list 1 for X, and inserts Y in list 2 if not found; thread B checks list 2 for Y, and inserts X in list 1 if not found. Transactional semantics say that either X or Y should be inserted. However, early release allows both X and Y to end up in the lists. Both the early release and fine-grained lock version are thus not *composable*." This observation provides further evidence on the dangers of using early release in a manual manner. It also indicates that any automated use of early release by a compiler cannot be applied as a local optimization. The global atomicity behavior of the application must be taken into account. Hence, early release may be difficult to use within library code.

Figure 1 shows the execution time of the four versions of the linked list code. The CG code allows overlapping of the additional per-access work, but no concurrency on the list. The FG code leads to 2.9x better performance in the 16-processor case. Since threads operate on different parts of the long list, they only occasionally block each other when one thread

inserts as the other tries to scan through. Nevertheless, FG is limited by the instruction overhead and cache misses on lock acquires and releases. The TM code suffers from frequent violations. Transactions that scan towards the end of the list are likely to be rolled back by insertions or deletions towards the front as all the list pointers end up in their read-set. Even so, TM is 2.5x faster than CG code in the 16-processor case, which has similar complexity, and is only marginally slower than FG code. Early release allows transactions to have the minimal possible read-set (two elements) and leads to the best performance. TM+ER eliminates violation overhead and offers a 1.5x speedup over simple TM. Looking at how the two TM versions scale, we begin to see higher violation overhead in the 32-processor case for TM. Hence, we surmise that early release would become increasingly important as we scale TM to larger processor counts for data-structures like linked lists. Note, however, that this is a best case scenario for TM+ER. If the workload is less write-intensive, conflicts are less frequent and less expensive, and the advantage of TM+ER over ER is quickly reduced.

4 Array-Based Heap

Due to space reasons, we do not present any of the code for the array-based heap. The heap exhibits the properties of the linked list taken to an extreme. Again, elements are arranged in a single sequence, but now we also have frequent element swapping throughout the heap on updates. Due to these bubble-up operations, violations are very frequent for the transactional versions with such a write-intensive workload. Just like the linked list case, the FG and TM+ER code for the heap attempts to hold a lock for two elements or include in the read-set two elements. The two algorithms have similar structure and almost identical complexity.

Figure 1 shows the execution time of the four versions of the heap code. Again, for a highly sequential structure with a write-intensive workload, TM+ER leads to significant performance benefits over TM (2.8x with 16 processors). Again, a workload with less insertions or deletions exhibits smaller differences. Nevertheless, FG is actually 50% faster than TM+ER. Early release eliminates many, but not all, violations for heap updates, which are very write-intenstive. Notice that for both the linked list and the heap, the maximum speedup we achieve with the 16-processor CMP is less than 7. The two data-structures are not well-suited for parallel accesses, particularly as we scale the number of processors in the system. Excluding the FG code, no other version scales well with more processors.

5 AVL Tree

AVL trees are balanced binary trees that search in $O(logN)$ time. Multiple threads can operate in parallel on different branches. Significant interference is only observed on rotations for rebalancing. Even in that case, we typically have to rotate only a sub-tree, so most threads are not affected. The CG and TM codes are again simple updates from the sequential code. The TM has a small read-set in most cases, as it only touches a single branch.

Figures 4 and 5 show the code for AVL tree insertion with fine-grain lock (FG) and TM with early release (TM+ER). The code for deletions is similar. The FG lock holds a lock to the immediate parent of the point of insertion. As in the case of the linked list, locks are acquired and released in an overlapped, hand-over-hand manner. If a rotation is necessary, a coarser-grain lock is acquired for the whole sub-tree. The TM+ER releases all nodes but the current parent as it scans down the branch using again an overlapped release process. Again, the FG and TM+ER version have very similar complexity.

Figure 1 shows the execution time of the four versions of the AVL tree code. Again, CG can only overlap additional work; all operations to the tree are serialized. The other three versions of the code, TM, FG, and TM+ER, perform and scale similarly. Despite the overhead of lock acquisition, FG has a performance advantage over the TM versions. Even though early release does eliminate some rollbacks, it does not provide a significant advantage over TM (less than 15% improvement is observed). Most rollbacks are due to subtree rotations that are difficult to avoid in both TM and TM+ER.

6 B-Tree and Hash Table

Due to space limitations, we do not present any of the code for the B-tree or the hash table. Much like the AVL tree, the B-tree allows for fast searches and concurrency across branches with minimum interference. The B-tree code is similar to the AVL tree in all cases, but properly adjusted to the degree and nature of the tree. The CG and TM code are trivial while the FG and TM+ER code requires careful locking or releasing as we scan through the tree. Figure 1 shows that the B-tree performance results are similar to the AVL results. In this case, rollbacks due to rotations are much less frequent, as the balancing requirement is relaxed and the tree nodes hold multiple pieces of data. Due to the small number of violations, we do not notice a visible performance advantage for early release. Interestingly, the TM code is even faster than the FG code

that suffers from lock acquisition overheads, while the TM code does not experience significant overheads due to violations.

The hash table is the most concurrent data structure of all in the group we studied. Searches are very fast and interference is unlikely. As there are 256 bins, two threads/transactions rarely work on the same bin. Again, the CG and TM code is trivial. In this case, however, the FG code is also simple as we only lock a bin at a time, as opposed to the individual elements within the bin (as in the linked list case). Figure 1 shows that TM, TM+ER, and FG perform nearly identically. Early release does not help, as we rarely have two transactions working on the same linked list.

7 Conclusions

The results in Figure 1 show that even for the very write-intensive workloads we studied, early release provides insignificant performance improvement (or no improvement at all) over simple, coarse-grain transactions for the data structures that scale well in parallel systems (trees and hash tables). On the other hand, for linear data-structures like linked-lists and array-based heaps, early release can significantly reduce the overhead due to violations. We have also shown through specific code examples that the complexity of early release can be similar to that of fine-grain locks. While missing early release statements do not affect correctness, a misplaced or additional early release is as bad as a misplaced or missing lock/unlock statement and may lead to atomicity breaches. Hence, instead of applying early release on linear data structures like linked-lists and heaps, the programmer is probably better off switching to a more concurrent data structure and using simple transactions.

Overall, our analysis suggests that early release is not a particularly useful construct for user-level programming with transactional memory. While our conclusions apply only to the workloads we studied, they suggest that the added programming complexity is not worth the limited performance boost from early release. Moreover, if it is difficult to use early release and extract performance benefits with regular data structure code, it is unlikely that it will be sufficiently useful with more complicated code. Coarse-grain transactions, potentially with nesting support [10], are sufficient to achieve good performance with the simple code that programmers expect from transactional memory.

On the other hand, there are certain cases where early release can provide significant performance advantages, as shown by the linked-list and heap workloads. Hence, early release may still be a useful to an optimizing compiler that is able to automatically identify when it is safe and profitable to use in transactional code.

References

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, 2005.

[2] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[4] C. Cole and M. Herlihy. Snapshots and Software Transactional Memory. *Science of Computer Programming*, 58(3):310–324, Dec. 2005.

[5] Cray. *Chapel Specification*. February 2005.

[6] T. Harris. Exceptions and side-effects in atomic blocks. In *2004 PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[7] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[8] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, July 2003. ACM Press.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[10] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, June 2006.

[11] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–74, Washington, DC, USA, September 2005. IEEE Computer Society.

```
1  int List_Insert_FineGrain(LinkedList *list, string searchKey, int data){
2          ListNode *insert = CreateNode(search, data);
3          ListNode *prev = list->head, *cur=prev->next;
4
5          Lock(list->head->lock);
6          while(cur!=NULL){
7                  Lock(cur->lock);
8                  if(searchKey<=cur->key){
9                          insert->next=cur;
10                         prev->next=insert;
11                         Unlock(prev->lock);
12                         Unlock(cur->lock);
13                         return 1;
14                 }
15                 Unlock(prev->lock);
16                 prev=cur;
17                 cur=cur->next;
18         }
19         insert->next=NULL;
20         prev->next=insert;
21         Unlock(prev->lock);
22         return 1;
23 }
```

Figure 2: Fine-grain (FG) locking code for linked list insert.

```
1  int List_Insert_TCC_EarlyRelease(LinkedList *list, string searchKey, int data){
2          ListNode *insert = CreateNode(search, data);
3          ListNode *prev=list->head, *cur=prev->next;
4
5          while(cur!=NULL){
6                  if(searchKey<=cur->key){
7                          insert->next=cur;
8                          prev->next=insert;
9                          Release(&prev->next);
10                         Release(&insert->next);
11                         return 1;
12                 }
13                 Release(&prev->next);
14                 prev=cur;
15                 cur=cur->next;
16         }
17         insert->next=NULL;
18         prev->next=insert;
19         Release(&prev->next);
20         Release(&insert->next);
21         return 1;
22 }
```

Figure 3: TM with early (TM+ER) for linked list insert.

```
1  AVLTree_Insert_FG(AvlTree *tree, string key, int data){
2          Lock(tree->root);
3          AVLTree_Insert_FG_Helper(tree->root, key, data);
4  }
5
6  AVLTree_Insert_FG_Helper(Node *current, string key, int data){
7          if(key==current->key){
8                  Unlock(current->Lock);
9                  return;
10         } else if(key<current->key){
11                 if(current->left==NULL){
12                         current->left=CreateNode(key,data);
13                         Unlock(current->lock);
14                         return;
15                 } else{
16                         Node *left=current->left;
17                         Lock(left->lock);
18                         Unlock(current->lock);
19                         AVLTree_Insert_FG_Helper(left,key,data);
20                         Balance(left);
21                 }
22         } else {
23                 if(current->right==NULL){
24                         current->right=CreateNode(key,data);
25                         Unlock(current->lock);
26                         return;
27                 } else{
28                         Node *right=current->right;
29                         Lock(right->lock);
30                         Unlock(current->lock);
31                         AVLTree_Insert_FG_Helper(right,key,data);
32                         Balance(right);
33                 }
34         }
35 }
```

Figure 4: Fine-grain (FG) locking code for AVL tree insert.

```
// Can avoid a wrapper function by calling the inner function directly.
AVLTree_Insert_EarlyRelease(AvlTree *tree, string key, int data){
        AVLTree_Insert_ER_Helper(tree->root, key, data);
}

AVLTree_Insert_ER_Helper(Node *current, string key, int data){
        if(key==current->key){
                return;
        } else if(key<current->key){
                if(current->left==NULL){
                        cur->left=CreateNode(key,data);
                        Release(current->left);
                        return;
                } else{
                        Node *left=current->left;
                        Release(current);
                        AVLTree_Insert_FG_Helper(left,key,data);
                        Balance(left);
                }
        } else {
                if(cur->right==NULL){
                        cur->right=CreateNode(key,data);
                        Release(current->lock);
                        return;
                } else{
                        Node *right=current->right;
                        Release(current);
                        AVLTree_Insert_FG_Helper(right,key,data);
                        Balance(right);
                }
        }
}
```

Figure 5: TM with early release (TM+ER) for AVL tree insert.