Motivation

The background is that, as modern architectures evolve to include more and more CPU cores, multi-threaded programming becomes more important. When programming with multiple threads, we should carefully manage shared data among parallel threads to avoid performance bottleneck. The problem is that, efficiently synchronizing concurrent accesses to shared data can be very challenging. Because on the one hand blocking synchronization techniques such as locks often provide limited fault tolerance and progress guarantee; while on the other hand nonblocking synchronization techniques, such as lock-free techniques, may require extensive code modifications to achieve high performance. To help developers address this problem, we propose a compiler-driven approach to automate non-blocking synchronization through compiler tech…

Automating Non-Blocking Synchronization

The basic idea is to Use a source-to-source compiler to Automatically convert sequential C++ data structure code to lock-free implementations. So that legacy sequential code can be easily reused and converted for multi-threaded use scenarios. Our compiler supports automated synchronization of a single data structure in the form of C++ class. By combining and adapting Read-Copy-update and Read-Log-Update, our compiler is able to support multiple synchronization strategies and will Automatically select the best strategy for each data structure at compile time. So that By restricting the scope of synchronization to a single data structure, our compiler can provide more advanced analysis and optimizations than state-of-the-practice

Non-Blocking Synchronization Strategies

The non-blocking synchronization strategies supported by our compiler are based on read-copy-update (RCU) and read-log-update (RLU). By combining and adapting them, our compiler is able to support four synchronization strategies which I will explain later. The only required hardware supports are atomic single-word CAS and total store ordering, which are widely supported by modern architectures.

Read Copy Update

To demonstrate how each synchronization strategy works, I use singly linked list as an example.
Here we have a singly linked list which has head and tail pointers pointing to the first and last node respectively. And we have a pop_front operation which wants to remove the head node n1 from the list, which means we need to modify the head pointer from n1 to the next node n2. The head and tail pointers can be easily copied. So to synchronize them with RCU, our compiler will define a new data type called RCU object to store the values of these two pointers in a continuous memory region. Read operation can access them directly, but to modify them, write operations, such as pop_front, need to first create a private copy of the shared RCU object, apply modifications locally to this private copy, and finally try to replace the shared RCU object with this modified copy through a single compare and swap. And we have the result data structure.

RCU+RLU

However, the internal nodes of the linked list such as ··· cannot be easily copied due to aliasing problem and expensive copying overhead. To modify them, We need to extend RCU with RLU. In this example, concurrently with pop_front(), we have another operation push_back, which wants to insert a new node n4 after the tail node n3. Same with pop_front, we still need to create a private copy of the shared RCU object, update the tail pointer locally, then to link n3 with n4, we need to create a modification log, saying we are going to modify its next pointer from empty to n4. This log will be publicized together with the private RCU copy through a single update and later the logged modification will be applied to shared data using compare and swap: find

the data location n3.next, compare its current value with the previously logged value empty, and update it with the new value n4. Now pop_front found that the RCU object has been modified when it tries to replace it. it has to start over, make a new private copy, helps push_back to finish the logged modification to make sure the shared data is up to date, and try to modify the head pointer again. So that finally, despite two operations begin at the same time, to the shared data it appears that pop_front begins after push_back ends.

RLU-only

Single RCU+RLU is the default option because it can always work correctly. But sometimes when none of the shared data can be easily synchronized via RCU or when there is more potential parallelism between concurrent operations, we can try the RLU-only strategy.

With RLU-only, we will synchronize all shared data, including the head and tail pointers, through modification logs. We still need to allocate an RCU object for each operation, which is mainly used to group all related modification logs for that operation. The overall pattern is also similar to RCU+RLU, First push_back publicizes its logs and applies the logged modifications to shared data, and then pop_front find the shared RCU object has been modified. The major difference is that, at this point, instead of starting over directly, pop_front will double check if it really conflict with push_back. In this example there is no conflict, so pop_front doesn't have to start over. It can still go ahead to publicize its logs and apply logged modifications. So that finally to the share data, it appears that pop_front and push_back progress in parallel. To conclude, the major difference between RLU-only and single-RCU+RLU is that, when one operation failed to replace the shared RCU object, RLU-only allows an additional double-check to avoid unnecessary retrying.

Multi-RCU+RLU

The basic idea of multi-RCU+RLU is that, when two parts of the shared data are never accessed together, we can separate them into disjoint groups and synchronize them via independently so that we can have more parallelism. Here is an example where we have a hash table which is essentially an array of separate linked lists. Nodes from different lists are never accessed together, so we can allocate one RCU object for each list and synchronize each list using Single-RCU+RLU independently.

Overall Compilation Strategy

The overall compilation strategy to automate nonblocking synchronization to convert a sequential data structure, takes 5 major steps. First our compiler will check the original code to remove problematic code pieces where our compiler cannot guarantee safe conversion through program analysis.

Then our compiler will perform program analysis to classify the internal data of the data structure into RCU-synchronized data that can be easily copied, and RLU synchronized data.

Then our compiler will try to partition the shared data into different groups that are never accessed together, so that we can use multi-RCU+RLU to synchronize them independently for more parallelism.

After data classification and partitioning, our compiler will select the synchronization schemes for different kinds of data. Single-RCU+RLU is the default option because it can always work correctly. If all data are RCU-synchronized, we will use the Single-RCU scheme. If all data are RLU-synchronized, we can use RLU-only. If the shared data can be partitioned, we will use multi-RCU+RLU. Finally our compiler will apply selected synchronization schemes by modifying source code to generate concurrent lock-free implementations.

Classification And Partition Of Data

The most challenging parts of program analysis are step 2 and 3 for classifying and partitioning shared data.

To determine whether a data reference is RCU-synchronized or RLU-synchronized, we need to check if the data is accessed through a unique name or path. If yes, it means the data can be easily copied and synchronized via RCU, otherwise it will be handled as RLU-synchronized data. Required program analysis for data classification includes Object connectivity analysis, side-effect analysis, reaching definition analysis.
To determine whether shared data can be partitioned into disjoint groups, we need to check if all data are reached only from member variables of the data structure class. if yes, it means the internal data can be potentially partitioned by grouping member variables, otherwise our compiler is unable to partition them.
Then our compiler will determine how to partition the internal data. If two member variables are never accessed together by the same functions, we can then separate them together with all internal data that can be reached only from them into distinct groups and synchronize them independently. Data partitioning is done through relocation analysis.

Experimental Setup

To evaluate the performance of our auto-generated lock-free data structures, we used benchmarks to compare our generated lock-free data structures with state-of-the-arts implementations manually written by experts and implementations based on RSTM which supports concurrent wait-free data access. Our benchmark initializes each data structure with 2 million elements and spawn parallel threads to collectively invoke 2 million pre-selected operations. The workload is large enough to get reliable performance results.
We use three workloads to collect performance results, including lightweight-writeonly, heavyweight-writeonly, heavyweight-mostlyread. Here writeonly and mostlyread represent different operation ratios. Write-only means all operations are write. Mostly-read means a combination of 90% read and 10% write.
For data structures such as queue, stack, and linked list, we use lightweight workload which invokes push and pop operations. This workload is called lightweight because each operation only access one end of the data structure. For search structures such as hash table, search tree and skiplist, we use the heavyweight workload which invokes insert delete and search operations. This workload is called heavyweight because each operation needs to search or go through the data structure.

Compare With Manual Implementations

First we compare our generated data structures with the state-of-the-arts implementations written by experts. The x axis of the figure shows the number of parallel threads. The y axis shows the normalized throughput of concurrent operations. Higher throughput means better performance. This figure shows the performance of queue, stack and linked lists evaluated with the lightweight-writeonly workload. The red lines represent our generated data structure. From the figure we can see that as number of threads increases, the generated Lightweight data structures eventually scale better than manual implementations, mainly because of the back-off mechanism which helps to reduce contention, except for flat-combining and sim implementations which used more efficient combining techniques. These two figures show the performance of hash table, search tree and skip list, evaluated with the heavyweight writeonly and heavyweight mostlyread workloads. From the figures we can see that The heavyweight data structures scale comparable to manual implementations, mainly because of data partitioning optimization for more parallelism.

Compare With RSTM Implementations

Then we compare our implementations with RSTM-based implementations. Even without optimization such as backoff and data partitioning, our generated data structures can still outperform RSTM.