Automating Non-Blocking Synchronization In Concurrent Data Abstractions

Jiange Zhang
University of Colorado at
Colorado Springs, CO, USA
jzhang3@uccs.edu

Qing Yi
University of Colorado at
Colorado Springs, CO, USA
qyi@uccs.edu

Damian Dechev
University of Central Florida
Orlando, FL, USA
dechev@cs.ucf.edu

***Abstract*—This paper investigates using compiler technology to automatically convert sequential C++ data abstractions, e.g., queues, stacks, maps, and trees, to concurrent lock-free implementations. By automatically tailoring a number of state-of-the-practice synchronization methods to the underlying sequential implementations of different data structures, our automatically synchronized code can attain performance competitive to that of concurrent data structures manually-written by experts and much better performance than heavier-weight support by software transactional memory (STM).**

I. Introduction

The advent of the multi-core era has brought multi-threaded programming to the mainstream and with it the challenges of managing shared data among the threads. Compared to traditional lock-based mechanisms [1], [2], [3], [4], [5], non-blocking synchronization offers lock-free progress guarantees and better fault tolerance, but at the expense of requiring much more extensive modifications to the sequential code, limiting their wide-spread use. This paper aims to ameliorate this difficulty.

Nowadays lock-free synchronization is programmed either by manually following various construction methods [6], [7], [8], [9], [10], [11], [12], [13], [14] or automatically by using the higher-level interface supported by software transactional memory (STM) [15], [16], [17]. This paper presents an alternative approach to STM, by using source-to-source compiler technology to automatically convert sequential C++ data abstractions, e.g., queues [6], [7], sets [9], [8], [10], and trees [14], [13], [12], to lock-free concurrent implementations, to relieve developers from the error-prone task of low-level concurrent programming.

Our compiler supports the synchronization of a single abstraction and therefore is not as general-purpose as STM, which supports the synchronization of whole software applications. However, a key insight from our work is that by restricting the scope of synchronization to a single self-contained abstraction and thereby selecting the best synchronization strategy to tailor to the underlying sequential implementation, runtime overhead of synchronization can be significantly reduced, resulting in much better performance scalability than using existing heavier-weight STM implementations to support all synchronization needs. We show that by automatically weaving synchronization schemes with sequential implementations of data structures, many efficient lock-free data structures can be made readily available, with performance competitive to that of the manually crafted ones by experts.

The key technical difference between our work and existing STM research is that we support multiple synchronization strategies and automatically select the best strategy for each piece of code at compile time. No existing compiler for STM supports automatic tailoring of synchronizations to what is needed by different pieces of code. The compiler algorithms we present are the first to do this. While we mostly rely on standard compiler techniques (e.g., pointer and data-flow analysis), the problem formulations and solution strategies do not yet exist in any compilers.

Our programming interface is easy to use as it does not require the user to declare anything (in contrast, STM requires all shared data references to be fully wrapped inside special regions called *transactions* [15], [16]). However, our compiler does require that shared data, together with all their operations, must be encapsulated into a single C++ class, which contains all the sequential implementations as the baseline for synchronization. The goal of our compiler-driven approach is to close the performance gap between using STM to support all synchronization needs of an application vs manually crafting much lighter weight (and thus more efficient) synchronizations for each individual shared concurrent data structure [5], [18], [11]. The performance attained by our auto-generated code can be limited by their original sequential implementations, e.g., by whether a linked list or an array is used to organize the data. To quantify such impact, we have experimented with using both data layout schemes for selected data structures. Overall, our technical contributions include:

- We present a cohesive strategy to effectively adapt and combine state-of-the-practice synchronization techniques, to automatically convert sequential data abstractions into concurrent lock-free ones.
- We present formulations and algorithms to automatically classify shared data based on how they are referenced in the sequential code and to automatically tailor their synchronizations to the varying characteristics of their concurrent operations.

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00074

```
 1 template <class T> SinglyLinkedList {
 2  Node<T>* head; Node<T>* tail; unsigned count;
 3
 4  void pushback(const T& o){
 5    Node<T>* e=new Node<T> (o); ++count;
 6    if (tail==0) {head=tail=e;} else {tail->next=e; tail=e;} }
 7
 8  bool is_empty(){ return count == 0; }
 9
10  bool lookup(const T& t) {
11    Node<T> *e = head; while (e!=0 && e->content!=t) e = e->next;
12    return (e!=0); }
13  ...  };
```

Fig. 1: Example: a sequential singly-linked list

- We have implemented a prototype source-to-source compiler to automatically support the lock-free synchronization of eight data structures, including queues, stacks, hash maps, and trees. We show that the performance of our auto-generated implementations are competitive against existing manually crafted implementations by experts and better than auto-generated implementations by using heavier-weight Software Transactional Memory.

Our prototype compiler, together with all the data structures evaluated in this paper, will be released as part of the source code release of the POET language [19].

The rest of the paper is organized as follows. Section II details the synchronizations supported by our compiler. Section III summarizes our overall compilation strategies. Section IV presents additional details of the compile-time formulations. Section V presents experimental results. Section VI discusses related work. Section VII concludes.

II. Details Of Synchronization

Given n concurrent operations op_1, op_2, ..., op_n, they are *linearizable* and thus properly synchronized, if every time they are evaluated, there exists a sequential ordering of the same operations that maintains real-time ordering and produces the same results [20]. The synchronization is non-blocking if the failure of any operation never disrupts the completion of others. It is additionally lock-free if some operation is always guaranteed to make progress.

Our compiler automatically combines two widely-used non-blocking synchronization techniques: read-copy-update [21] (RCU) and read-log-update [22] (RLU), into four overall schemes: single-RCU, single-RCU+RLU, RLU-only, and multi-RCU+RLU, each scheme incrementally extending the previous ones to be more sophisticated. Single-word compare and swap (CAS) and total store ordering are the only hardware supports required. All schemes guarantee lock-free progress in that if multiple operations try to modify an abstraction via CAS at the same time, at least one of them will succeed, while the others restart. The RLU-only and multi-RCU+RLU schemes allow independent modifications to move forward concurrently, while the lighter weight single-RCU/single-RCU+RLU schemes essentially sequentialize all concurrent modifications. All schemes allow read-only operations to make full progress independent of ongoing modifications. The following details each of these schemes.

```
 1 typedef struct List_state {
 2  Node<T>* head; Node<T>* tail; unsigned count; unsigned id;
 3  typedef struct ModLog {
 4     Node<T>** addr; /*address being modified*/
 5     Node<T> *old, *new; /*old and new values of addr*/} ModLog;
 6  atomic<vector<ModLog>*> modlogs;
 7
 8  void copy(List_state *that) {id=that->id+1;...copy members...}
 9
10  void new_modlog(Node<T>** _addr,Node<T>* _old,Node<T>* _new)
11   { check_conflict(this,_addr,_old);
12     vector<ModLog> *c = load_modlog(modlogs);
13     c->push_back(ModLog(_addr,_old,_new)); }
14
15  void modlog_apply()
16   { vector<ModLog> *c=finalize_modlog(modlogs); if (c==0) return;
17     for (vector<ModLog>::iterator p=c->begin();p!=c->end();p++)
18     { ModLog t = (*p);
19       atomic<Node<T>*>* t1 =
20          reinterpret_cast<atomic<Node<T>*>*>(t.addr);
21       cas_mod(t1,&t.old,t.new,id); /*use weak CAS to modify t1*/
22    } }
23 } List_state;
24
25 atomic<List_state*> state;
```

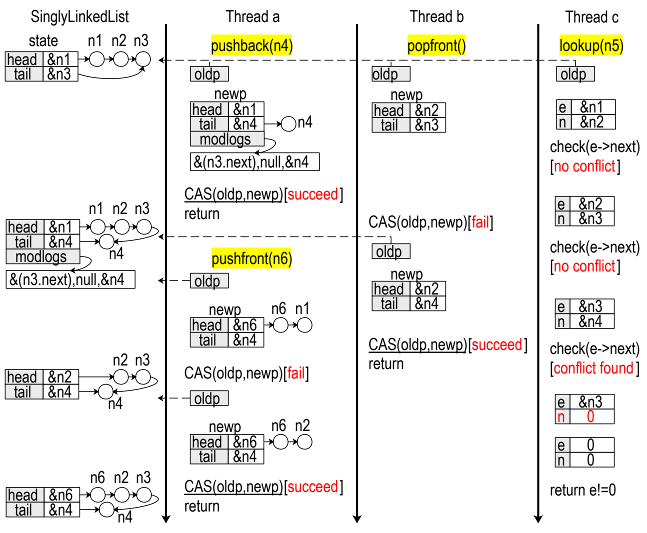
Fig. 2: Relocating data of Figure 1 (text in red is related to RCU synchronization; the rest is related to RLU)

```
 1 void pushback(const T& o) {
 2   Node<T>* e=new Node<T> (o);
 3   SinglyLinkedList_state *oldp=0, *newp=allocate_state();
 4   while (true)
 5   { try{ oldp=load_current_state(MOD_RCU); oldp->modlog_apply();
 6          newp->copy(oldp); ++newp->count;
 7          if (newp->tail==0) {newp->head=newp->tail=e;}
 8          else{ newp->new_modlog(&(newp->tail->next),
 9                                  newp->tail->next,e);
10                newp->tail = e; }
11          if (state.compare_exchange_strong(oldp,newp))
12             { newp->modlog_apply(); break;  }
13        } catch(const ModLogConflict& e){} } }
14
15  bool is_empty()
16  { SinglyLinkedList_state *oldp=load_current_state(READ_RCU);
17    oldp->modlog_apply(); return oldp->count == 0; }
18
19  bool lookup(const T& t)
20  {  SinglyLinkedList_state *oldp=0;
21    while (true)
22    { try{ oldp=load_current_state(READ_RLU); oldp->modlog_apply();
23           Node<T> *e = oldp->head;
24           while (e!=0 && e->content!=t)
25           { Node<T>* n=e->next; check_conflict(oldp,&e->next,n);
26             e=n;}
27           return e != 0;
28         } catch(const ModLogConflict& e){} } };
```

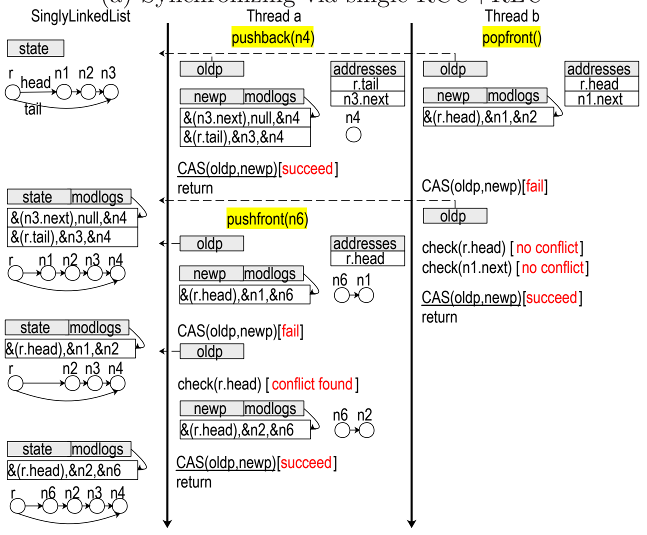
Fig. 3: Example: synchronizing a singly-linked list (text in red is related to RCU synchronization)

a) **single-RCU:** where all shared data of an abstraction are considered a single jointly-accessed group, to be collectively copied and synchronized via the RCU (Read-Copy-Update) scheme, which includes four steps: (1) collect all shared data in the group into a continuous region whose address is stored in an atomic pointer p; (2) at the beginning of each operation f, atomically load the address e stored in p; if f modifies shared data, additionally copy data of e into a new region $e2$ (3) f proceeds as in a sequential setting, except all shared data accesses are redirected through address e (or $e2$ if the data in e has been copied to $e2$); and (4) upon completion, if f has modified data in $e2$, it tries to use $e2$ to replace the content of p, using a single compare-and-swap (CAS): if the CAS succeeds, f returns; otherwise, a conflict is detected, and f restarts by going back to step (2). If f does not modify shared data, it simply completes and returns, as the content of address e is intact, even if other threads have modified the atomic pointer p in the mean time.

To illustrate, Figure 2 shows the new type defined to relocate the three member variables declared at line 2 of the singly-linked list in Figure 1. A shared atomic



(a) Synchronizing via single-RCU+RLU



(b) Synchronizing via RLU-only

Fig. 4: Synchronizing via single-RCU+RLU vs RLU-only

pointer, *state*, is then declared at line 25 to hold the most current address of the relocated data, termed a *RCU object*. Each RCU object is given a unique number (named *id* at line 2 of Figure 2), to track the linear sequence of successively committed RCU Objects whose addresses have been held by the shared atomic pointer. Additional details of synchronization are illustrated in Figure 3 via the red-colored text. Here each operation starts by instantaneously obtaining the most recent address of the *state* atomic pointer, by invoking *load_current_state* at lines 5, 16 and 22. Each read-only operation (e.g., *is_empty* and *lookup*) only has to redirect all the reads through the loaded address (in variable *oldp*). Each modification operation (e.g., *pushback*) first makes a copy of the shared data (line 6), then redirects all modifications to its local copy (lines 7-10), and finally uses the address of the local copy to modify the shared atomic pointer using a CAS at line 11.

b) **Single-RCU+RLU:** where single-RCU is combined with the RLU (Read-Log-Update) scheme to address situations when some internal data of the abstraction cannot be easily copied or are too expensive to copy. A RLU synchronization includes three steps: (1) each operation f saves its shared data modifications as private logs locally, (2) when done with modifications, f tries to publicize its delayed logs to all threads; if the publication succeeds, f uses a sequence of weak atomic updates to apply the logs to physical shared memory before returning; and (3) if f fails to publicize its logs, a conflict is detected, and it starts over by going back to step (1). By enforcing that all threads that observe a set of publicized logs immediately help to complete these logs via weak atomic updates, where concurrent redundant updates are simply ignored, all the publicized logs can be sorted in the shared space and applied atomically in their real-time ordering. To make sure all publicized modifications are observed, all threads also dynamically examine the publicized logs for each shared address they access to detect conflicts in time.

To combine RCU with RLU, our compiler augments each RCU struct with an array of modification logs, e.g., at lines 3-5 of Figure 2. These logs are created and saved locally inside a local copy of the RCU struct, by invoking its *new_modlog* method, e.g., at line 8 of Figure 3. All the delayed logs are finally publicized, e.g., at line 11 of Figure 3, when the local copy is used to replace the shared atomic pointer. Once successfully publicized, these logs are applied as soon as they are observed by all threads, by having these threads immediately invoke the *modlog_apply* method of each newly loaded RCU object to help complete its logged modifications (e.g., at lines 5, 12, 17, and 22 of Figure 3). To ensure each operation observes the new publicized logs in time, a *check_conflict* function is invoked (e.g., at line 25 of Figure 3) to dynamically detect conflicted logs immediately after loading each shared address e and immediately before recording any modification log for e. Upon detecting a conflict, it throws an exception if the ongoing operation modifies shared data. However, if the ongoing operation is read-only, the original value before the conflicted modification is retrieved, to allow the read-only operation to complete using its old data irrespective of the detected conflicts.

Figure 4(a) illustrates how three concurrent threads operating on the singly-linked list in Figure 3 interact with each other when synchronized through this scheme. Here the three dependent modifications by threads (a) and (b) are essentially sequentialized, while the read-only operation by thread (c) is allowed to fully complete using old data irrespective of the ongoing modifications.

c) **RLU-only:** where dynamic conflict detection is used to synchronize each shared address of the abstraction via the RLU scheme, while using the RCU objects only to group related RLU logs and in turn to linearize these groups. Figure 5 illustrate details of this scheme. Here a new local array (declared at line 1) is declared to

```
 1 std::vector<void*> addresses; /*all shared addresses referenced*/
 2 abstraction_state *oldp=0, *newp=allocate_state();
 3 while (true) {
 4  try {
 5   oldp=load_current_state(MOD_RLU);oldp->modlog_apply();
 6   newp->copy(oldp);
 7   ... addresses.push_back(addr);
 8       check_conflict(oldp,addr,*addr);/*is addr out-of-date?*/
 9   ... addresses.push_back(addr);
10       newp->new_modlog(addr,oldv,newv); ...
11   if (newp->modlog_empty()) break; /*no modification*/
12   else if (state.compare_exchange_strong(oldp,newp))
13       { newp->modlog_apply(); break; }
14   else {
15    while (true) {
16      oldp=load_current_state(MOD_RLU);oldp->modlog_apply();
17      newp->copy(oldp);
18      for (std::vector<void*>::const_iterator p = addresses
19            .begin(); p != addresses.end(); ++p)
20         { void* addr = *p;
21            check_conflict(oldp,addr,*addr); /*addr out-of-date?*/}
22      if (state.compare_exchange_strong(oldp,newp))
23         { newp->modlog_apply(); break; } }
24    break; }
25  } catch(const ModLogConflict& e){}
26 }
```

Fig. 5: Synchronizing via RLU only

accumulate all the shared addresses read/modified before each address is checked for conflicts (lines 7-10). At the end of the operation, if no modification log has been created (line 11), the operation has not modified any shared data (hence is read-only) and can simply return. If the operation has modification logs to commit, it tries to publicize the logs using the CAS at line 12. If the CAS fails, the shared RCU atomic pointer must have been modified by another thread. Here the most recent RCU object is re-loaded, and all the shared addresses that have been referenced are re-validated by invoking *check_conflict* before a new CAS is used to re-commit the logs (lines 18-23). If the re-validation fails, an exception is thrown by the conflict detection invocation; otherwise, the operation returns when the next CAS succeeds.

Figure 4(b) illustrate how two concurrent threads modifying a singly-linked list interact with each other when synchronized via RLU-only. Here in spite of two failed CAS attempts, both the pop invocation by thread (b) and the second push invocation by thread (a) were allowed to complete after re-validating the addresses they referenced. So the independent modifications are allowed to move forward concurrently.

d) **Multi-RCU+RLU:** where the shared data are partitioned into multiple disjoint groups, each group independently synchronized via the single-RCU+RLU scheme. To use this scheme, each data item in the abstraction must belong to exactly one of the groups, and each interface function of the abstraction must access data from at most a single group. Since there is no intersection among the independently synchronized groups, no conflict can arise from operations on different groups of data.

e) **Dynamic conflict detection:** For each shared address e synchronized via RLU logs, our dynamic conflict detection code keeps track of the latest version of RLU logs that have modified e, by recording this version number inside a unique wrapper object allocated for e. Additionally, a shared *activity* array is used to map each thread id to a classification of its ongoing activity (e.g., whether its operation is read-only and whether it modifies RCU-synchronized data), and another shared RCU-*modlog* array is used to map the version of each shared RCU object to its RLU logs, so that given an arbitrary version number, all the later committed RLU logs can be readily retrieved. If a thread t invokes conflict detection, the RCU object c currently in use by t and the classification of its ongoing function f are first retrieved. If c is the same as the one held by the shared RCU atomic pointer, no conflict exists; otherwise, the classification of f is examined. If f is a modification operation synchronized via combined RCU+RLU, an exception is thrown, causing f to abort due to its obsolete RCU object. If f is a modification operation synchronized via RLU only, the latest RCU version v that has modified e is retrieved, and an exception is thrown only if v is later than the version of c. If f is a read-only operation, it is allowed to finish by all means, by examining all the RLU logs committed later than c to recover the original value for e before these logs are carried out, and the recovered value is returned to f to continue. Overall, each dynamic invocation of conflict detection is fairly lightweight (involving one or two memory operations) unless it is inside a read-only operation, where the high cost is allowed so that read operations can always complete.

f) **Correctness Guarantee:** We rely on existing literature to argue for the correctness of the standalone RCU and RLU synchronizations [23], [21], [22]. How we combine them do not change how each fundamentally works. In particular, for each group of data protected by a single RCU struct (via the single-RCU or single-RCU+RLU scheme), all concurrent modifications to this group of data are linearized by the use of a single shared RCU object, which can be modified only with a sequence of successful atomic CAS operations. The linear sequence of different RCU objects saved as values of the shared atomic pointer is therefore versioned exactly according to the real-time ordering of the successful CAS operations (the linearization point of the modifications). The RLU logged modifications are made part of this versioning scheme, because the RLU logs inside each RCU object are always carried out immediately after the RCU object is committed, concurrently by each thread that observes the publicized RCU object. To ensure all the committed RLU logs are correctly observed, the version of each shared memory address that has been modified is dynamically traced, each shared memory access is checked for conflicts, and its value is used (considered valid) only if the version number of the value matches that of the RCU struct being used by the accessing thread.

Our RLU-only synchronization uses dynamic conflict detection to synchronize all internal data of an abstraction. Here our strategy differs from that in the literature [22] only in that instead of relying on a real-time clock, we use a shared RCU atomic pointer to group and linearize related RLU logs, Similar to existing work, when concurrent modifications are detected, re-validation of all the shared data accsssed by the conflicting thread (including both the read-set and the modification-set) must be conducted before its modification is allowed to move forward.

Our multi-RCU+RLU scheme is used only when the data of an abstraction can be partitioned into multiple independently accessed groups, where each concurrent operation can access (modify, or read, or both) at most a single group. If two concurrent operations access distinct groups of data, they do not need to be synchronized, since no sharing is present; on the other hand, if they access the same group of data, they are synchronized via the single-RCU+RLU scheme for the involved group, where the correctness is guaranteed by the single-RCU+RLU.

All synchronizations are guaranteed to be correct only if they are implemented correctly. Our compiler-driven approach is advantageous in this case because the same implementation strategies are automatically applied to all data structures. In our experimental evaluation, the correctness of our compiler and its auto-generated code is empirically validated by manually comparing the results of running each workload sequentially vs concurrently and then verifying the equivalence of the results.

III. Overall Strategies

Our compiler serves to automatically apply the synchronization designs in Section II to an existing sequential data abstraction, represented by a self-contained C++ class (with or without template parameters), while guaranteeing correctness through conservativeness — specifically if the compiler cannot use program analysis to guarantee correctness for any piece of the input code, the problematic code piece is deemed ineligible for conversion, and the unsafe operation moved to a private section of the converted abstraction. Our compiler first analyzes and pre-classifies all data references inside the abstraction to select a most appropriate synchronization scheme (single-RCU, single-RCU+RLU, RLU-only, or multi-RCU+RLU). It then tailors the selected scheme to the underlying abstraction implementation through a set of systematic program transformations. Custom lock-free garbage collection [24], [25] and ABA prevention [26], synthesized from scratch, are then inserted to ensure correctness of the final code. The following provide details of these steps.

a) **Pruning of unsafe operations:** A concurrent abstraction must not allow addresses of internal data to escape to the outside, e.g., by being passed as external function parameters or results, because the address can become obsolete at any moment and cause memory corruption. Given an arbitrary abstraction x, all the internal data, including all member variables of x, must be made private so that they can be synchronized exclusively in x. Further, x must be free of function calls that have unknown side effects or have their own internal synchronizations. To guarantee safety, our compiler first prunes all such unsafe operations from the abstraction interface before proceeding to synchronize the remaining operations.

For example, Figure 1 shows a subset of the pruned interface functions of a sequential singly-linked list. Note that while additional operations (e.g., pushfront, popback, is_full) can be added to the interface, some operations, e.g., those inserting or erasing from an arbitrary location in the list, must be eliminated from the concurrent interface because they allow internal addresses to be passed from or returned to the outside. Most C++ data abstractions in principle can be at least partially converted by our compiler, by excluding public functions that are unsafe.

b) **Classification and partitioning of data:** Before determining how to synchronize an abstraction, our compiler needs to classify all data references of the abstraction to be synchronized via RCU or RLU. To this end, it classifies RCU-synchronized data to include all variable references that are never aliased (that is, the data either reside in a unique variable or can be reached only through a unique path of pointers, starting from a unique variable). All the shared data references that can be aliased (that is, the data can be potentially reached through multiple paths of pointer referencing) are classified to be synchronized via RLU. For each data reference classified as RCU-synchronized, a unique pointer chasing path, starting from a unique member variable of the abstraction, is constructed, so that all data on the reference path can be copied if needed in a straightforward fashion. For each data reference classified as RLU-synchronized, a set of member variables of the abstraction are similarly identified, as the only places through which these data can be reached.

If all data referenced in an abstraction are reached only through member variables of the abstraction, these member variables can be potentially partitioned into disjoint groups that can be synchronized independently. For example, a hash map may contain many buckets that are mapped to distinct hash keys, and operations that modify or read distinct buckets are fully independent of each other, so they can be independently synchronized if each function accesses at most a single hash key. On the other hand, in Figure 1, although the *head* and *tail* variables are often independently accessed, they must be cohesively updated when the list has ≤ 1 items, so they cannot be partitioned into independently synchronized groups.

To partition member variables of an abstraction, including those that have array types, into independently synchronized groups, our compiler analyzes each array variable to determine whether all the data referenced by each interface function can be reached from at most a single array entry, so that distinct entries can be independently synchronized. The rest of the member variables of the abstraction are then partitioned similarly, so that the data referenced by each interface function can be reached by at most a single group of variables.

c) **Selection of synchronization schemes:** Among the four synchronization schemes we support, the single-RCU+RLU scheme is the most general as it can always be used to correctly synchronize an arbitrary abstraction. It is therefore used as the default scheme selected by our compiler at the beginning before any further analysis is performed. It is then simplified into the single-RCU scheme if the compiler discovers that all data references of the abstraction are classified to be RCU-synchronized. On the other hand, it is changed into the RLU-only scheme if the compiler discovers all references are classified to be RLU-synchronized. Finally, as long as RLU-only is not selected, the compiler proceed to try partition the internal data of the abstraction into distinct groups, so that different groups can be safely synchronized independently via the multi-RCU+RLU.

d) **Classifying and synchronizing each function:** After analyzing the internal data references of an abstraction and then selecting an overall synchronization scheme, the selected scheme is implemented by modifying the source code of each interface function f. To this end, our compiler first collects all the memory references modified and read by f. If f does not access shared data, no synchronization is needed. Otherwise, a single RCU struct and its associated atomic pointer are identified, by searching through all the data referenced by f and matching them to a single synchronization group. Additional details are then synthesized by classifying f into four types:

- READ_RCU: if f doesn't modify anything and reads only RCU-synchronized data, it is synchronized by following the *is_empty* method in Figure 3;
- READ_RLU: if f reads both RCU and RLU synchronized data, without modifying anything, it is synchronized by following the *lookup* function in Figure 3;
- MOD_RLU: if f modifies RLU-synchronized data, and RLU-only is selected as the overall scheme, f is synchronized via RLU logs only, by following Figure 5.
- MOD_RCU: if f modifies shared data, and the overall scheme is not RLU-only, it is synchronized by following the *pushback* method in Figure 3.

e) **ABA prevention, lock-free Garbage Collection, and grace periods:** Our compiler uses weak CAS updates to physically apply all RLU logs. These updates are guaranteed to be correct only if each log uses a new unique value to replace an old unique one. To address the ABA problem [26], where multiple CAS updates set a value first to A, then to B, and then back to A, while violating real-time ordering of the updates, when logging RLU synchronized modifications, our compiler creates a new uniquely addressed wrapper object to hold each new value. These wrapper objects are eventually garbage collected together with their containing RLU logs.

To avoid allocating too many small objects (which is prohibitive at runtime), for each abstraction, our compiler-generated code pre-allocates a large memory pool shared by all threads to hold all the RCU objects and RLU logs, and two thread-local pools inside each RCU object to hold its affiliated ABA wrappers and user-freed data. Each pointer-free instruction in the original code is replaced with a special instruction that saves the freed pointer in the thread-local pool of the RCU-object, which will be garbage collected together with the RCU object.

Our compiler automatically synthesizes our custom lock-free garbage collector, which is triggered only when memory is exhausted, to manage these memory pools and reclaim unreachable memory. To safely reclaim RCU objects, a shared *activity* array is maintained to map each thread *id* to the address of the shared RCU object it currently uses, so that if any thread runs out of memory, it can simply reclaim a RCU object whose address is no longer present in the *activity* array. The RLU logs that are contained inside the reclaimed RCU object can be reclaimed if no older RCU object is still active, i.e., no active function can conflict with addresses modified by these logs; otherwise, these RLU logs are saved together with their RCU version number until all the older RCU objects have become obsolete and inactive.

As contention is a major source of performance degradation when a large number of threads concurrently operate, our compiler optionally make each thread *sleep* a designated amount of time before retrying upon detecting a conflict. This grace period is triggered inside the *check-conflict* function, invoked at lines 25 of Figure 3, after a conflict is detected but before an exception is thrown. Currently our compiler simply uses a constant, made reconfigurable at compile-time in the generated code via a cpp macro, to configure how long each thread will sleep before throwing an exception. In our experimental evaluation, we empirically tried different values for this macro and then selected a duration for the grace-period that provides the best overall scalability for each workload. This approach is likely to be inferior to a more sophisticated strategy that makes each thread dynamically adjust. However, it is extremely easy for a compiler-driven approach to implement and works reasonably well in our experimental evaluation. Exploring more sophisticated grace-period settings belongs to our future work.

IV. Compiler Automation

The compiler solutions in this paper are developed by adapting standard techniques (e.g., pointer aliasing, which determines whether multiple pointer variables may refer to the same object) to support automatic synchronization of concurrent data abstractions. The main technical novelty lies in formulating the necessary solutions to solve the new problems at hand, specifically to synchronize an arbitrary data abstraction x. The overall solution includes:

- *Structural analysis and normalization*, which identifies key components of x, specifically member variables and public methods that do not contain unsafe operations, to synchronize. Then, it normalizes x to be processed by later stages, e.g., by moving unsafe methods to private sections of the C++ class, inlining base classes, and inlining function calls inside the public methods, so that these public methods serve as a closed set of concurrent operations on x.

- *Pointer and data flow analysis*, including object connectivity analysis, function side-effect analysis, and reaching definition analysis, to determine aliasing relations among internal data of x, side effects of each method, and data-flow relations among memory references inside all member methods of x.
- *Data relocation*, which re-organizes the data of x after using the object-connectivity analysis to classify each memory reference inside x to be synchronized either by RCU or RLU. RCU-synchronized data are then partitioned into independent groups, with a new struct type and a new atomic pointer variable, illustrated in Figure 2, defined to relocate each group of data. All references to these relocated data are then redirected through their new atomic pointers.
- *Synchronization*, which uses the results of side effect and reaching definition analysis to classify and augment each interface function f of x for synchronization. First, f is restructured after selecting the synchronization template (READ_RCU, READ_RLU, MOD_RCU, or MOD_RLU) best suited for its purpose, based on results of its function side-effect analysis. Then using the result of data relocation analysis, memory references inside f are modified, e.g., to use the proper RCU copy or to create the proper RLU logs. Next, reaching definition analysis is used to relocate uses of these locally modified data, before inserting final augmentations, e.g., for ABA prevention and garbage collection.

The correctness of the above steps is guaranteed by the conservativeness of the analysis algorithms they use — if these algorithms cannot sufficiently understand some piece of code, the most conservative assumption is made to guarantee correctness. The scope of each analysis is flow-sensitive and intra-procedural (global but within a single function) [27]. In particular, each algorithm tries to analyze each individual interface function of x in isolation while assuming arbitrary values for non-local variables (the most conservative assumption). It then simply combines the results from analyzing all functions to represent all possible situations for x. This approach ensures the cost of all analysis algorithms is manageable (not overly steep) and is found to be sufficient for our purpose. The following text outlines the main analysis algorithms.

a) **Object Connectivity Analysis:** Detailed in Algorithm 1, this analysis aims to model the objects created and connected by each interface function of x. The algorithm follows the standard structure of data-flow analysis by first constructing a control flow graph (cfg) for the input at line 2. Each cfg node is initially associated with an empty set except the entry node, which is assigned with a graph given as a parameter to the algorithm. The connectivity graphs associated with each cfg node b are then iteratively modified, by collecting the results of using statements in b to modify connectivity graphs of all predecessors of b, through the *points_to_modification*

Algorithm 1: Object connectivity analysis

```
1  Function analyze_connectivity($g_0$ : initial graph, input: a single
     interface function)
2    cfg = build_control_flow_graph(input);
3    foreach basic block $b \in cfg$ do pt[b] = $\emptyset$ ;
4    pt[entry_node(cfg)] = $g_0$; change=true;
5    while change == true do
6      change = false;
7      foreach basic block b in cfg do
8        $g_1$ = pt[b]; pt[b] = $\emptyset$ /* recompute pt[b]*/;
9        foreach predecessor p of b in cfg do
10         pt[b] = pt[b] $\cup$ {points_to_modification(b, pt[t])};
11       if pt[b] $\neq g_1$ then change = true;
12   res = $\emptyset$;                // collect graphs at the exit of function
13   foreach basic block $b \in cfg$ that has no successor do
14     res = res $\cup$ pt[b];
15   return res;
```

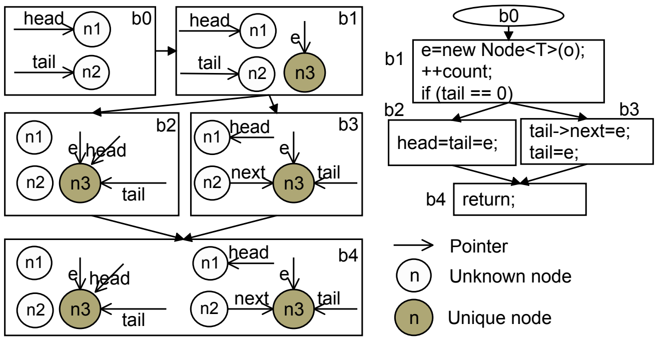


Fig. 6: Example connectivity graphs

function at line 10. The algorithm terminates when the results for all cfg nodes no longer change. Due to space constraints, we omit details of the *points_to_modification* function, which essentially modifies a given set of connectivity graphs based on the semantics of each statement that modifies any pointer variables. The points-to analysis is field sensitive (traces the addresses of different member variables of an object) but not array index sensitive (does not distinguish different subscripts of array references).

Figure 6 shows the resulting connectivity graphs from using this algorithm to analyze the *pushback* function at lines 3-6 of Figure 1(a). Here the entry node b_0 is associated with the initial connectivity graph, which includes two unknown objects, $n1$ and $n2$, pointed to by the *head* and *tail* member variables respectively. The compiler knows nothing about $n1$ and $n2$, so they can be both null or aliased to each other. Node $b1$ has $b0$ as a single predecessor and contains a single pointer-related operation, e=new Node<T>(o). Its connectivity graph therefore extends g_0 with a new unique node $n3$, pointed to by e. Node $b2$ and $b3$ both have $b1$ as predecessor but modify the pointers differently. They therefore each have their own connectivity graphs. These graphs are then collected together at node $b4$, which has both $b2$ and $b3$ as predecessors. The connectivity graphs of $b4$ would then be returned by the algorithm. In the end, the connectivity graphs from all interface functions represent all the possible ways different objects can be connected.

Since the algorithm terminates when the set of con-

Algorithm 2: Relocation Analysis

```
Function data_analysis(c: input, ocg: object connect graph)
    cp_vars=unaliased_member_variables(c, ocg);
    partition = { cp_vars }; foreach interface function f of c do
        ref_vars = trace_to_variables(side_effect_analysis(f));
        p1 = ref_vars ∩ cp_vars; if p1 == cp_vars then
            partition = { cp_vars }; break;
        p2 = p1; foreach p3 ∈ partition do
            if p1 ⊆ p3 then
                break;
            if p1 ∩ p3 == ∅ then
                continue;
            p2 = p1 ∪ p3; partition = partition - { p3 };
        if p2 == cp_vars then
            partition = { cp_vars }; break;
        if p2 ⊃ p1 (p2 subsumes p1 and other members) then
            partition = partition ∪ { p2 };
    log_vars=∅;
    foreach unknown node x in ocg do
        log_vars = log_vars ∪ memory_references_of(x);
    foreach unique node x reachable from cp_vars in ocg do
        if x is never modified in c then continue;
        else
            p = ∅;
            foreach g ∈ ocg do
                p = p ∪ summarize_paths(g,cp_vars,x);
            if p has only one entry from member variable e then
                cp_vars = cp_vars ∪ {references_of(x) → e};
            else log_vars = log_vars ∪ {references_of(x)};
```

nectivity graphs assigned to each cfg node no longer changes, the termination of the algorithm is guaranteed if the overall number of different connectivity graphs is bounded by a constant, as each iteration of the algorithm can only add new connectivity graphs to the result already computed by previous iterations. To guarantee termination, our algorithm allocates at most one object for each memory reference. The number of nodes in each connectivity graph is thus bounded by the number of memory references analyzed (R), and the number of edges by R^2. Since all graphs have the same nodes, the number of different graphs is bounded by a constant.

b) Side Effect and Reaching Definition Analysis: which respectively identifies the memory references read and modified by each interface function of x and discovers the set of data modifications that can reach each memory reference. Both use standard program analysis algorithms available in most compiler books [27]. Our main extension is that when considering modifications to indirectly referenced objects, the object connectivity graphs are used to help resolve pointer aliasing issues. In particular, each memory reference is mapped to a node in the connectivity graphs, and if the node is tagged as *unknown*, it may be aliased with all the other *unknown* nodes. No distinction is made between array variables vs pointer variables — each of them is by default assumed to refer to an arbitrary memory region. All array references are simply treated as arbitrary indices into the array memory regions,

c) Data Relocation Analysis: Algorithm 2 shows our algorithm for classifying and partitioning the internal data of an abstraction x. It first finds all member variables (including array/pointer variables) whose addresses are never taken to be later relocated to a RCU-synchronized struct type (cp_vars at line 2). Then, the set of member variables accessed by each interface function f is collected (lines 4-5) and used to generate a partitioning of cp_vars, with variables in different partitions never accessed together inside any function (lines 3-16). Our algorithm supports partitioning of arrays by allowing each distinct entry of an array to be an independently synchronized group, if it can be verified that each interface function of the abstraction uses array subscripting to access only a single entry of the array/pointer variable. Finally, for each memory reference mapped to a unique node in the connectivity graphs, the algorithm examines the number of paths reaching the object from the non-aliased member variables and categorizes the object as non-aliased (RCU data) only if it is reachable through only a single path from some variable in cp_vars (lines 20-28). For example, in Figure 6, the $n3$ unique node can be reached either through the head or the tail member variables of the list, so $n3$ cannot be relocated. All memory references that do not belong to cp_vars are simply classified to be synchronized by RLU and saved in log_vars (lines 17-19 and 28).

V. Experimental Evaluation

We have implemented a prototype compiler using the POET language [28] and have used our compiler to automatically convert eight sequential C++ classes, shown in the second column of Table I. These C++ classes are collected from two existing C/C++ open-source projects: the ROSE compiler [29] and the Tervel framework [30]. By default, our compiler has used single-RCU+RLU to synchronize the first five abstractions in Table I, multi-RCU+RLU to synchronize the hash-set abstraction, and RLU-only to synchronize the binary tree (BST) and Multi-dimensional list (MDlist). To additionally compare the multi-RCU+RLU and RLU-only synchronization schemes, we have slightly modified the top-level data organization of the original sequential implementations of BST and MDlist, by using an array to store their top-level nodes to enhance concurrency among operations that access different portions of the linked data structures. The array-based and list-based implementations are identical except differences in a few lines, including the top-level declaration of the abstraction member variable, and the beginning of each operation that accesses the top-level variable. To reduce contention, each auto-generated version uses a Cpp macro to set how long each thread should sleep before throwing an exception to restart its operation.

Our compiler is general purpose in that all of its algorithms take an arbitrary C/C++ class as input, with no inherit assumption about specifics of the underlying data structure. Our prototype implementation is limited only by idiosyncrasies of the C++ language and completeness of the internal analysis algorithms in handling all such idiosyncrasies. Out of the eight C++ classes in Table I, our

Workload	Auto-generate	RSTM [31]	TBB [32]	FC[33]& SIM[34]	Boost[35] Back-off	CDS [36]

Lightweight write-only	Ringbuffer
	Stack(array-based)
	Deque(list-based)
	Singly/Doubly linked list
			Queue	Queue
				Stack
					Michael-Scott queue[6]
					Treiber stack[37]	

Heavyweight write-only& mostly-read	HashSet(array of lists) Binary tree(unbalanced) Multi-dimensional list[38]		Hash-map	-		Michael's hashset[9] Herlihy's skiplist[2] Ellen's BST[12]

TABLE I: Benchmark and workload configurations

compiler was able to successfully synchronize 40-100% of their original interface functions, while pruning the others.

Our results indicate that the performance of our auto-synchronized code is competitive to that of manually written implementations by experts. One potential threat to the validity of our conclusion is that there might be better manual data structure implementations that we did not compare with. Similarly a new hardware architecture might favor a different set of synchronization techniques than those we support. An expert can certainly fine tune an existing implementation to far exceed the performance of our auto-generated code. However, our purpose of investigating a compiler-driven approach is exactly to automate such manual state-of-practice so that an expert does not need to repeatedly apply his/her expertise to implement different data structures. The generality of our approach makes advanced lock-free synchronization techniques much more readily available to average users. This paper is only a first step towards this direction.

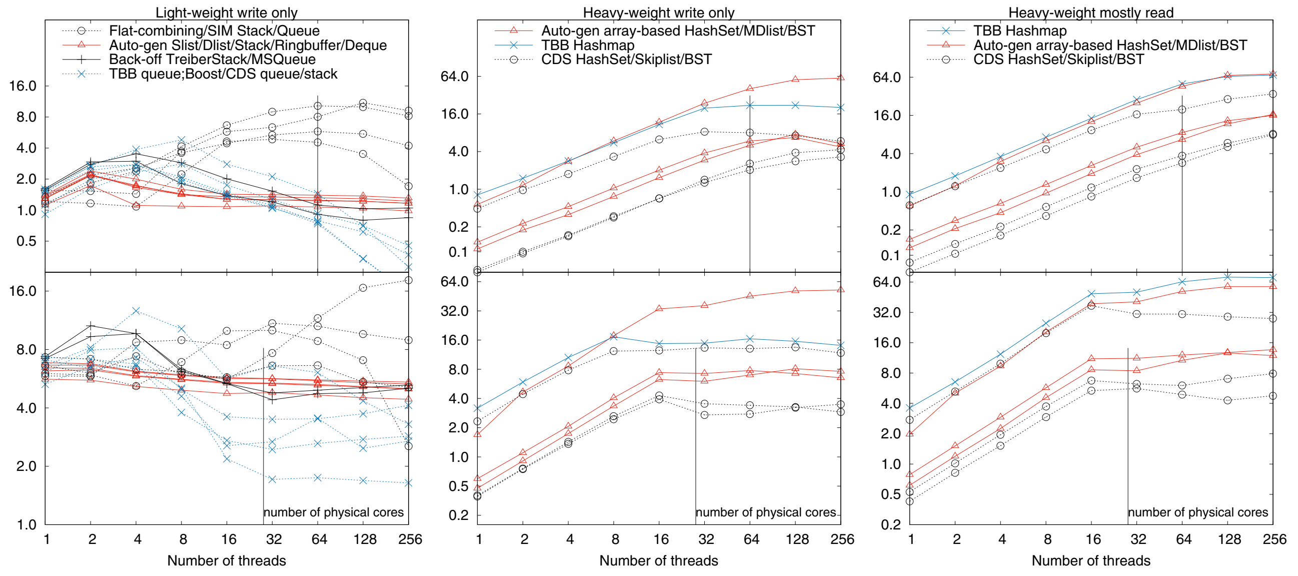
The last four columns of Table I show a set of manually synchronized concurrent data structures we can find for comparison, each crafted from scratch by expert researchers in the field. To compare with STM, for each of the eight sequential data abstractions in Table I, we also manually synchronized them via RSTM [31], one of the fastest obstruction-free STM implementations. Experimentation is used to find the best configuration parameters for each implementation (including the length of hash keys in all the hast-set implementations, and back-off/sleep time in michael-scott queue, treiber stack, and our auto-generated code) when using different numbers of threads. We manually implemented a set of micro-benchmarks to test all data structures in Table I using three workloads: *light-weight write-only*, which tests abstractions in the first row, by treating each as a concurrent queue/stack and concurrently invoking two operations, a *push* and a *pop*, each with 50% probability; *heavy-weight write-only*, which tests the heavier weight abstractions in the second row, by treating each as a set/map and concurrently modifying it via two operations, an insert and an erase, each with 50% probability; and *heavy-weight mostly-read*, which alternatively tests the heavy-weight abstractions by reducing modifications to 10% while adding 90% probability of a read-only operation that searches through the data. To make sure each data structure always contains enough elements for meaningful operations, and the test runs a sufficiently long time for timing stability, our benchmark first initializes each data structure with a large number (set to be $2.56 * 10^6$) of elements and then spawns a pre-configured number of threads to collectively invoke a pre-determined number (again set to $2.56 * 10^6$) of the pre-selected operations. This configuration potentially offers more concurrency among operations in the light-weight write-only workload and therefore magnifies the importance of exploring such concurrency, which is currently not explored (a weakness) in our synchronization schemes. All workloads are measured by their throughput, computed by dividing the number of operations completed with the average time taken to complete the operations.

We evaluated all the tests on two platforms: a 64-core Intel Xeon Phi 7210 processor, with 4 hyper threads on each core; and an SMP node with two Intel Xeon E5-2695 v3 processors, each with 14 cores, from one of the Xsede servers [39]. Both machines run CentOS Linux as the underlying operating system. All data structure implementations are compiled using g++ 4.8.5 with the *-O3* flag and c++11 enabled. Each measurement is repeated 10 times, and the averages reported. Variations across runs of the same measurement are under 10%.

A. Performance of RCU Synchronized Implementations

Figure 7 compares the best performance of our auto-synchronized data structures, specifically those via single- and multi-RCU+RLU schemes, with that of manually synchronized implementations. To improve readability, we grouped the implementations so that related ones are displayed using the same line style. The members in each group are listed in decreasing order of their performance. The different groups of data structures all also listed in decreasing order of their collective performance.

Overall, the performance of our auto-generated RCU-synchronized implementations are among the best across all workloads, ranking first in the heavy-weight (hash-set/MDlist/BST) write-only workload and second in the other two workloads, based on the best throughput that was eventually attained when using 1-256 threads for the workloads. The main observation is that the multi-RCU synchronization (used by our compiler in the heavy-weight workloads) produces superb scalability by allowing a high-degree of concurrency, as no interaction is required among fully independent operations. In the case of the hashset, each RCU object synchronizes all updates to the same hash key together. The auto-synchronized MDlist and BST lag behind the hashset, because only a constant number of RCU objects are used at the top level, in contrast to the much larger array of atomic pointers used in the hashset. The Single-RCU strategy is used by our compiler to synchronize the five light-weight data structures. Here because it sequentializes all the update operations, it lags behind the manual implementations when using a small number of threads. However, it scales better when the number of threads exceeds 32, mostly because the tuning of sleep-time reduces contention when there is no concurrency among the operations. Flat combining performed



* Top graphs: on the Intel Xeon Phi server; Bottom graphs: On the Xsede server

Fig. 7: Throughput (million ops/second) of RCU-synchronized and manually synchronized implementations

the best in the light-weight workload because it combines multiple updates to reduce overhead.

B. Comparing With STM

Figure 8 compares all variations of our auto-generated code, including those synchronized via single-RCU+RLU with and without tuning of sleep-time-before-retry, those synchronized via multi-RCU+RLU (hashset, MDlist, and BST), and those synchronized via RLU-only (MDlist and BST), with the corresponding implementations synchronized via STM. Here significant performance improvement is observed by the tuning of sleep-time for the light-weight workload and by using the array-based sequential implementation for MDlist and BST. Note that even without any manual intervention, our compiler synchronized implementations performed much better than the RSTM implementations, which performed competitively only under the heavy-weight mostly-read workload. For the write-only workloads, the STM versions performed badly when the number of threads exceeds 16 and conflicts among operations become high. In the heavy-weight write-only workload, the RSTM implementations reached their max throughput when the number of threads is only 4, due to heavier modifications to the shared addresses. For the lightweight workload, the RSTM versions performed reasonably only when the number of threads are ≤ 4 and demonstrated a worst level performance degradation among all implementations afterwards.

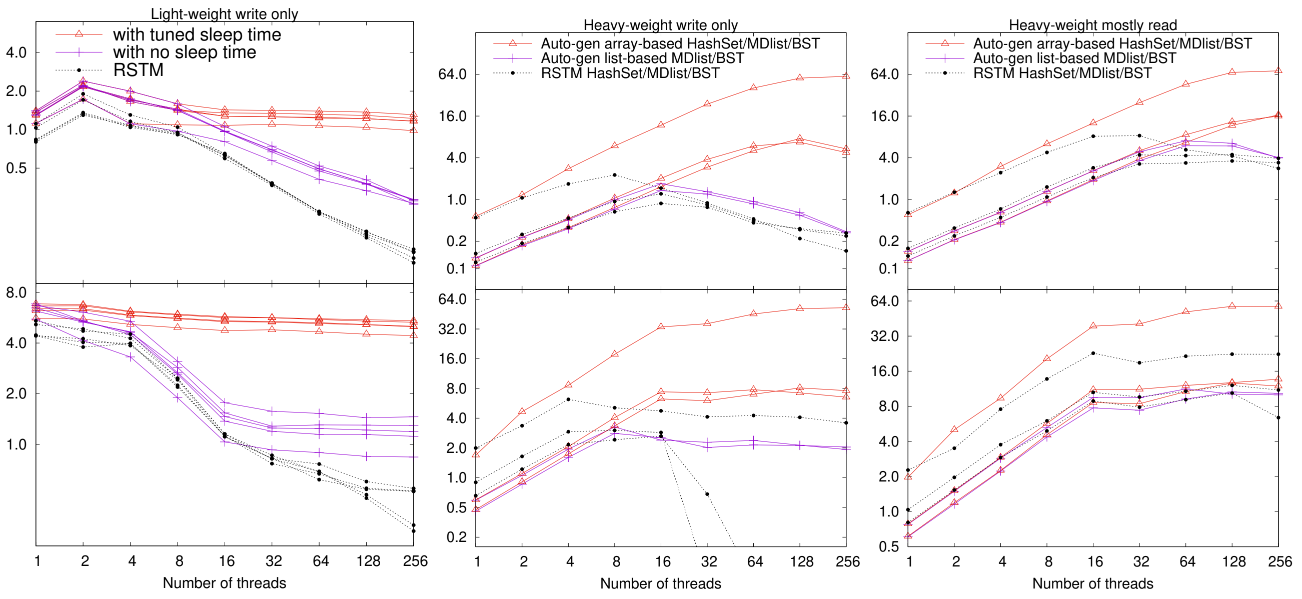
C. Estimating Synchronization Overhead

Figure 9 shows the run-time overhead introduced by our auto-inserted synchronizations for the various concurrent data structures, estimated using equation $(T_p * P - T_s)/T_s$, where T_p is the elapsed time of using P threads to concurrently complete a workload, and T_s is the time required when using their original sequential implementations. Here because we plot the cumulative overhead of all the threads, the overhead can scale linearly as the number of threads increases, as we observe for the light-weight write-only workload, where the single-RCU+RLU scheme virtually sequentialized their concurrent modifications. However, for the heavy-weight write-only and mostly-read workloads, the overhead stayed relatively low and constant irrespective of the increasing number of threads, until there are more threads than the number of CPU cores.

VI. Related Work

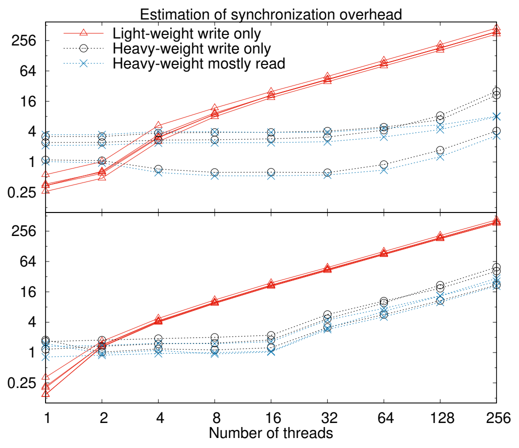
In contrast to existing work on the manual design of non-blocking concurrent data structures, e.g., queues [6], [7], lists [8], [9], maps [9], [10], and trees [11], [12], [13], [14], this paper represents the first attempt at using compiler technology to automate the process. Note that existing compilers that support transactional memory programming merely translate the higher-level STM programming interface down to lower-level library invocations, without involving any compile-time analysis to tailor synchronizations to the characteristics of different pieces of code. Michael and Scott [5] studied the performance of non-blocking algorithms vs locking and observed that efficient data-structure-specific non-blocking algorithms outperform the other alternatives. This paper automatically tailors general techniques to the needs of individual data structures and has demonstrated similar advantages.

The synchronization adopted by our compiler is a combination [23] of read-copy-update [21] and read-log-update [22]. Our retrieval of older values through saved RLU-logs for read-only functions is similar in ideas to



* Top graphs: on the Intel Xeon Phi server; Bottom graphs: On the Xsede server

Fig. 8: Comparing Throughput (million ops/second) with STM-synchronized abstractions



* Top: Intel Xeon Phi; Bottom: Xsede

Fig. 9: Synchronization overhead of auto-generated code

the multi-versioning extension of RLU [40]. Single-word compare and swap (CAS) has been widely used as a primitive for implementing lock-free or wait-free synchronizations [41], [23]. Our compiler-driven approach can be potentially used to automate other advanced synchronization mechanisms as well, e.g., fine-grained locking [14], flat combining [33], among others [42], [43], [44], [45], [46].

Herlihy and Moss [47] first proposed transactional memory as a hardware architecture, the materialization of which includes Intel TSX [48] and AMD ASF [49]. Our auto-generated code can be converted to using hardware-level transactional operations if needed. However, since we focus on software-level synchronization, our experimental study uses conventional hardware architectures.

Software transactional memory was first proposed by Shavit and Touitou [50] and was later extended to support dynamically sized data structures [16], conditional critical regions [51], transactional monitors [52], composition of blocking transactions [17], among others [53]. Modern STM systems have been implemented both by using lock-based [54], [55] and non-blocking synchronizations [51], [16], [18], [11], Steep runtime cost is required when implementing STM [11], [18], [56]. Many optimizations, e.g., obstruction-free synchronization [16], transaction logging [17], fine-grained object disambiguation [51], have been developed to reduce such overhead. Our work similarly follows this direction.

Our work is complementary to automated refactoring of existing code for concurrency via libraries [57] and automated fixing of concurrency bugs [58], [59], [60]. The sketch synthesis algorithm [61] tries to iteratively complete the sketch of a concurrent data structure from developers until reaching a given criteria. Vechev et al [62], [63] automatically inferred synchronizations to avoid interleavings that violate user specifications. Our compiler requires only the sequential implementation of C++ classes and aims to automate non-blocking synchronization.

VII. Conclusion

This paper presents compiler techniques to automatically convert sequential data abstractions into concurrent lock-free ones, by adapting existing state-of-the-practice synchronization mechanisms to maximize concurrency. We present experimental results to show that our auto-generated implementations can attain performance that is competitive to manually crafted ones by experts.

Acknowledgement

This research is funded by NSF through grants CCF-1261584, CCF-1421443, CCF-1717515, and OAC-1740095.

References

[1] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, p. 569, Sep. 1965.

[2] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[3] A. Karlin, K. Li, M. Manasse, and S. Owicki, "Empirical studies of competitve spinning for a shared-memory multiprocessor," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 41–55, Oct. 1991.

[4] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, pp. 21–65, Feb. 1991.

[5] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 51, pp. 1–26, May 1998.

[6] ——, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96. New York, NY, USA: ACM, 1996, pp. 267–275.

[7] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ser. ICDCS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 522–.

[8] V. L. M. M. W. N. S. I. Steve Heller, Maurice Herlihy and N. Shavit, "A lazy concurrent list-based set algorithm," in *Proceedings of the 9th International Conference On Principles Of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 3–16.

[9] M. M. Michael, "Safe memory reclamation for dynamic lock-free objects using atomic reads and writes," in *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, ser. PODC '02. New York, NY, USA: ACM, 2002, pp. 21–30.

[10] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "Transactional predication: High-performance concurrent sets and maps for stm," in *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '10. New York, NY, USA: ACM, 2010, pp. 6–15.

[11] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, May 2007.

[12] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '10. New York, NY, USA: ACM, 2010, pp. 131–140.

[13] T. Crain, V. Gramoli, and M. Raynal, "A contention-friendly binary search tree," in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. an Mey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 229–240.

[14] M. Arbel and H. Attiya, "Concurrent updates with rcu: Search tree as an example," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14. New York, NY, USA: ACM, 2014, pp. 196–205.

[15] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, pp. 99–116, 1997.

[16] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, ser. PODC '03. New York, NY, USA: ACM, 2003, pp. 92–101.

[17] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 48–60.

[18] V. J. Marathe, W. N. Scherer, and M. L. Scott, "Adaptive software transactional memory," in *Proceedings of the 19th International Conference on Distributed Computing*, ser. DISC'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 354–368.

[19] Q. Yi, "The POET language manual," 2008, www.cs.utsa.edu/∼qingyi/POET/poet-manual.pdf.

[20] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects." ACM, 1990, pp. 463–492.

[21] P. E. McKenney and J. D. Slingwine, "Read-copy-update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, 1998, p. 509âĂŞ518.

[22] A. Matveev, N. Shavit, P. Felber, and P. Marlier, "Read-log-update: A lightweight synchronization mechanism for concurrent programming," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 168–183.

[23] M. Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Trans. Program. Lang. Syst.*, vol. 15, pp. 745–770, Nov. 1993.

[24] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr., "Lock-free reference counting," *Distrib. Comput.*, vol. 15, no. 4, Dec. 2002.

[25] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas, "Efficient and reliable lock-free memory reclamation based on reference counting," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 8, Aug. 2009.

[26] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the aba problem in descriptor-based lock-free designs," in *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ser. ISORC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 185–192.

[27] K. Cooper and L. Torczon, *Engineering a Compiler*. Morgan Kaufmann, 2004.

[28] Q. Yi, "POET: A scripting language for applying parameterized source-to-source program transformations," *Software: Practice & Experience*, pp. 675–706, May 2012.

[29] D. Quinlan, M. Schordan, R. Vuduc, and Q. Yi, "Annotating user-defined abstractions for optimization," in *POHLL06: Workshop on Performance Optimization for High-Level Languages and Libraries*, Rhode Island, Greece, 2006.

[30] D. Dechev, P. LaBorde, and S. Feldman, "Lc/dc: Lockless containers and data concurrency: A novel nonblocking container library for multicore applications," 2013.

[31] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of nonblocking software transactional memory," in *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, 2006.

[32] *Intel Threading Building Blocks Reference Manual*, Intel Corporation, 2014. [Online]. Available: http://www.threadingbuildingblocks.org/docs/help/reference

[33] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 355–364.

[34] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 325–334.

[35] *Boost 1.56.0 Library Documentation*, 2014. [Online]. Available: http://www.boost.org/doc

[36] M. K. aka khizmax, *Concurrent Data Structures library*, 2012. [Online]. Available: http://libcds.sourceforge.net/doc

[37] R. K. Treiber, *Systems Programming: Coping with Parallelism*, ser. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[38] D. Zhang and D. Dechev, "An efficient lock-free logarithmic search data structure based on multi-dimensional list," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, June 2016, pp. 281–292.

[39] *Bridges User Guide*. [Online]. Available: https://portal.xsede.org/psc-bridges

[40] J. Kim, A. Mathew, S. Kashyap, M. K. Ramanathan, and C. Min, "Mv-rlu: Scaling read-log-update with multi-versioning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 779–792. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304040

[41] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 124–149, Jan. 1991.

[42] J. Turek, D. Shasha, and S. Prakash, "Locking without blocking: Making lock based concurrent data structure algorithms nonblocking," in *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '92. New York, NY, USA: ACM, 1992, pp. 212–222.

[43] Y. Afek, D. Dauber, and D. Touitou, "Wait-free made fast," in *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '95. New York, NY, USA: ACM, 1995, pp. 538–547.

[44] A. Turon, "Reagents: Expressing and composing fine-grained concurrency," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 157–168.

[45] M. Arbel and A. Morrison, "Predicate rcu: An rcu for scalable concurrent updates," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 21–30.

[46] J. F. Martínez and J. Torrellas, "Speculative synchronization: Applying thread-level speculation to explicitly parallel applications," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 18–29.

[47] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300.

[48] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, Mar 2014.

[49] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière, "Evaluation of amd's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 27–40.

[50] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. New York, NY, USA: ACM, 1995, pp. 204–213.

[51] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 388–402.

[52] A. Welc, S. Jagannathan, and A. L. Hosking, "Transactional monitors for concurrent objects," in *In Proceedings of the European Conference on Object-Oriented Programming*. SpringerVerlag, pp. 519–542.

[53] M. Moir, "Transparent support for wait-free transactions," in *Proceedings of the 11th International Workshop on Distributed Algorithms*, ser. WDAG '97. London, UK, UK: Springer-Verlag, 1997, pp. 305–319.

[54] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: A high performance software transactional memory system for a multi-core runtime," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '06. New York, NY, USA: ACM, 2006, pp. 187–197.

[55] W. N. Scherer, III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '05. New York, NY, USA: ACM, 2005, pp. 240–248.

[56] D. Dice and N. Shavit, "Understanding tradeoffs in software transactional memory," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 21–33.

[57] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 397–407. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070539

[58] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 221–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387902

[59] P. Černý, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, "Efficient synthesis for concurrency by semantics-preserving transformations," in *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, ser. CAV 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pp. 951–967.

[60] H. Lin, Z. Wang, S. Liu, J. Sun, D. Zhang, and G. Wei, "Pfix: Fixing concurrency bugs based on memory access patterns," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 589–600. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238198

[61] A. Solar-Lezama, C. G. Jones, and R. Bodik, "Sketching concurrent data structures," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 136–148.

[62] M. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, NY, USA: ACM, 2010, pp. 327–338.

[63] M. Vechev and E. Yahav, "Deriving linearizable fine-grained concurrent objects," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 125–135.