Received: 23 August 2021 | Revised: 2 January 2023 | Accepted: 29 September 2023

DOI: 10.1002/cpe.7935

RESEARCH ARTICLE

WILEY

Compiler-driven approach for automating nonblocking synchronization in concurrent data abstractions

Jiange Zhang[1] | Qing Yi[1] | Christina Peterson[2] | Damian Dechev[2]

[1]University of Colorado at Colorado Springs, Colorado Springs, Colorado, USA

[2]University of Central Florida, Orlando, Florida, USA

Correspondence
Christina Peterson, University of Central Florida, Orlando, FL, USA.
Email: clp8199@knights.ucf.edu

Funding information
National Science Foundation, Grant/Award Numbers: CCF-1261584, CCF-1421443, CCF-1717515, OAC-1740095

Summary

This paper presents an extended version of our previous work on using compiler technology to automatically convert sequential C++ data abstractions, for example, queues, stacks, maps, and trees, to concurrent lock-free implementations. A key difference between our work and existing research in software transactional memory (STM) is that our compiler-based approach automatically selects the best state-of-the-practice nonblocking synchronization method for the underlying sequential implementation of the data structure. The extended material includes a broader collection of the state-of-the-practice lock-free synchronization techniques, additional formal correctness proofs of the overall integration of the different synchronizations in our system, and a more comprehensive experimental study of the integrated techniques. We evaluate our compiler-generated nonblocking data structures both by using a collection of micro-benchmarks, including the Synchrobench suite, and by using a multi-threaded application Dedup from PARSEC. Our automatically synchronized code attains performance competitive to that of concurrent data structures manually-written by experts and much better performance than heavier-weight support by STM.

KEYWORDS
compiler technology, lock-free synchronization, read-copy-update

1 | 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,[1-5] nonblocking 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 is an extended version of our previous work[6] on using source-to-source compiler technology to automatically convert sequential C++ data abstractions to lock-free concurrent implementations, to relieve developers from the error-prone task of manually crafting and testing the cumbersome low-level synchronization details, for example, those illustrated in Figures 4–6. Such carefully crafted lock-free synchronous implementation details have been published for a variety of data structures, for example, queues,[7,8] sets,[9-11] lists,[9,10] maps,[9,11] and trees,[12-15] by following a large collection of systematic construction methods.[7-15] This paper represents the first attempt at using compiler technology to automate this process.

Compared to our previous work, this paper includes a broader collection of the state-of-the-practice lock-free synchronization techniques. Additionally, formal proofs of correctness are included to demonstrate that the compiler-generated nonblocking data structures provide a high-level of correctness guarantees for their implementations. A more comprehensive experimental evaluation of the integrated synchronization techniques

is also provided, including an evaluation of the compiler-generated data structures on the micro-benchmark suite Synchrobench[16] and on the application Dedup from PARSEC.[17]

As an alternative to our compiler-driven approach, lock-free synchronization can also be automatically supported via software transactional memory (STM),[18-20] which provides a higher-level programming interface for lock-free synchronization in whole software applications. Compared to STM, our compiler supports synchronization at the granularity of a single abstraction and therefore is not as general-purpose. However, a key insight from our work is that restricting the scope of synchronization to a single self-contained abstraction yields significant reductions in run-time overhead of synchronization. The reduced scope facilitates an automatic selection of the best synchronization strategy to tailor to the underlying sequential implementation, 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.

A 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*[18,19]). 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.[5,12,21] The performance attained by our auto-generated code can be limited by their original sequential implementations, for example, 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. The synchronization methods include variations of read-copy-update, read-log-update, and flat combining.
- 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 concurrent operations.
- We present formal proofs that our technique for automating nonblocking synchronization generates concurrent data structures whose operations are correct.
- 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 evaluate our compiler-generated nonblocking data structures both by using synthetic benchmarks, including the Synchrobench,[16] and by using a realistic application benchmark Dedup from PARSEC.[17] 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 STM.

Our prototype compiler, together with all the data structures evaluated in this paper, has been released as part of the GitHub source code release of the POET language,[22] an interpreted language for fast prototyping of source-to-source compiler analysis and transformations.[23,24]

2 | DETAILS OF SYNCHRONIZATION

Given n concurrent operations $op_1, op_2, \ldots, op_n$, they are *linearizable* if there exists a sequential ordering of the same operations that maintains real-time ordering and produces the same results.[25] The synchronization is nonblocking 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 three widely-used nonblocking synchronization techniques: read-copy-update[26] (RCU), read-log-update[27] (RLU), and flat-combining,[28,29] into five synchronization schemes: single-RCU, single-RCU+RLU, single-RCU+RLU with combining, 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. The lighter weight single-RCU/single-RCU+RLU schemes sequentialize all concurrent modifications. The single-RCU+RLU+combining scheme allows independent lightweight operations to be combined to reduce synchronization overhead. All schemes allow read-only operations to make full progress independent of ongoing modifications. The following uses the singly-linked list in Figure 1 as example to detail each of these schemes.

Single-RCU

Here all shared data of an abstraction are 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 *e*2 (3) *f* proceeds as in a sequential setting, except all shared data accesses are redirected through address *e* (or *e*2 if the data in *e* has been copied to *e*2); and (4) upon completion, if *f* has modified data in *e*2, it tries to use *e*2 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). As shown in Figure 2, if *f* does not modify shared data, it 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.

Figure 3 illustrates how this scheme is used to synchronize two threads concurrently modifying a singly-linked list, whose data are stored in a contiguous region, with the beginning address stored in an atomic pointer. A copy of this address is retrieved and saved into a private variable *oldp* by each of the concurrent threads. Each thread then proceeds to make a private copy of linked list pointed to by *oldp*. The private copies of the linked list are then concurrently modified, where thread *a* pushes a new node *n*4 at the end of its private list, and thread *b* pops off the first element in its private list. Both threads then try to commit their changes at the same time via a CAS operation over the atomic pointer, where thread *a* succeeds and moves on to the next operation, while thread *b* fails and retries its pop operation.

The main difference between our RCU implementation and that originally published in Reference 26 is that our implementation immediately restarts an operation on a CAS failure if other threads have modified the shared atomic pointer. In contrast, the original RCU implementation waits for a *quiescent period* (a period of thread inactivity) before retrying updates.

```
template <class T> SinglyLinkedList {
private:
 Node<T>* head; Node<T>* tail; unsigned count;
 Node<T>* First() const { return head; }
 Node<T>* Last() const { return end; }
 Node<T>* Next(Node<T>* current) const { return current->next; }

public:
 void pushback(const T& o){
   Node<T>* e=new Node<T> (o); ++count;
   if (tail==0) {head=tail=e;} else {tail->next=e; tail=e;} }

 bool is_empty(){ return count == 0; }

 bool lookup(const T& t) {
   Node<T> *e = head; while (e!=0 && e->content!=t) e = e->next;
   return (e!=0); }
 ...  };
```

FIGURE 1 Example: a sequential singly-linked list.

```
typedef struct List_state {
 Node<T>* head; Node<T>* tail; unsigned count; unsigned id;
 typedef struct ModLog {
    Node<T>** addr; /*address being modified*/
    Node<T> *old, *new; /*old and new values of addr*/} ModLog;
 atomic<vector<ModLog>*> modlogs;

 void copy(List_state *that) {id=that->id+1;...copy members...}

 void new_modlog(Node<T>** _addr,Node<T>* _old,Node<T>* _new)
  { check_conflict(this,_addr,_old);
    vector<ModLog> *c = load_modlog(modlogs);
    c->push_back(ModLog(_addr,_old,_new)); }

 void modlog_apply()
  { vector<ModLog> *c=finalize_modlog(modlogs); if (c==0) return;
    for (vector<ModLog>::iterator p=c->begin();p!=c->end();p++)
    { ModLog t = (*p);
      atomic<Node<T>*>* t1 =
         reinterpret_cast<atomic<Node<T>*>*>(t.addr);
      cas_mod(t1,&t.old,t.new,id); /*use weak CAS to modify t1*/
   } }
} List_state;

atomic<List_state*> state;
```

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

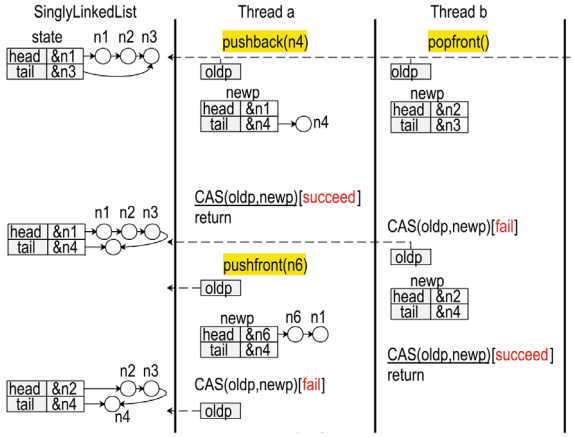


FIGURE 3 Example: single RCU synchronization.

```
1 void pushback(const T& o) {
2   Node<T>* e=new Node<T> (o);
3   List_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 { List_state *oldp=load_current_state(READ_RCU);
17   oldp->modlog_apply(); return oldp->count == 0; }
18
19 bool lookup(const T& t)
20 { List_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){} } };
```

FIGURE 4 Example: synchronizing a singly-linked list (text in red is related to RCU synchronization).

To illustrate our implementation, 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 pointer, *state*, is then declared at line 25 to hold the most current address of the relocated data, termed an *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 4 via the red-colored text. Black-colored text corresponds to single-RCU+RLU synchronization and program logic from the sequential data structure. The functions synchronized using single-RCU include both the red-colored text and the program logic from the sequential counterpart.

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. If the list is not empty, then the modification requires two updates: (1) the tail must be set to the new node, and (2) linkage between the previous tail and the new node must be added. Since this requires two steps, a nonempty list requires the single-RCU+RLU scheme. Garbage collection for *oldp* at line 11 of Figure 4 is handled by replacing the pointer-free instruction in the original code with an instruction that saves the freed pointer in the thread-local pool of the RCU-object for garbage collection and reclamation, discussed in further details in Section 4.

To apply single-RCU to other data structure types, all member variables are relocated to a struct representing the state of the data structure. An atomic pointer *state* is declared to maintain the RCU object, and an *id* field is declared in the RCU struct to uniquely identify the RCU object. The member operations are augmented with additional code to manage accesses using the RCU object. If the operation is read-only, the reads are redirected through the loaded RCU object. If the operation performs a modification, then it loads the RCU object, makes a copy of the shared data,

and performs all modifications to the local copy. The updates are publicized by applying a CAS to update the RCU object address to the address of the local copy.

Single-RCU+RLU

Here single-RCU is combined with the RLU (Read-Log-Update) method 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). A weak CAS permits a spurious failure, where the CAS may fail even if the contents of the memory location is equivalent to the expected value. Weak atomic updates are used because they yield better performance than strong atomic updates. If the weak CAS spuriously fails, a conflict is detected and the operation is restarted. By enforcing that all threads that observe a set of publicized logs immediately help to complete these logs via weak atomic updates (concurrent redundant updates are 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.

The difference between the proposed approach and the original RLU technique is that the proposed approach incorporates thread helping to ensure that all publicized logs are performed in sequence. The original RLU technique maintains a write-log per thread and a global clock that is used to decide whether to use the old version of the data or the logged version. Once the global clock is updated by a thread, this thread waits for all readers with a lower clock value to complete.

To combine RCU with RLU, our compiler augments each RCU struct with an array of modification logs, for example, at lines 3–5 of Figure 2. The *new_modlog* function calls the *check_conflict* function at line 11 of Figure 2 to dynamically detect conflicted logs prior to adding a new modification log to the *modlogs* vector. The *load_modlog* function obtains the most recent address of the *modlogs* atomic pointer on line 12 of Figure 2, then appends the new modification log to the back of the *modlogs* vector on line 13 of Figure 2. The *push_back* invoked on the pointer to *modlogs* on line 13 of Figure 2 is guaranteed to be thread-safe because a *newp* is allocated on line 3 of Figure 4 and *oldp* is copied to *newp* on line 6 of Figure 4. This ensures that *new_modlog* is performed locally since *newp* is not accessible to other threads until the CAS succeeds on line 11 of Figure 4. The *finalize_modlog* function invoked at line 16 of Figure 2 is implemented so that before returning the array of logged modification, it first makes sure that all the modification logs for a RCU object have been finalized and publicized.

The modification logs are created and saved inside a local copy of the RCU struct, by invoking its *new_modlog* method, for example, at line 8 of Figure 4. All the delayed logs are finally publicized, for example, at line 11 of Figure 4, when the local copy is used to replace the shared atomic pointer. Once successfully publicized, these logs are applied when 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 4). To ensure each operation observes the new publicized logs in time, a *check_conflict* function is invoked (e.g., at line 25 of Figure 4) 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 occurred is retrieved to allow the read-only operation to complete using its old data irrespective of the detected conflicts.

Figure 5A illustrates how three concurrent threads operating on the singly-linked list in Figure 4 interact with each other when synchronized through this scheme. The *check* function is the dynamic conflict detection function. *check* retrieves the RCU object and if it is the most update-to-date object held by the shared RCU atomic pointer, then no conflict is detected. Otherwise a conflict is detected and the operation is restarted. A conflict is detected for *n*3->*next* because *n*3 was updated by thread (a) after thread (c) had retrieved the current state stored in *oldp*. Dynamic conflict detection is described in detail at the end of this section. The three dependent modifications by threads (a) and (b) are sequentialized, while the read-only operation by thread (c) is allowed to complete using old data irrespective of the ongoing modifications.

Single-RCU+RLU with flat-combining

The single-RCU+RLU scheme is further extended to allow each concurrent thread to help complete modifications previously announced by other threads. This allows all the publicly announced modifications to be combined and executed by a single succeeding thread, reducing synchronization overhead when multiple threads heavily contend with each other to modify shared data. In the Single-RCU+RLU with combining scheme, each thread takes four steps to complete a modification operation *f* (1) announce to the public that *f* needs to be completed; (2) load the current list of all announced operations that have not been completed; (3) iterate through the loaded operations and try to complete all of them using a single RCU+RLU synchronization; and (4) try commit the local results of all the loaded operations. Irrespective of whether the commit is successful or not, consider *f* is completed and move on. Note that flat-combining is applied only to modification operations as single RCU+RLU already allows read-only operations to always complete independently of the other concurrent operations.

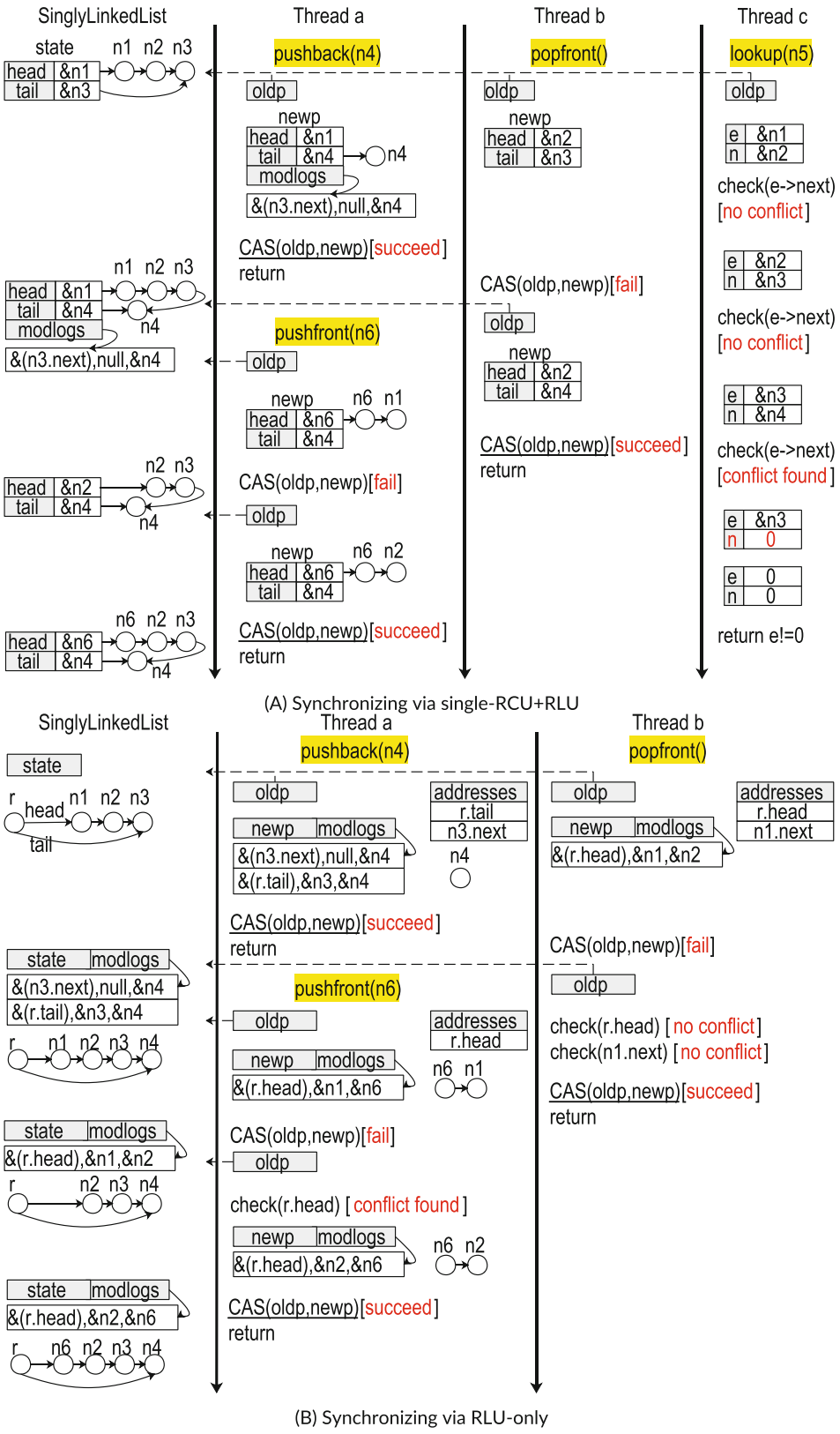


FIGURE 5 Synchronizing via single-RCU+RLU vs RLU-only, (A) Synchronizing via single-RCU+RLU, (B) Synchronizing via RLU-only.

Figure 6 illustrates data declarations for the singly-linked list abstraction in Figure 1 to store information of its announced modifications (line 17), each entry including an operation code and a list of arguments (line 16). Each thread announces its new modification simply by modifying its dedicated entry inside the array *active* (line 20), which is shared among all threads. As the thread completes each publicly announced modification *m*, it modifies the *applied* and *retvals* arrays (line 24–25) inside its local copy of the RCU object. Each thread can determine whether an arbitrary modification *m* announced by thread *i* has been already completed by checking whether *applied*[*i*] == *active*[*i*]. Similar to P-SIM,[29] we implemented both the *active* and *applied* arrays by using bit vectors.

Figure 7 illustrates the single-RCU+RLU with combining scheme. The *handle_req* function provides a case to handle each member operation, where the large switch-case block (lines 9–26) details the singly-linked list example. To generalize single-RCU+RLU with combining for any data structure type, each member operation is provided a copy of the RCU object *newp* and all modifications are redirected through *newp*. The outermost *while*(*true*) loop at lines 30–49 (which implements single-RCU+RLU) is invoked to process and collectively complete all ongoing requests of shared

```
1 // Types for combining write operations
2 enum op_code {PUSHBACK, POPFRONT};
3
4 struct pushback_arg_type {T o;};
5 struct popfront_arg_type {T result;};
6 struct popfront_ret_type {T result;};
7
8 union arg_type {
9    pushback_arg_type pushback_arg;
10   popfront_arg_type popfront_arg;};
11
12 union ret_type {
13   popfront_ret_type popfront_ret;};
14
15 // Announced request by each thread
16 struct req_type {op_code op; arg_type arg;};
17 req_type requests[N_THREADS];
18
19 // Stores the modification announced by each thread.
20 op_stat active[N_THREADS];
21
22 typedef struct List_state {
23    ...
24    op_stat applied[N_THREADS]; // The completed modifications announced by each thread.
25    ret_type retvals[N_THREADS]; // The return values announced by each thread.
26 } List_state;
```

FIGURE 6 Data structures for combining modifications of a singly-linked list.

```
1 // Announce a new active operation
2 void announce_req(int id, op_code op, arg_type arg) {
3    requests[id].op = op; requests[id].arg = arg;
4    active[id].update(); }
5
6 bool handle_req(List_state* newp, int id) {
7    op_code op = requests[id].op;
8    arg_type arg = requests[id].arg;
9    switch(op) {
10   case PUSHBACK: T o = arg.pushback_arg.o;
11      Node<T>* e = new Node<T> (o); ++newp->count;
12      if(newp->tail==0) new->head=newp->tail=e;
13      else { newp->new_modlog(&(newp->tail->next), newp->tail->next, e);
14             newp->tail = e;}
15      break;
16   case POPFRONT:
17      if (newp->count>0) {
18         newp->retvals[id].popfront_ret.result=newp->head->value();
19         Node<T>* head_next = newp->head->next;
20         if(!check_conflict(newp, &newp->head->next, head_next)) return false;
21         newp->head = head_next;
22         newp->count --;
23         if (newp->count==0) newp->tail = 0;
24      }
25      break;
26   ...}
27   newp->applied[id] = active[id]; return true;
28 }
29
30 ret_type handle_combined_req() {
31   List_state *oldp=0, *newp=allocate_state();
32   while(true) {
33   try {
34      oldp = load_current_state(MOD_RCU);
35      oldp->modlog_apply();
36      if(active[tid]==oldp->applied[tid])return oldp->retvals[tid];
37      newp->copy(oldp);
38      for(int id = 0; id < N_THREADS; id=id+1) {
39          if (active[id]!=newp->applied[id]) {
40              if(!handle_req(newp, id)) throw exception;
41          }
42      }
43      if(state.compare_exchange_weak(oldp, newp))  {
44         newp.modlog_apply();
45         return newp->retvals[tid];
46      }
47   } catch(const ModLogConflict& c){}
48   }
49 }
```

FIGURE 7 Methods to announce and combine requests.

data modifications. In particular, after each thread *t* loads the address of the most current RCU object, it first checks whether its own request has already been completed by other threads by comparing whether *applied*[*t*] = *active*[*tid*] (line 36). If yes, it can find its result stored at *retvals*[*tid*]; otherwise it collects all the active modifications across all threads and try complete them together (lines 38–42).

Figure 8 shows the new implementations used to replace the original sequential *pushback* and *popfront* operations in the abstraction. Figure 9 illustrates how two concurrent threads carrying out these modifications interact with each other. Here when thread (b) fails to publicize its modifications and starts over, it determines that its request has already been completed and returns immediately, after comparing its entries inside the *applied* and *active* arrays.

RLU-only

Here dynamic conflict detection is used to synchronize all shared addresses 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 10 illustrates details of this scheme. Here a new local array is declared (at line 1) to accumulate all the shared addresses read/modified. This array is updated immediately before each shared address needs to be accessed and is checked for conflicts (e.g., lines 7–10). At the end of the synchronized operation, if no modification log has been created (line 11), the operation has not modified any shared data (hence is read-only) and can 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. 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.

```
 1 void pushback(const T& o) {
 2    arg_type arg;
 3    arg.pushback_arg.o = o;
 4    announce_req(tid, PUSHBACK, arg);
 5    handle_combined_req();
 6    return;
 7 }
 8
 9 void popfront(T& result) {
10    arg_type arg;
11    arg.popfront_arg.result = result;
12    announce_req(tid, POPFRONT, arg);
13    ret_type ret = handle_combined_req();
14    result = ret.popfront_ret.result;
15    return;
16 }
```

FIGURE 8 Example: synchronizing a singly-linked list with combining technique.

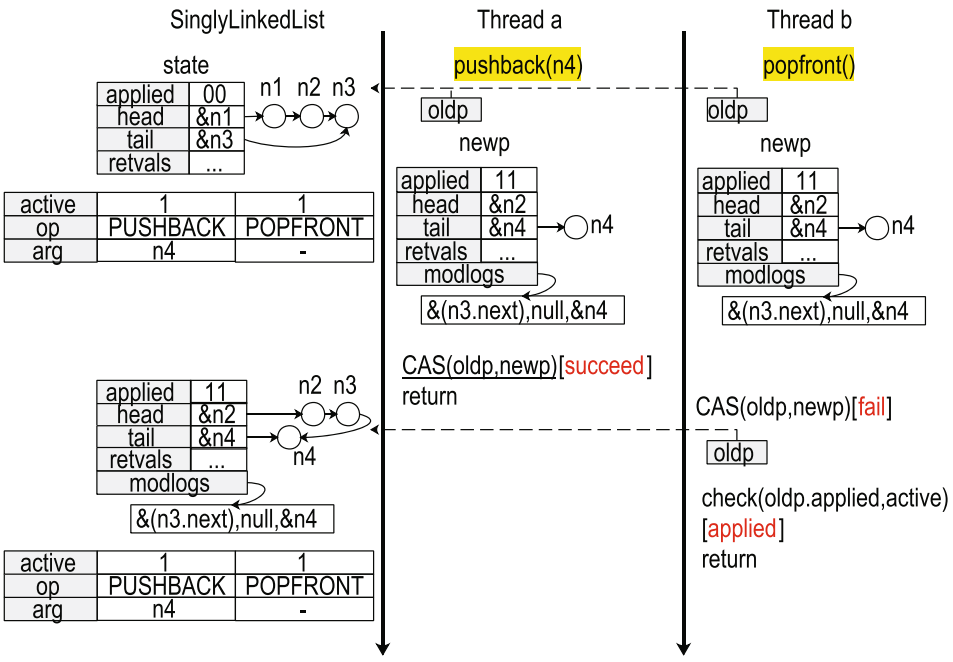


FIGURE 9 Synchronizing with combining technique.

```
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
7   newp->copy(oldp);
8   ... addresses.push_back(addr);
9       check_conflict(oldp,addr,*addr);/*is addr out-of-date?*/
10  ... addresses.push_back(addr);
11      newp->new_modlog(addr,oldv,newv); ...
12  if (newp->modlog_empty()) break; /*no modification*/
13  else if (state.compare_exchange_strong(oldp,newp))
14      { newp->modlog_apply(); break; }
15  else {
16   while (true) {
17     oldp=load_current_state(MOD_RLU);oldp->modlog_apply();
18     newp->copy(oldp);
19     for (std::vector<void*>::const_iterator p = addresses
20          .begin(); p != addresses.end(); ++p)
21        { void* addr = *p;
22          check_conflict(oldp,addr,*addr); /*addr out-of-date?*/}
23     if (state.compare_exchange_strong(oldp,newp))
24        { newp->modlog_apply(); break; } }
25   break; }
26 } catch(const ModLogConflict& e){}
27 }
```

FIGURE 10 Synchronizing via RLU only.

Figure 5B 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.

Multi-RCU+RLU

The shared data are partitioned into multiple disjoint groups, each group independently synchronized via the single-RCU+RLU scheme (with or without combining of operations). To use this scheme, each data item in the abstraction must belong to exactly one of the independently synchronized 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. Multi-RCU+RLU synchronization can be used when a data abstraction contains multiple independent groups of data. For example, a hash table may be implemented as a large array of independently operating buckets, each bucket mapping to a distinct hash key. Processes concurrently modifying or reading the hash table can independently look up the bucket that they intend to access. Then only those that intend to access the same bucket need to be synchronized among each other, while threads that access other distinct buckets can operate independently.

Dynamic conflict detection

For each shared address synchronized via RLU logs, for example, the address *n3.next* in Figure 5B, our RLU synchronization scheme dynamically detects possible conflicts, for example, other concurrent modifications to *n3.next*, by keeping track of its latest version of committed modification logs, specifically the version number of the shared *state* object immediately after a thread commits its modifications. This version number is recorded inside a unique wrapper object allocated to hold the latest committed modlog for each shared address. 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 most up-to-date object 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 or RLU-only, all the modification logs committed later than the version of RCU object held by *f* are examined, and if the address being modified is amid these logs, a conflict is detected, and an exception is thrown, causing *f* to abort due to its obsolete RCU object.

3 | CORRECTNESS GUARANTEE

Our nonblocking synchronization schemes are correct in that they generate concurrent data structures whose operations are *linearizable* and they guarantee lock-free progress. Given a concurrent operation *f* over shared data *D*, we define a *state-read point* of *f* to be the atomic statement within *f* where the state of memory locations in *D* are read. Similarly, we define the *linearization point* of *f* to be the atomic statement within *f* where the result of the overall operation is determined, and all of its modifications to shared data are visible to other threads. Given *n* concurrent operations $f_1, f_2, \ldots, f_n$, the evaluation is represented by an arbitrary interleaving of SR_i and LR_i pairs, where i = 1, … , n, SR_i and LR_i are the state-read point and linearization point of f_i (SR_i is guaranteed to always precede LR_i in all interleavings). The evaluation is *linearizable* and thus properly synchronized, if the arbitrary interleaving is equivalent to a sequential ordering of (SR_i, LR_i) pairs, where each state-read point immediately precedes its linearization point (i.e., evaluations of operations are not overlapped).

Note that since all the synchronization schemes in our compiler are implemented using CAS, ∀ concurrent operation f_i, its linearization point is represented by a CAS, which succeeds only if the address of shared data stored in pointer p equals to that read in the state-read point for p (i.e., no other concurrent threads have modified p). We show that for each of the synchronization schemes that an arbitrary interleaving is equivalent to a sequential ordering of (SR_i, LR_i) pairs.

Lemma 1. *The operations of a concurrent data structure synchronized using our single-RCU scheme are linearizable.*

Proof. Referring to Figure 4 (text in red), the state-read point occurs when the pointer to the contiguous region of shared data is read on line 5 (*pushback*), line 16 (*is_empty*), and line 22 (*lookup*). The linearization point for *pushback* occurs when the CAS succeeds on line 11 of Figure 4. The linearization point for *is_empty* occurs at the state-read point on line 16 of Figure 4 since the value loaded affects the outcome of the return statement on line 17 of Figure 4. The linearization point for *lookup* occurs at the state-read point on line 22 of Figure 4 since the value loaded determines the content of *oldp*. The *is_empty* and *lookup* methods are guaranteed to be linearizable because the state-read point and linearization point for each method are the same atomic instruction. The *pushback* method is guaranteed to be linearizable because the CAS in *pushback* corresponding to linearization point LR_i will only succeed if no writes occur to the RCU object between the state-read point SR_i of the RCU object and LR_i. Since an interleaving of subsequent state-read points are commutative, SR_i corresponding to LR_i can be reordered with respect to other subsequent state-read points so that it is ordered immediately before LR_i, forming a sequential ordering of the (SR_i, LR_i) pair. ∎

Lemma 2. *The operations of a concurrent data structure generated using the single-RCU+RLU synchronization scheme are linearizable.*

Proof. Referring to Figure 4 (all text), the state-read point occurs when the pointer to the contiguous region of shared data is read on line 5 (*pushback*), line 16 (*is_empty*), and line 22 (*lookup*). Since *modlog_apply*, detailed on lines 15–21 of Figure 2, is called immediately after the state-read point for all operations, it is guaranteed that all logged modifications will be completed prior to performing local updates to the shared data. The linearization point for *pushback* occurs when the CAS succeeds on line 11 of Figure 4 since this is the instant in which the *pushback* is publicized to other threads and establishes its order of effect due to all operations being required to call *modlog_apply* after an atomic load. The *pushback* method is guaranteed to be linearizable because the CAS in *pushback* corresponding to linearization point LR_i will only succeed if no writes occur to the RCU object between the state-read point SR_i of the RCU object and LR_i due to *check_conflict*. Although *modlog_apply* may be in progress by another thread when the CAS succeeds, the updates by *modlog_apply* modify the shared data structure itself and not the RCU object. Therefore, reads to the RCU object commute with updates from *modlog_apply* performed by the other threads. Since an interleaving of subsequent state-read points and updates from *modlog_apply* by other threads are commutative, SR_i corresponding to LR_i can be reordered with respect to other subsequent state-read points and *modlog_apply* updates by other threads so that it is ordered immediately before LR_i, forming a sequential ordering of the (SR_i, LR_i) pair.

For *lookup*, *check_conflict* is called which ensures that all the committed RLU logs are correctly observed by dynamically tracing the version of each shared memory address that has been modified, checking each shared memory access for conflicts, and using its value (which is considered valid) only if the version number of the value matches that of the RCU struct being used by the accessing thread. Since both the content of *oldp* and the outcome of *check_conflict* are established when *modlog_apply* returns, the linearization point LR_i for *lookup* is line 22 of Figure 4. The linearization point LR_i for *is_empty* occurs when *modlog_apply* returns on line 17 of Figure 4 because all logged modifications are guaranteed to be completed at this instant, which affects the outcome of the return statement on line 17 of Figure 4. The *is_empty* and *lookup* methods are guaranteed to be linearizable because all operations call *modlog_apply* immediately after the state-read point SR_i of the RCU object. The state-read point SR_i commutes with the *check_conflict* function when called by other threads for *lookup* because the *check_conflict* function only validates that no updates have been made to the loaded RCU object and therefore makes no updates to the RCU object itself. Using this reasoning along with the reasoning for commutative updates provided for *pushback*, SR_i corresponding to LR_i can be reordered with respect to other subsequent state-read points, *modlog_apply* updates by other threads, and *check_conflict* calls by other threads so that it is ordered immediately before LR_i, forming a sequential ordering of the (SR_i, LR_i) pair. ∎

Lemma 3. *The operations of a concurrent data structure generated using the RLU-only synchronization scheme are linearizable.*

Proof. Referring to Figure 10, the state-read point SR_i occurs when the pointer to the contiguous region of shared data is read on line 5 of Figure 10. If the operation is read-only, the linearization point LR_i occurs when *modlog_apply* returns on line 5 of Figure 10 since completing the logged modifications could affect the outcome of *check_conflict*. If the operation is not read-only, the linearization point LR_i occurs on line 11 of Figure 10 if the CAS is successful on the first attempt. If the operation is not read-only and the CAS on line 11 of Figure 10 is not successful on the first attempt, the linearization point occurs on line 23 of Figure 10 after re-validation of the addresses saved in the local array. Using the reasoning for commutative updates provided for single-RCU+RLU synchronization

in Lemma 2, the RLU-only scheme is guaranteed to be linearizable because SR_i corresponding to LR_i can be reordered with respect to other subsequent state-read points, *modlog_apply* updates by other threads, and *check_conflict* calls by other threads so that it is ordered immediately before LR_i, forming a sequential ordering of the (SR_i, LR_i) pair. ∎

Lemma 4. *The operations of a concurrent data structure generated using the multi-RCU+RLU synchronization scheme are linearizable.*

Proof. Since the abstraction is partitioned into multiple groups of independent data that are each synchronized by the single-RCU+RLU scheme, the state-read point SR_i, linearization point LR_i, and formation of the sequential ordering of the (SR_i, LR_i) pair as described for each method generated by the single-RCU+RLU scheme hold for the multi-RCU+RLU scheme. ∎

Lemma 5. *The dynamic conflict detection function preserves linearizability for operations of a concurrent data structure generated using either single-RCU+RLU, RLU-only, or the multi-RCU+RLU synchronization scheme due to asynchronous observation of the publicized logs by the threads.*

Proof. A version number is maintained for address *e* and the RLU logs, where the version for *e* identifies the latest version of the RLU logs that have modified *e*. A version number is also maintained for the RCU object that is assigned according to the real-time ordering of the successful CAS operations. If the RCU object in use by a thread is the same as the most recent RCU object held by the shared RCU atomic pointer, no conflict exists since the thread is referencing an RCU object that is still up-to-date. Otherwise, the thread's ongoing function *f* is examined. If *f* is synchronized according to the single-RCU+RLU scheme or multi-RCU+RLU scheme, a *ModLogConflict* exception is thrown. If *f* is synchronized according to the RLU-only scheme, a *ModLogConflict* exception is only thrown if the latest RCU version that has modified *e* is later than the version of the thread's RCU object. These actions ensure linearizability because *f* is aborted due to the thread's RCU object being out-of-date. If *f* is a read-only operation, the RLU logs with a version later than the thread's RCU object version are examined to recover the original value for *e*. This action ensures linearizability because the read can trace back to the original value for *e* so that the read takes effect prior to modifications by the RLU logs with a version later than the thread's RCU object version. In all cases, the operations of a concurrent data structure generated using either single-RCU+RLU, RLU-only, or the multi-RCU+RLU synchronization scheme handles conflicting observation of the publicized logs in a way that preserves linearizability. ∎

Lemma 6. *The operations of a concurrent data structure generated using the single-RCU+RLU with combining technique are linearizable.*

Proof. When extending single-RCU+RLU synchronization scheme with combining technique, concurrent operations that are combined and associated to the same RCU object have already been locally linearized, while concurrent operations associated to different RCU objects are still linearized in the same way as single-RCU+RLU without combining technique. The state-read point SR_i for a *pushback* operation occurs when *newp* → *tail* is read on line 12 of Figure 7 or when *newp* → *tail* → *next* is read on line 13 of Figure 7. The local linearization point LP_i for *pushback* is on line 12 or line 14 when *newp* → *tail* is set to *e*. The linearization point for *pushback* is on line 21 of Figure 2 after `compare_exchange_weak` successfully updates *state* to *newp* on line 43 of Figure 7 and *modlog_apply* is called on either line 35 of Figure 7 (helping thread) or line 44 of Figure 7 (calling thread).

The state-read point SR_i for a *popfront* operation occurs when *newp* → *head* → *value*() is read on line 18 of Figure 7 and when *newp* → *head* → *next* is read on line 19 of Figure 7. The local linearization point LP_i for *popfront* is on line 21 when *newp* → *head* is set to *head_next*. The linearization point for *popfront* is on line 43 of Figure 7 when `compare_exchange_weak` successfully updates *state* to *newp*.

All threads call *modlog_apply* prior to starting a new combining operation, ensuring that all logged operations linearize prior to the new operation requests. Since *modlog_apply* is performed sequentially, the resulting outcome of *modlog_apply* is guaranteed to be equivalent to a legal sequential history. All operations stored in a *modlog* are performed in a fixed order on line 17 of Figure 2 to ensure that the same result is obtained regardless of which thread performs the updates.

The *handle_combined_req* function is performed sequentially by all active threads, where each thread stores local updates in *newp*. Only one thread will successfully perform a CAS to update *state* to *newp*. Since each thread performs *handle_req* sequentially in a fixed order according to thread id prior to performing a CAS to update *state* to *newp*, the state-read points SR_i for each request will immediately precede LR_i in all cases, enabling the SR_i and LR_i points for the *pushback* and *popfront* methods to be arranged to form a sequential ordering of the (SR_i, LR_i) pairs. The *check_conflict* function called by *popfront* will detect an out-dated version of *state*, which will prevent the CAS from succeeding if the updates made to *newp* are out-dated.

When a *pushback* is invoked, it calls *announce_req* on line 4 of Figure 8 to post its operation and arguments in the *requests* array at position *tid*. A *pushback* then calls *handle_combined_req* on line 5 of Figure 8. Since the *pushback* operation is guaranteed to be completed by some thread before the *pushback* returns (line 35 of Figure 7 for a helping thread or line 44 of Figure 7 for the calling thread), and the history of combined operations is equivalent to a legal sequential history, *pushback* is linearizable.

When a *popfront* is invoked, it calls *announce_req* on line 12 of Figure 8 to post its operation and arguments in the *requests* array at position *tid*. A *popfront* then calls *handle_combined_req* on line 13 of Figure 8. Since the *popfront* operation is guaranteed to be completed by some thread before the *popfront* returns, and the history of combined operations is equivalent to a legal sequential history, *popfront* is linearizable. ∎

Lemma 7. *The operations of a concurrent data structure generated using the single-RCU, single-RCU+RLU, single-RCU+RLU with combining, RLU-only, and multi-RCU+RLU synchronization schemes are lock-free.*

Proof. The *pushback* method generated using single-RCU (Figure 4, text in red) is lock-free because failure of the CAS on line 11 of Figure 4 implies that some other thread successfully applied CAS to update *state* to *newp* and exited the while loop. The *is_empty* method generated using single-RCU (Figure 4, text in red) is lock-free because the atomic load is guaranteed to complete in a finite number of steps. The *lookup* method generated using single-RCU (Figure 4, text in red) is lock-free because the list contains a finite number of elements and the while loop on line 24 of Figure 4 terminates once it finds the element of interest or reaches the end of the list.

The *pushback* method generated using single-RCU+RLU (Figure 4, all text) is lock-free because the *modlog_apply* function, called on line 5 and line 12 of Figure 4, iterates through a finite number of *ModLog* objects and performs a CAS on line 21 of Figure 2 to perform the updates in the *ModLog* object. Similar to single-RCU, failure of the CAS on line 11 of Figure 4 implies that some other thread successfully applied CAS to update *state* to *newp* and exited the while loop. The *is_empty* method generated using single-RCU+RLU (Figure 4, all text) is lock-free because the atomic load on line 16 of Figure 4 requires only a single atomic step and *modlog_apply* function, called on line 17 of Figure 4, iterates through a finite number of *ModLog* objects. The *lookup* method generated using single-RCU+RLU (Figure 4, all text) is lock-free because the *modlog_apply* function, called on line 22 of Figure 4, iterates through a finite number of *ModLog* objects and the while loop on line 24 of Figure 4 is guaranteed to terminate since the list has a finite number of elements. It follows that the multi-RCU+RLU synchronization is also lock-free because the data is partitioned into multiple disjoint groups and independently synchronized with single-RCU+RLU.

The *announce_req* method generated using single-RCU+RLU with combining (Figure 7) is lock-free because it updates the *requests* and *active* arrays at position *id* with the operation information in a finite number of steps. The *handle_req* method generated using single-RCU+RLU with combining (Figure 7) is lock-free because a requested *pushback* is added at the end of the list of operations in a finite number of steps and a requested *popfront* retrieves the value from the head of the list of operations and checks for a conflict in a finite number of steps. The *handle_combined_req* method generated using single-RCU+RLU with combining (Figure 7) is lock-free because the number of threads traversed by the for loop on line 38 of Figure 7 is bounded and failure of CAS to update *state* to *newp* on line 43 of Figure 7 implies that another thread successfully updated *state* and exited the while loop. The *pushback* and *popfront* methods generated using single-RCU+RLU with combining (Figure 4) are lock-free because *announce_req* and *handle_combined_req* are lock-free.

The RLU-only synchronization (Figure 10) is lock-free because a failed CAS to update *state* to *newp* on either line 11 or line 23 of Figure 10 implies that another thread successfully updated *state* and exited the while loop. ∎

4 | OVERALL STRATEGIES OF AUTOMATION

Our compiler automatically applies the synchronization designs in Section 2 to a sequential data abstraction, represented by a self-contained C++ class 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 ineligible for conversion and the unsafe operation is moved to a private section of the converted abstraction. Our compiler first analyzes and preclassifies all data references inside the abstraction to select a most appropriate synchronization scheme (single-RCU, single-RCU+RLU, single-RCU+RLU with combining, 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[30,31] and ABA prevention,[32] synthesized from scratch, are then inserted to ensure correctness of the final code.

Pruning of unsafe operations

A concurrent abstraction must not allow addresses of internal data to escape to the outside, for example, by passing them to calls to external functions or by returning them as results, as these addresses can become obsolete at any moment and cause memory corruption. Given an arbitrary abstraction *x*, all of its internal data, including all member variables of *x*, must be made private so that they can be synchronized exclusively inside *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. Our compiler automatically performs this pruning step by first identifying such unsafe operations via straightforward inspections of each operation implementation and then moving them to the private sections of the abstraction, before proceeding to synchronize the remaining safe operations.

For example, the private section of the C++ class in Figure 1 shows a subset of the original interface functions of a sequential singly-linked list, for example, First, Last, and Next, which have been considered unsafe operations 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 moving similar unsafe interface functions to their private sections.

Classification and partitioning of data

Before determining how to synchronize an abstraction, our compiler classifies RCU-synchronized data to include all variable references that are never aliased (that is, the data either resides 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.

Our compiler automatically partitions all member variables of an abstraction, including those that have array types, into independently synchronized groups. It first 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.

Selection of synchronization schemes

Among all the 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. As long as RLU-only is not selected, the compiler attempts to partition the internal data of the abstraction into distinct groups, so that different groups can be safely synchronized independently via the multi-RCU+RLU. Finally, if single-RCU+RLU is selected as the scheme to synchronize an abstraction, flat-combining is added as an optional optimization if collectively all the update operations of the abstraction contain only a relatively small number of RLU-synchronized modifications (say ≤ 16) even when combined.

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 five 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 4;
- READ_RLU: if *f* reads both RCU and RLU synchronized data, without modifying anything, it is synchronized by following the *lookup* function in Figure 4;
- 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 10.
- MOD_RCU: if *f* modifies shared data, and the overall scheme is single-RCU or single-RCU+RLU, it is synchronized by following the *pushback* method in Figure 4.

- MOD_RCU_WITH_COMBINING: if *f* modifies shared data, and the overall scheme is single-RCU+RLU with combining, it is synchronized by following the *pushback*/*popfront* methods in Figure 8.

ABA prevention and lock-free garbage collection

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. The ABA problem[32] occurs when 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. To address the ABA problem, 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, for each abstraction, our compiler-generated code preallocates 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. 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, that is, 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.

5 | COMPILER AUTOMATION

Our compiler serves to automate the strategies in Section 4 and are developed by adapting standard techniques, for example, those shown in Figure 11 and summarized below, 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 automate the synchronization of an arbitrary data abstraction *x*, the sequence of steps outlined in Figure 11 are applied by our compiler to transform the input C++ class, with the output of each step immediately used as input to the next step. If any of these steps fails to completely analyze the input code, for example, due to undefined abstraction functions whose bodies are not provided, the transformation is aborted to preserve correctness guarantees. More details about these seven steps are summarized below.

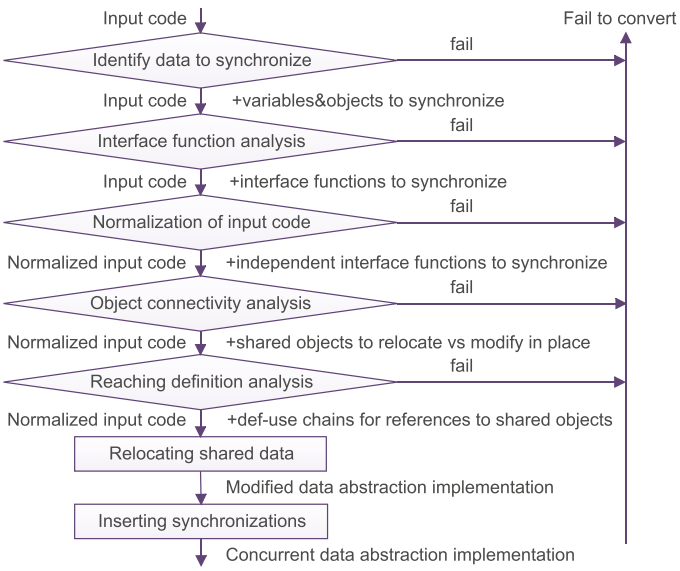


FIGURE 11 Example connectivity graphs.

Algorithm 1. Object connectivity analysis

Function *analyze_connectivity(g_0 : initial graph which connects pointers to the objects they might point to, input: a single interface function of a C++ class) : returns a map that associates each basic block in input to a connectivity graph representing points-to relations among objects*

```
    cfg = build_control_flow_graph(input)
    foreach basic block b ∈ cfg do pt[b] = ∅;
    pt[entry_node(cfg)] = g0; change=true while change == true do
        change = false foreach basic block b in cfg do
            g1 = pt[b]; pt[b] = ∅ /* recompute pt[b]*/ foreach predecessor p of b in cfg do
                pt[b] = pt[b] ∪ {points_to_modification(b, pt[t])}
            if pt[b] ≠ g1 then change = true;
    res = ∅;                                          // collect graphs at the exit of function
1   foreach basic block b ∈ cfg that has no successor do
2       res = res ∪ pt[b]
3   return res
```

- *Structural analysis and normalization*, which identifies member variables and public methods of *x* that do not contain unsafe operations to synchronize. Then, the step normalizes *x* to be processed by later stages, 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.
- *Overall scheme selection*, which selects an overall scheme (single-RCU, single-RCU+RLU, Single-RCU+RLU with combining, RLU-only, or multi-RCU) to synchronize the entire abstraction. The selection is based on the partitioning of RCU- vs RLU- synchronized data in the data relocation step and whether the overall number of RLU-synchronized data modified by each operation can be limited by a small constant.
- *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, MOD_RLU, or MOD_RCU_WITH_COMBINING) best suited for its purpose, based on results of its function side-effect analysis and the overall scheme selected to synchronize the entire abstraction. Then using the result of data relocation analysis, memory references inside *f* are modified, for example, 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, for example, 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 (globally within a single function).[33] In particular, each algorithm tries to analyze each individual interface function of *x* in isolation while assuming arbitrary values for nonlocal variables (the most conservative assumption). It then simply combines the results from analyzing all functions to represent all possible situations for *x*. By restricting our analysis and optimization scope to each individual self-contained abstraction, our approach ensures the cost of all the analysis algorithms, including the safety check before and after each optimization, and the pruning of unsafe interface functions, can be fully automated without incurring steep compile-time overhead.

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* function at line 10. The algorithm terminates when the results for all cfg nodes no longer change. The points-to

Algorithm 2. Relocation Analysis

Function *data_analysis(c: input C++ class being analyzed, ocg: object connect graph that models points-to relations of pointer objects of c) : returns cp_vars: the set of member variables of c to relocate, and log_vars: the set of unknown memory references of c objects.*

```
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)};
```

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 12 shows the resulting connectivity graphs from using this algorithm to analyze the *pushback* function at lines 4–7 of Figure 1A. Here the entry node b_0 is associated with the initial connectivity graph, which includes two unknown objects, *n*1 and *n*2, pointed to by the *head* and *tail* member variables respectively. The compiler knows nothing about *n*1 and *n*2, so they can be both null or aliased to each other. Node *b*1 has *b*0 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 *n*3, pointed to by *e*. Node *b*2 and *b*3 both have *b*1 as predecessor but modify the pointers differently. They therefore each have their own connectivity graphs. These graphs are then collected together at node *b*4, which has both *b*2 and *b*3 as predecessors. The connectivity graphs of *b*4 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.

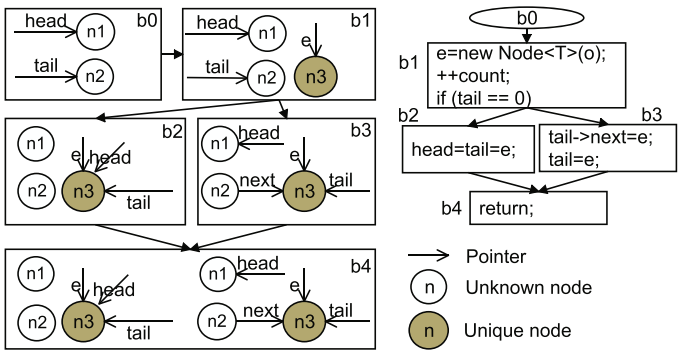


FIGURE 12 Example connectivity graphs.

Since the algorithm terminates when the set of connectivity graphs assigned to each cfg node no longer changes, termination 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.

Side effect analysis and reaching definition analysis

The side effect analysis identifies the memory references read and modified by each interface function of *x*. The reaching definition analysis discovers the set of data modifications that can reach each memory reference in *x*. Both use standard program analysis algorithms available in compiler books.[33] 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 versus pointer variables—each of them is by default assumed to refer to an arbitrary memory region.

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*, where variables in different partitions are 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 nonaliased member variables and categorizes the object as nonaliased (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 12, the *n*3 unique node can be reached either through the head or the tail member variables of the list, so *n*3 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).

6 | EXPERIMENTAL EVALUATION

We have implemented our prototype compiler using the POET[24] interpreted language for developing specialized source-to-source compilers, and have used our compiler to automatically convert eight sequential C++ classes collected from two existing open-source projects: the ROSE compiler[34] and the Tervel framework.[35] Our compiler is general purpose in that all of its algorithms take an arbitrary C/C++ class as input. Out of the eight C++ classes, shown in the second column of Table 1, our compiler was able to successfully synchronize 40-100% of their original interface functions, while pruning the others by moving them to the protected sections of the synchronized classes. The unsuccessful constructions were due to accepting shared pointers as parameters or returning pointers to internal shared data structures, discussed in pruning of unsafe operations in Section 4.

To evaluate the overall effectiveness of our compiler-driven approach for synchronizing data abstractions, we have compared the performance of our automatically synchronized data abstractions both with those manually crafted by experts and those automatically synchronized by state-of-the-art STM frameworks. We additionally studied the impact of these data abstractions by using them inside an existing application benchmark, Dedup, from PARSEC.[17]

TABLE 1 Benchmark and workload configurations.

Workload	Auto-generate RSTM[36]	TBB[37]	FC[28] & SIM[29]	Boost[38] Back-off	CDS[39]
Lightweight write-only	Ringbuffer Stack (array-based) Deque (list-based) Singly (Slist)/Doubly linked list (Dlist)	Queue	Queue Stack	Michael-Scott queue (MSQueue)[7] Treiber stack[40]	
Heavyweight write-only& mostly-read	HashSet (array of lists) Binary tree (unbalanced) Multi-dimensional list[41]	Hash-map	-		Michael's hashset[9] Herlihy's skiplist[2] Ellen's BST[13]

TABLE 2 System configuration.

Server	Model name	Sockets	Cores	Threads	GCC	Boost	TBB	CDS	RSTM
Intel Xeon Phi	Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz	1	64	256 (4 hyper threads per core)	4.8.5	1.58.0	9.103	2.3.2	v7
Xsede	Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz	2	28	28	4.8.5	1.53.0	11.103	2.3.2	v7

6.1 | Experimental configuration

Shown in the first two columns of Table 1, we classified the sequential data structures we collected into three workloads and manually written a set of micro-benchmarks to test each workload, detailed below.

- The *light-weight write-only* workload, which includes the five abstractions in the first row of Table 1. To test this workload, we treat each abstraction as a concurrent queue/stack and concurrently invoke two operations, a *push* and a *pop*, each with 50% probability;
- The *heavy-weight write-only* workload, which tests the heavier weight abstractions in the second row of the table, by treating each concurrent data abstraction as a set/map and concurrently modifying it via two operations, an insert and an erase, each with 50% probability;
- The *heavy-weight mostly-read* workload, which 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 contains enough elements for meaningful operations and that each test runs sufficiently long for timing stability, our micro-benchmarks initialize each data structure with a large number (set to be $2.56 * 10^6$) of elements before spawning a preconfigured number of threads to collectively invoke a predetermined number (again set to $2.56 * 10^6$) of the preselected operations. Idle loops are inserted to emulate the behavior of local computations performed by real world applications in between accessing/modifying shared data structures. All workloads are measured by their throughput, computed by dividing the number of operations completed with the average time taken to complete the operations.

By default, our compiler used single-RCU+RLU with combining to synchronize the first five abstractions in Table 1, 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 evaluate the multi-RCU+RLU synchronization scheme, we 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. These slight modifications enable the compiler to use multi-RCU+RLU to synchronize the array-based abstraction implementations.

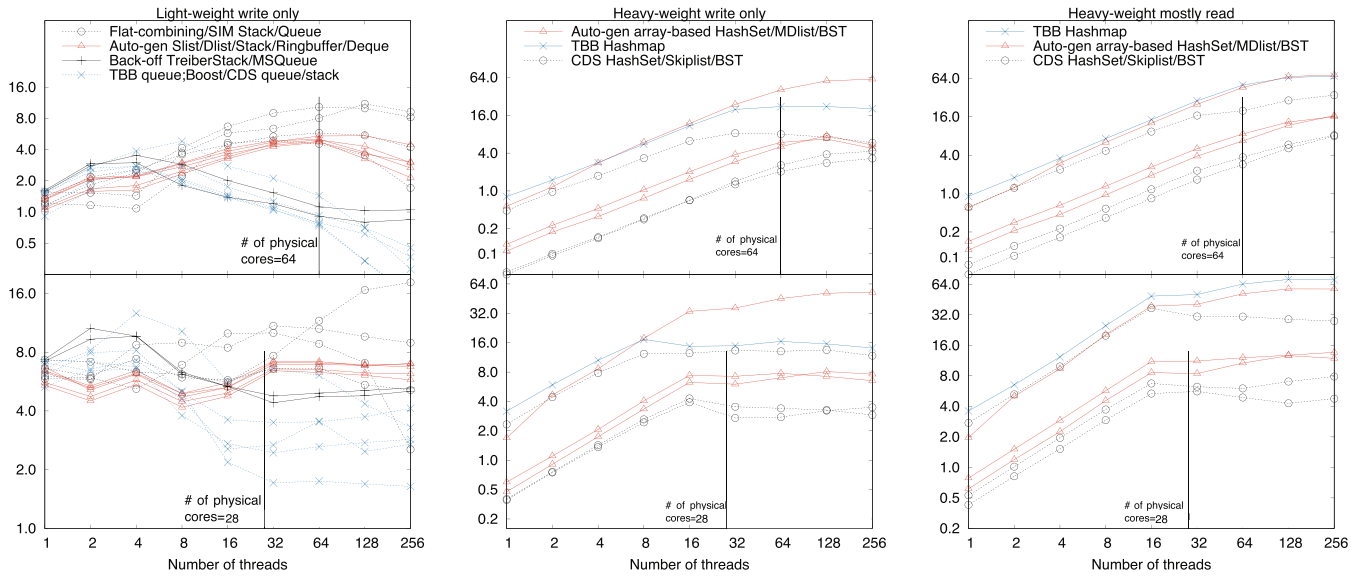
The last four columns of Table 1 show the collection of concurrent data structures crafted by experts. To compare with STM, for each of the eight sequential data abstractions in Table 1, we also manually synchronized them via RSTM,[36] one of the fastest obstruction-free STMs. Experimentation is used 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) when using different numbers of threads.

We also evaluated our collection of synchronous data abstractions by using an open-source micro-benchmark suite *Synchrobench*.[42] Synchrobench is different from our micro-benchmarks in three respects: (1) instead of invoking a predetermined number of operations, Synchrobench is configured to run for a predetermined amount of time, during which parallel threads attempt to complete as many concurrent operations as possible. (2) Synchrobench inserts no idle loops in between operations, resulting in heavier contention. (3) Instead of selecting operations with pre-specified probabilities, Synchrobench dynamically decides which operation to invoke next, while trying to match the operation ratio to the portion of operations that successfully modified the data structure. Similar to Reference 42, we configured synchrobench to run each benchmark for 5000 milliseconds. Update ratios of write-only and mostly-read workloads are 100% and 10% respectively. Each data structure is initialized with 65536 elements. Element keys are selected from [0, 132072].

Additionally, we evaluated the performance impact of the different concurrent data abstractions on the *Dedup* application benchmark from PARSEC.[43] We evaluated all the benchmarks 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.[44] Both machines run CentOS Linux as the underlying operating system. All data structure implementations (including the sequential, automatically synchronized, and manually synchronized variations) 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. The system configuration is provided in Table 2.

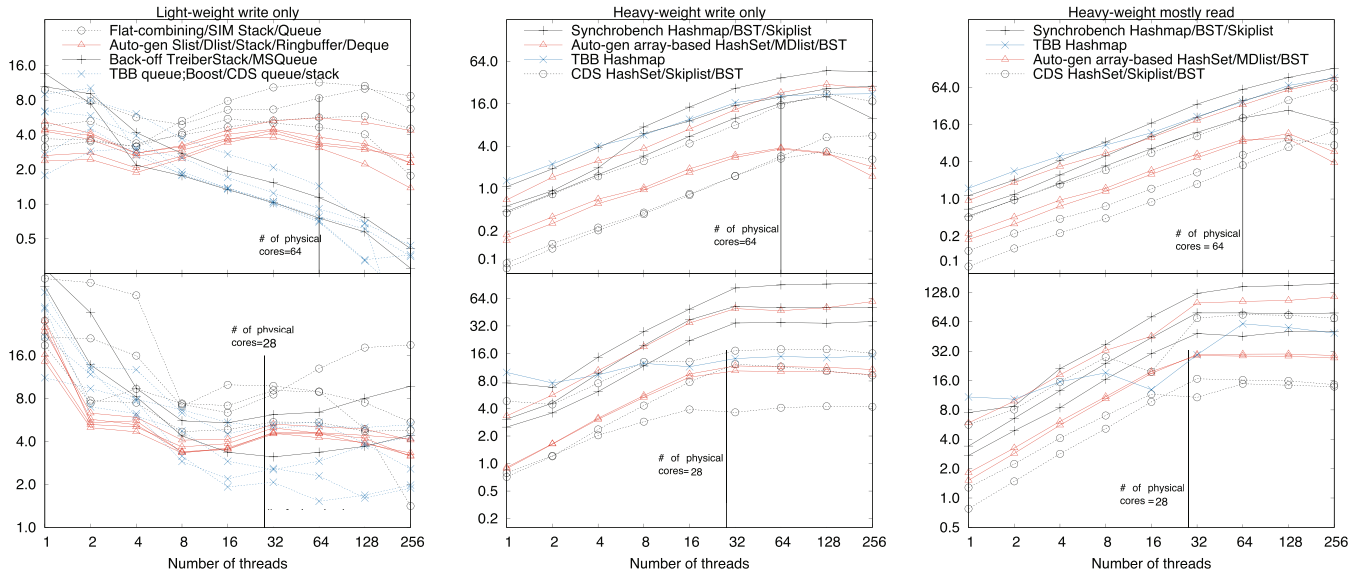
6.2 | Comparing with manually synchronized implementations

Figure 13 (microbenchmarks) and Figure 14 (Synchrobench) compare the performance of our compiler-synchronized implementations with those manually crafted by experts. Besides those listed in Table 1, the performance reported additionally includes three manually implemented concurrent



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

FIGURE 13 Throughput (y-axis: million ops/second) of compiler-synchronized and manual implementations reported by our micro-benchmarks * Top graphs: on the Intel Xeon Phi server; Bottom graphs: On the Xsede server.



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

FIGURE 14 Throughput (y-axis: million ops/second) of compiler-synchronized and manual implementations reported by Synchrobench * Top graphs: on the Intel Xeon Phi server; Bottom graphs: On the Xsede server.

data structures which are part of Synchrobench, specifically a hashmap based on Harris's lock-free linked lists,[45] Natarajan's lock-free binary search tree,[46] and Fraser's lockfree skiplist.[47] To improve readability, we grouped the implementations so that those in the same group are displayed using the same line style. The members in each group are listed in decreasing order of their performance. Different groups are also ordered in decreasing order of their collective performance.

The performance of our compiler-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. The main observation is that the multi-RCU synchronization (used 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

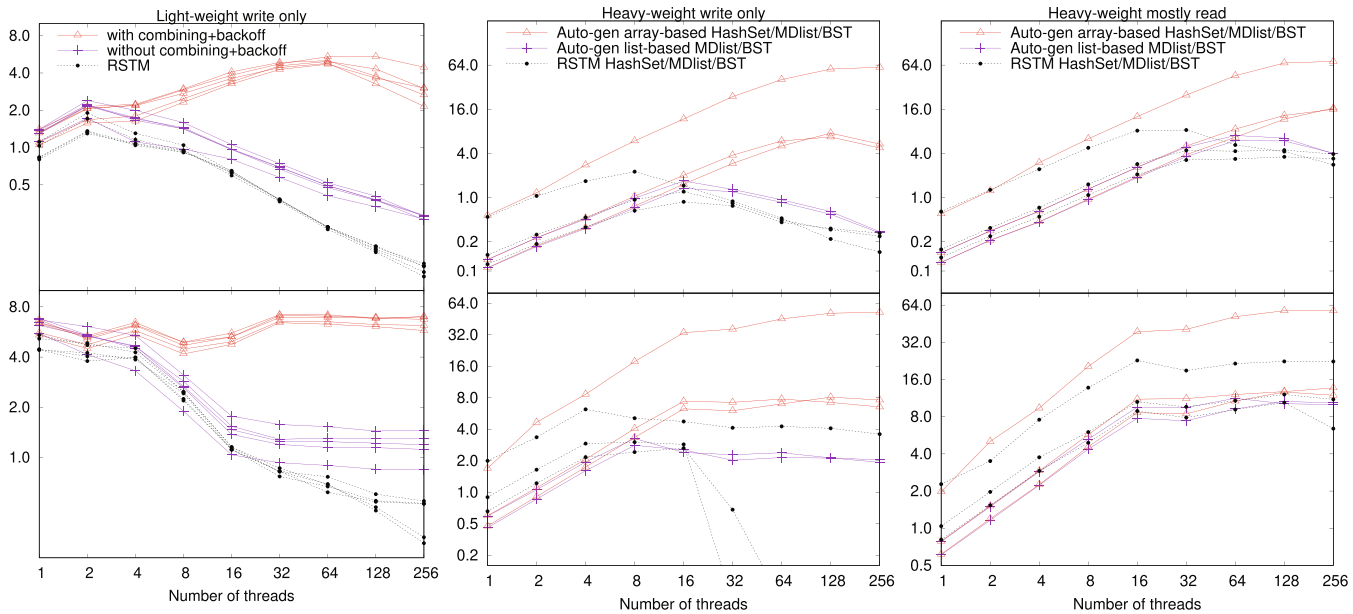
case of the hashset, each RCU object synchronizes all updates to the same hash key. The auto-synchronized MDlist and BST lag behind the hashset, because a constant number of RCU objects are used at the top level, in contrast to the much larger array of atomic pointers used in the hashset.

For the lightweight write-only workload, our compiler used Single-RCU+RLU with combining to synchronize the five light-weight data structures. Since it sequentializes all the updates and has to maintain RCU copy and RLU logs to support universal construction, it lags behind the manual implementations when using a small number of threads. However, it scales better than about half of the manual implementations when the number of threads exceeds 32 since it combines multiple updates to reduce overhead and the tuning of sleep-time reduces contention when there is no concurrency among the operations. The best performing implementations, specifically flat-combining stack and SIM queue, are manually synchronized with combining techniques.

When comparing the throughput achieved by the differently implemented synchronous data abstractions on both the Intel Xeon Phi and Xsede, Figures 13 and 14 demonstrate that while the lightweight workloads generally lack scalability, most of heavy-weight workloads can attain close to linear speedups when their numbers of threads are less than or equal to the number of physical cores of the servers. The speedups then gradually reach their plateaus afterwards. All workloads generally attain better throughput on the Xsede server, which has fewer number of physical cores (28 vs 64) than the Intel Phi server. The enhanced throughput on the Xsede server is likely due to the higher bandwidth supported by its memory and network connections among the different processor cores.

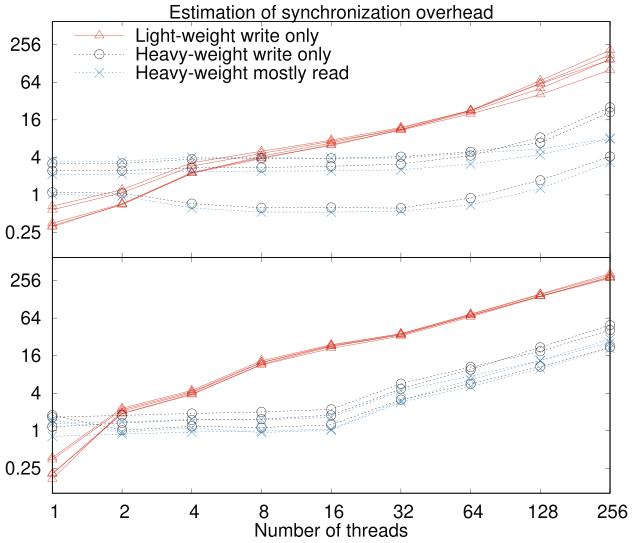
6.3 | Comparing with STM-synchronized abstractions

Figure 15 compares our compiler-synchronized implementations, both with and without the sleep-time-before-retry tuning and the flat-combining technique, with performance attained when we manually used STM to synchronize the original sequential implementations. 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. However, even without these optimizations, our compiler synchronized implementations generally performed better than the RSTM implementations, which performed competitively only under the heavy-weight mostly-read workload. For the lightweight workload, even without tuning of sleep time or flat-combining, our compiler-synchronized implementations performed much better than RSTM, as our RCU-based synchronizations are generally much lighter weight than the RLU-only synchronizations used by STM. For the heavyweight write-only workload, our RLU-only implementations demonstrated similar behavior as the RSTM implementations.



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

FIGURE 15 Comparing Throughput (y-axis: million ops/second) with STM-synchronized abstractions * Top graphs: on the Intel Xeon Phi server; Bottom graphs: On the Xsede server.



* Top: Intel Xeon Phi; Bottom: Xsede

FIGURE 16 Synchronization overhead of compiler-generated code * Top: Intel Xeon Phi; Bottom: Xsede.

6.4 | Estimating synchronization overhead

Figure 16 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. The plot of the cumulative overhead of all the threads shows that the overhead for the light-weight write-only workload scales linearly as the number of threads increases because the single-RCU+RLU scheme sequentializes their concurrent modifications. However, for the heavy-weight write-only and mostly-read workloads, the overhead stayed low and constant irrespective of the increasing number of threads, until there are more threads than the number of CPU cores.

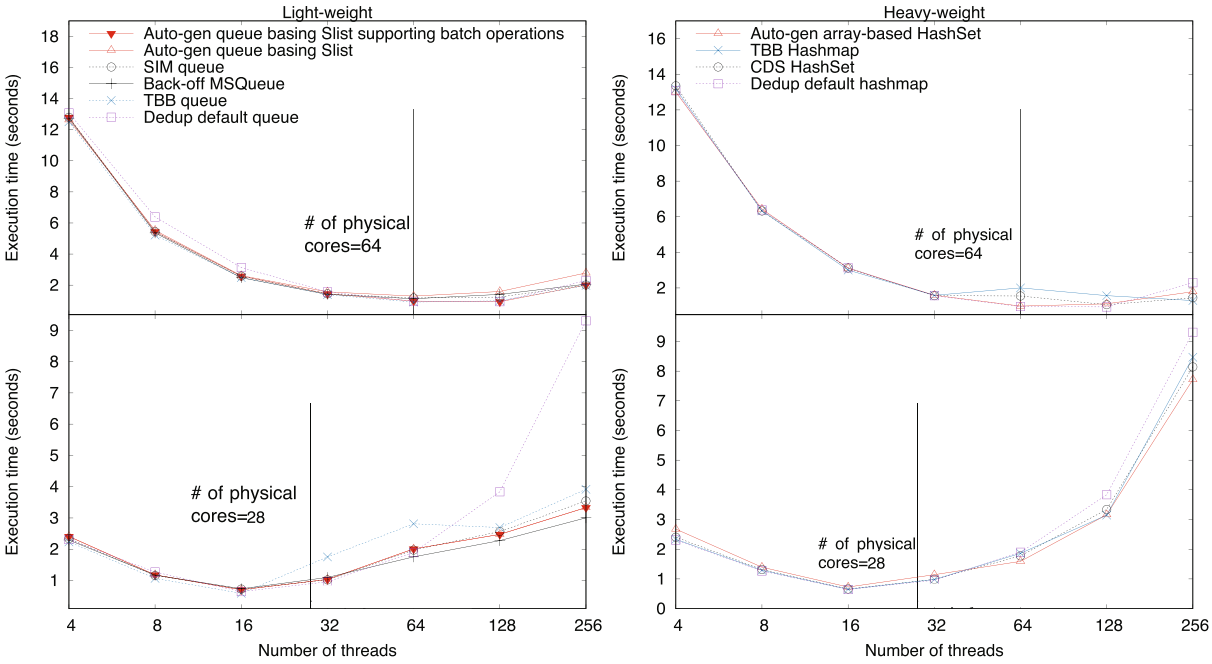
6.5 | Application study: Dedup

Dedup, an application benchmark from PARSEC,[17] uses a variety of data structures to compress an input data stream through five pipeline stages, out of which the middle three stages are parallelized. Data chunks are transferred between stages via four task queues, and a global hash table is used to de-duplicate data chunks. In our application study, we manually modified Dedup to alternatively use the data structures from our lightweight workload as task queues and to use data structures from our heavy-weight workload as implementations to the original global hash table in Dedup. Each Dedup task queue uses multiple lock-based ringbuffers to reduce contention, each ringbuffer shared by at most 4 threads, and each enqueue/dequeue operation tries to push/fetch 20 data chunks at a time. When using our lightweight workload data abstractions in Dedup, we used each queue object throughout its lifetime without decomposing it into multiple smaller queues, to stress the contention. To allow exploitation of a higher degree of concurrency, we have modified the benchmark to use two levels of arrays to support faster indexing and reordering. Further, instead of dedicating a predetermined set of threads to each pipeline stage, we modified the application to allow a single pool of threads to work on all middle three parallel stages in a Round-Robin fashion to enhance concurrency.

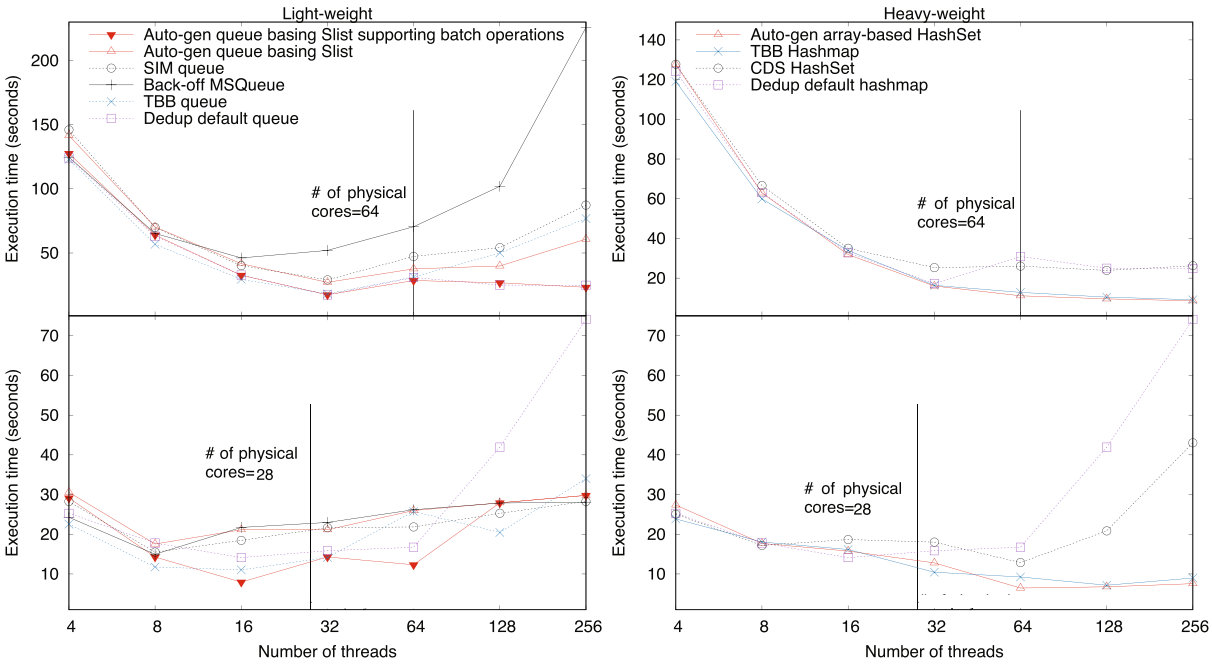
Since the original dedup application batches enqueue/dequeue operations to operate on 20 data chunks at a time, we use this opportunity to test the ability of our compiler-driven synchronization in supporting automatic synchronization of nonconventional APIs. To do this we manually extended our sequential singly linked list (Slist) implementation in Figure 1 to additionally support batch enqueue/dequeue operations. Our compiler is then invoked to automatically synchronize this new API via single-RCU+RLU without combining since thread-private buffers are needed to save multiple results of the batch operations.

6.5.1 | Performance impact of data structure implementations

Figure 17 shows the elapsed time of running the Dedup application when using different concurrent data abstractions to implement its task queues and global hash table. Two sets of input data are used respectively in the evaluations: (1) the *native* input provided by PARSEC, a 671 MB



(A) Evaluated using the *native* input



(B) Evaluated using *enwik9* as input

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

FIGURE 17 Impact of using different data structure implementations in Dedup (after algorithmic optimization), (A) evaluated using the *native* input, (B) evaluated using *enwik9* as input, * Top graphs: on the Intel Xeon Phi server; Bottom graphs: On the Xsede server.

ISO image of Fedora Core 6; and (2) the *enwik9* input, the first 10^9 bytes (953 MB) of the English Wikipedia dump.[48] Dedup has partitioned the *native* input into 369950 chunks, out of which 201596 are duplicated, and compressed with Zstandard,[49] a fast real-time lossless compression algorithm released by Facebook. For the *enwik9* input, Dedup has partitioned it into about 21 million data chunks, among which about 2 million chunks are duplicated. The deduplicated data are then compressed by using *Smaz*,[50] an efficient lightweight algorithm designed to compress small text strings.

From Figure 17A, when using its *native* input, the runtime of Dedup did not vary significantly when using alternative data structure implementations, due to the relatively low degree of contentions among the concurrent threads, which are insufficient to stress test the scalability of the synchronizations among concurrent data accesses. This result is consistent with previous findings in similar application studies.[51]

From Figure 17B, however, when tested with the much large data set *enwik9*, the runtime of Dedup varied more significantly when with data structure implementations of varying efficiencies and scalabilities. In particular, the lowest execution time for Dedup is observed when using our compiler-generated hash map and the TBB hash map, which significantly outperformed the default lock-based hash table implementation of Dedup. When tested with alternative task queues, Dedup scaled well with large numbers of threads when using implementations that support combining technique, including the SIM queue, and our compiler-synchronized singly-linked list. It performed best when using our compiler-synchronized queues that support batch operations. The original Dedup queue performed well because of its support for the batch APIs, although limitations of its lock-based synchronization show when using large numbers of threads.

6.5.2 | Impact of manual modifications to Dedup

When studying the impact of using different data structure implementations in Dedup, we have applied two algorithmic modifications to Dedup: dynamic load balancing to allow all threads to work on all parallel pipelining stages, and a faster data chunk reordering algorithm through the use of two-level arrays. Figure 18 plots the performance impact of these algorithm modifications to Dedup, when applied along,

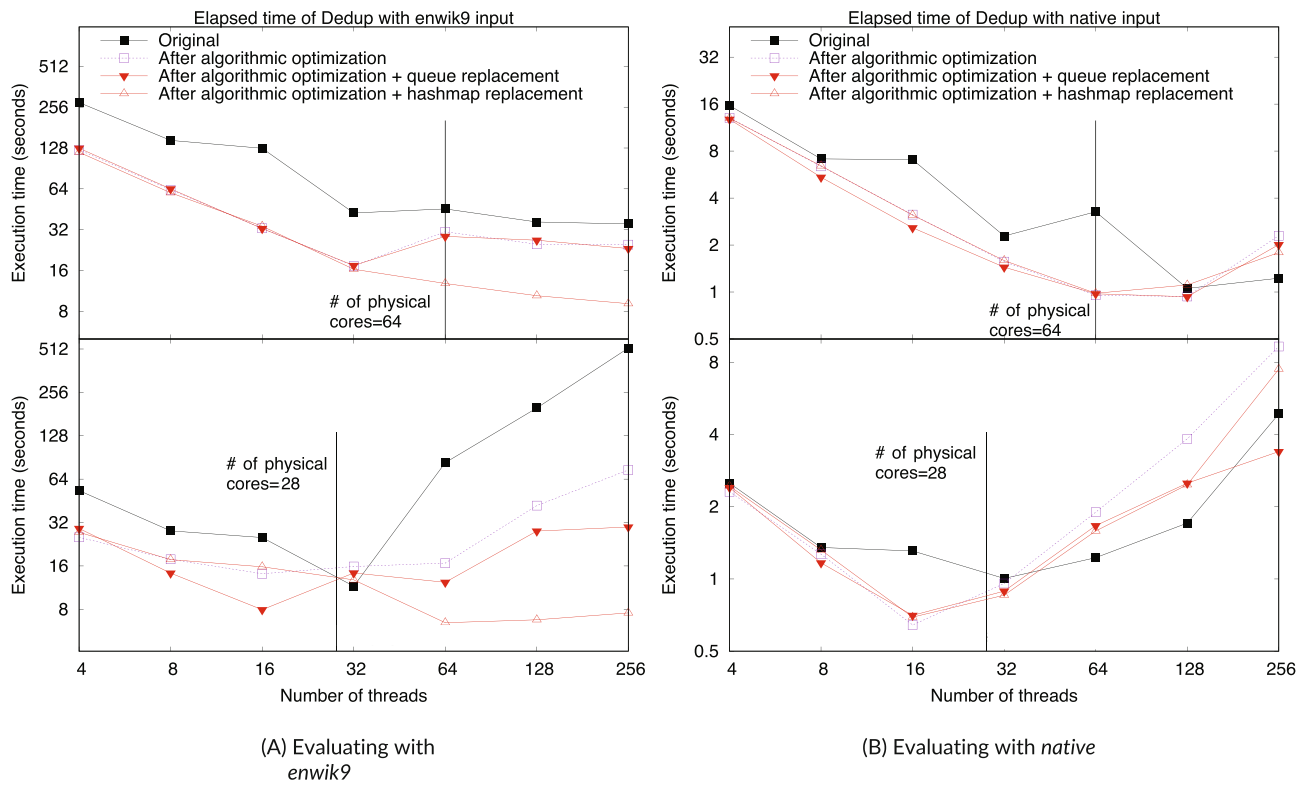


FIGURE 18 Impact of different levels of optimizations in Dedup, (A) evaluating with *enwik9*, (B) evaluating with *native*, * Top: Intel Xeon Phi; Bottom: Xsede.

combined with alternative task queue implementations, and when combined with alternative hash table implementations. From the results we can see that our algorithm-level modifications overall improves the performance of Dedup, especially when combined with more scalable data structure implementations. In particular, compared to the default Dedup implementation, it produced better or similar best attained performance on all combinations of input data and platforms. It triggered negative performance slowdowns only when using small input data (the *native* input) combined with an unnecessarily large number of threads. The algorithm modifications can significantly speed up Dedup on its own, with the data structure optimizations producing additional speedups when a high level of contention is present among the threads.

7 | RELATED WORK

In contrast to existing work on the manual design of nonblocking concurrent data structures, for example, queues,[7,8] lists,[9,10] maps,[9,11] and trees,[12-15] 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 nonblocking algorithms vs locking and observed that efficient data-structure-specific nonblocking 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[52] of read-copy-update[26] and read-log-update.[27] Our retrieval of older values through saved RLU-logs for read-only functions is similar in ideas to the multi-versioning extension of RLU.[53] Single-word compare and swap (CAS) has been widely used as a primitive for implementing lock-free or wait-free synchronizations.[52,54] Our compiler-driven approach can be potentially used to automate other advanced synchronization mechanisms as well, for example, fine-grained locking,[15] flat combining,[28] among others.[55-59]

Herlihy and Moss[60] first proposed transactional memory as a hardware architecture, the materialization of which includes Intel TSX[61] and AMD ASF.[62] 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[63] and was later extended to support dynamically sized data structures,[19] conditional critical regions,[64] transactional monitors,[65] composition of blocking transactions,[20] among others.[66] Modern STM systems have been implemented both by using lock-based[67,68] and nonblocking synchronizations,[12,19,21,64] Steep runtime cost is required when implementing STM.[12,21,69] Many optimizations, for example, obstruction-free synchronization,[19] transaction logging,[20] fine-grained object disambiguation,[64] 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[70] and automated fixing of concurrency bugs.[71-73] The sketch synthesis algorithm[74] tries to iteratively complete the sketch of a concurrent data structure from developers until reaching a given criteria. Vechev et al.[75,76] automatically inferred synchronizations to avoid interleavings that violate user specifications. Our compiler requires only the sequential implementation of C++ classes and aims to automate nonblocking synchronization.

8 | 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.

FUNDING INFORMATION

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

DATA AVAILABILITY STATEMENT

Data available on request from the authors.

ORCID

Christina Peterson https://orcid.org/0000-0002-8070-7633

REFERENCES

1. Dijkstra EW. Solution of a problem in concurrent programming control. *Commun ACM*. 1965;8:569.
2. Herlihy M, Shavit N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc.; 2008.
3. Karlin A, Li K, Manasse M, Owicki S. Empirical studies of competitve spinning for a shared-memory multiprocessor. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. ACM; 1991:41-55.
4. Mellor-Crummey JM, Scott ML. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans Comput Syst*. 1991;9:21-65.
5. Michael MM, Scott ML. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J Parallel Distrib Comput*. 1998;51:1-26.
6. Zhang J, Yi Q, Dechev D. Automating non-blocking synchronization in concurrent data abstractions. *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE; 2019:735-747.
7. Michael MM, Scott ML. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. ACM; 1996:267-275.
8. Herlihy M, Luchangco V, Moir M. Obstruction-free synchronization: Double-ended queues as an example. *ICDCS'03: Proceedings of the 23rd International Conference on Distributed Computing Systems*. IEEE Computer Society; 2003:522.
9. Michael MM. High performance dynamic lock-free hash tables and list-based sets. *SPAA'02: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM; 2002:73-82.
10. Heller S, Herlihy M, Luchangco V, Moir M, Scherer WN III, Shavit N. A lazy concurrent list-based set algorithm. *Proceedings of the 9th International Conference On Principles of Distributed Systems*. Springer-Verlag; 2005:3-16.
11. Bronson NG, Casper J, Chafi H, Olukotun K. Transactional predication: High-performance concurrent sets and maps for STM. *PODC'10: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM; 2010:6-15.
12. Fraser K, Harris T. Concurrent programming without locks. *ACM Trans Comput Syst*. 2007;25(2):5-es. doi:10.1145/1233307.1233309
13. Ellen F, Fatourou P, Ruppert E, Breugel vF. Non-blocking binary search trees. *PODC'10: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. ACM; 2010:131-140.
14. Crain T, Gramoli V, Raynal M. A contention-friendly binary search tree. In: Wolf F, Mohr B, Mey aD, eds. *Euro-Par 2013 Parallel Processing*. Springer Berlin Heidelberg; 2013:229-240.
15. Arbel M, Attiya H. Concurrent updates with RCU: Search tree as an example. *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*. ACM; 2014:196-205.
16. Synchrobench Micro-Benchmark Suite. GitHub. 2020 https://github.com/gramoli/synchrobench
17. *The PARSEC Benchmark Suite*. Princeton University; 2020. https://parsec.cs.princeton.edu/ Retrieved 10/12/2020.
18. Shavit N, Touitou D. Software transactional memory. *Distrib Comput*. 1997;10:99-116.
19. Herlihy M, Luchangco V, Moir M, Scherer WN. Software transactional memory for dynamic-sized data structures. *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*. ACM; 2003:92-101.
20. Harris T, Marlow S, Peyton-Jones S, Herlihy M. Composable memory transactions. *PPoPP'05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM; 2005:48-60.
21. Marathe VJ, Scherer WN, Scott ML. Adaptive software transactional memory. *DISC'05: Proceedings of the 19th International Conference on Distributed Computing*. Springer-Verlag; 2005:354-368.
22. Yi Q. Downloading The POET Compiler Prototyping Language. Github 2020 https://github.com/qingyi-yan/POET
23. Yi Q, Seymour K, You H, Vuduc R, Quinlan D. POET: Parameterized optimizations for empirical tuning. *POHLL'07: Workshop on Performance Optimization for High-Level Languages and Libraries*. IEEE; 2007.
24. Yi Q. POET: A scripting language for applying parameterized source-to-source program transformations. *Soft Pract Exp*. 2012;42(6):675-706.
25. Herlihy MP, Wing JM. Linearizability: A correctness condition for concurrent objects. *ACM Trans Program Lang Syst*. 1990;12(3):463-492.
26. McKenney PE, Slingwine JD. Read-copy-update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*. Citeseer; 1998:509-518.
27. Matveev A, Shavit N, Felber P, Marlier P. Read-log-update: A lightweight synchronization mechanism for concurrent programming. *SOSP'15: Proceedings of the 25th Symposium on Operating Systems Principles*. ACM; 2015:168-183.
28. Hendler D, Incze I, Shavit N, Tzafrir M. Flat combining and the synchronization-parallelism tradeoff. *SPAA'10: Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM; 2010:355-364.
29. Fatourou P, Kallimanis ND. A highly-efficient wait-free universal construction. *SPAA'11: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM; 2011:325-334.
30. Detlefs DL, Martin PA, Moir M, Steele GL. Lock-free reference counting. *Distrib Comput*. 2002;15:4.
31. Gidenstam A, Papatriantafilou M, Sundell H, Tsigas P. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans Parallel Distrib Syst*. 2009;20(8):1173-1187.
32. Dechev D, Pirkelbauer P, Stroustrup B. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. *ISORC'10: Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE Computer Society; 2010:185-192.
33. Cooper K, Torczon L. *Engineering a Compiler*. Morgan Kaufmann; 2004.
34. Quinlan D, Schordan M, Vuduc R, Yi Q. Annotating user-defined abstractions for optimization. *POHLL06: Workshop on Performance Optimization for High-Level Languages and Libraries*. Rhode Island; 2006.
35. Dechev D, LaBorde P, Feldman S. LC/DC: Lockless Containers and Data Concurrency: A Novel Nonblocking Container Library for Multicore Applications. 2013.
36. Marathe VJ, Spear MF, Heriot C, et al. Lowering the overhead of nonblocking software transactional memory. *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. University of Rochester; 2006.
37. Intel Corporation. Intel Threading Building Blocks Reference Manual. 2014.
38. Boost 1.56.0 Library Documentation. 2014.

39. khizmax aMK. Concurrent data structures library. sourceforge.net 2012.
40. Treiber RK. Systems programming: Coping with parallelism. *Research Report RJInternational Business Machines Incorporated*. Thomas J. Watson Research Center; 1986.
41. Zhang D, Dechev D. An efficient lock-free logarithmic search data structure based on multi-dimensional list. *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE; 2016:281-292.
42. Gramoli V. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: Cohen A, Grove D, eds. *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. ACM; 2015:1-10.
43. Bienia C, Li K. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*. Semantic Scholar; 2009.
44. Bridges User Guide. 2020.
45. Harris TL. A pragmatic implementation of non-blocking linked-lists. *DISC'01: Proceedings of the 15th International Conference on Distributed Computing*. Springer-Verlag; 2001:300-314.
46. Natarajan A, Mittal N. Fast concurrent lock-free binary search trees. *PPoPP'14: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery; 2014:317-328.
47. Fraser K. Practical lock-freedom. 2003.
48. Large Text Compression Benchmark. 2020.
49. Zstandard v1.4.5. 2020.
50. SMAZ—compression for very small strings. 2009.
51. Feldman SD, Bhat A, LaBorde P, Yi Q, Dechev D. Effective use of non-blocking data structures in a deduplication application. *SPLASH'13: Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*. ACM; 2013:133-142.
52. Herlihy M. A methodology for implementing highly concurrent data objects. *ACM Trans Program Lang Syst*. 1993;15:745-770.
53. Kim J, Mathew A, Kashyap S, Ramanathan MK, Min C. MV-RLU: Scaling read-log-update with multi-versioning. *ASPLOS'19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM; 2019:779-792.
54. Herlihy M. Wait-free synchronization. *ACM Trans Program Lang Syst*. 1991;13:124-149.
55. Turek J, Shasha D, Prakash S. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. *PODS'92: Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM; 1992:212-222.
56. Afek Y, Dauber D, Touitou D. Wait-free made fast. *STOC'95: Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*. ACM; 1995:538-547.
57. Turon A. Reagents: Expressing and composing fine-grained concurrency. *PLDI'12: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM; 2012:157-168.
58. Arbel M, Morrison A. Predicate RCU: An RCU for scalable concurrent updates. *PPoPP 2015: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM; 2015:21-30.
59. Martínez JF, Torrellas J. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. *ASPLOS X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM; 2002:18-29.
60. Herlihy M, Moss JEB. Transactional memory: Architectural support for lock-free data structures. *ISCA'93: Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM; 1993:289-300.
61. Hammarlund P, Martinez AJ, Bajwa AA, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*. 2014;34(2):6-20. doi:10.1109/MM.2014.10
62. Christie D, Chung JW, Diestelhorst S, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. *EuroSys'10: Proceedings of the 5th European Conference on Computer Systems*. ACM; 2010:27-40.
63. Shavit N, Touitou D. Software transactional memory. *PODC'95: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. ACM; 1995:204-213.
64. Harris T, Fraser K. Language support for lightweight transactions. *OOPSLA'03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*. ACM; 2003:388-402.
65. Welc A, Jagannathan S, Hosking AL. Transactional monitors for concurrent objects. *Proceedings of the European Conference on Object-Oriented Programming*. SpringerVerlag; 2004:519-542.
66. Moir M. Transparent support for wait-free transactions. *WDAG'97: Proceedings of the 11th International Workshop on Distributed Algorithms*. Springer-Verlag; 1997:305-319.
67. Saha B, Adl-Tabatabai AR, Hudson RL, Minh CC, Hertzberg B. McRT-STM: A high performance software transactional memory system for a multi-core runtime. *PPoPP'06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM; 2006:187-197.
68. Scherer WN, Scott ML. Advanced contention management for dynamic software transactional memory. *PODC'05: Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*. ACM; 2005:240-248.
69. Dice D, Shavit N. Understanding tradeoffs in software transactional memory. *CGO'07: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society; 2007:21-33.
70. Dig D, Marrero J, Ernst MD. Refactoring sequential java code for concurrency via concurrent libraries. *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society; 2009:397-407.
71. Jin G, Zhang W, Deng D, Liblit B, Lu S. Automated concurrency-bug fixing. *OSDI'12: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association; 2012:221-236.
72. Černý P, Henzinger TA, Radhakrishna A, Ryzhyk L, Tarrach T. Efficient synthesis for concurrency by semantics-preserving transformations. *CAV 2013: Proceedings of the 25th International Conference on Computer Aided Verification—Volume 8044*. Springer-Verlag New York, Inc.; 2013:951-967.
73. Lin H, Wang Z, Liu S, Sun J, Zhang D, Wei G. PFix: Fixing concurrency bugs based on memory access patterns. *ASE 2018: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM; 2018:589-600.

74. Solar-Lezama A, Jones CG, Bodik R. Sketching concurrent data structures. *PLDI'08: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM; 2008:136-148.
75. Vechev M, Yahav E, Yorsh G. Abstraction-guided synthesis of synchronization. *POPL'10: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM; 2010:327-338.
76. Vechev M, Yahav E. Deriving linearizable fine-grained concurrent objects. *PLDI'08*. ACM; 2008:125-135.

How to cite this article: Zhang J, Yi Q, Peterson C, Dechev D. Compiler-driven approach for automating nonblocking synchronization in concurrent data abstractions. *Concurrency Computat Pract Exper*. 2024;36(5):e7935. doi: 10.1002/cpe.7935