Annales Mathematicae et Informaticae
Accepted manuscript
DOI: https://doi.org/10.33039/ami.2023.08.003
URL: https://ami.uni-eszterhazy.hu

Visualization of Read-Copy-Update synchronization contexts in C code

Endre Fülöp, Attila Gyén, Norbert Pataki

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
gamesh411@gmail.com
gyenattila@gmail.com
patakino@elte.hu

Abstract. The Read-Copy-Update (RCU) mechanism is a way of synchronizing concurrent access to variables with the goal of prioritizing read performance over strict consistency guarantees. The main idea behind this mechanism is that RCU avoids the use of lock primitives while multiple threads try to read and update elements concurrently. In this case, elements are linked together through pointers in a shared data structure. RCU is used in the Linux kernel, but there are user-space libraries which implement the technique as well. One of the user-space solutions is liburcu that is a C language library. Earlier, we defined our code comprehension framework for easing the development of RCU solutions. In this paper, we present our visualization techniques for the Microsoft's Monaco Editor.

AMS Subject Classification: 68W10 Parallel algorithms

1. Introduction

Read-copy-update (RCU) mechanism is used for synchronizing memory access in a way that guarantees deterministic read-access even during concurrent writes to the same memory region [4]. Unsynchronized access from multiple threads can lead to the evaluation of completely unexpected values, which in turn almost negates the programmers ability to reason about possible outcomes [11].

There are multiple families of solutions to this problem. One traditional solution is locking, where multiple threads are sequentially ordered at runtime, thus accesses to a memory region are mutually exclusive among threads. This can, however, lead

Submitted: July 30, 2023
Accepted: August 7, 2023
Published online: August 8, 2023

to performance degradations, live- and deadlock problems. Locking solutions use synchronization primitives like mutexes and various kinds of locks [5]. Another possible solution is lock-free programming, where synchronization is solved without explicit exclusion, eliminating most locking issues [9]. Many lock-free solutions use memory barriers and atomic variables [15]. RCU is a solution of higher abstraction level than those mentioned before. RCU can be implemented in the kernel- or in the user-space. Linux kernel uses data structures with RCU implementation since 2002 [13].

RCU can also be implemented in the user-space, one such library is *liburcu* written in C [3]. In order to provide synchronization using the liburcu library, the user must intersperse the application code with calls to library functions. In effect the side-effects of these invocations produce a context along the execution paths where accesses to a memory region are guaranteed to have desired properties. To help the comprehension of the synchronization provided by the library, we have devised a visualization technique. The goal of the proposed technique is to provide the users of the library a visual and interactive way of exploring the code, thus facilitating the correct and intended usage of the library. There is no silver bullet in software engineering [2]. However, visualization is an important aspect [6]. Visualization improves the comprehension in many ways [18]. Code comprehension often requires visualization on the top of the source code [10]. However, subtle details are in-use for more sophisticated approaches [16]. Our previous work presented our framework for the code comprehension of RCU contexts [7]. In the previous paper, we focus on framework, more precisely, the static analysis and Monaco Editor-related techniques. Unfortunately, the actual visualization has not been presented properly. In this paper, the major contribution belongs to the visualization of the contexts.

This rest of this paper is organized as follows. In Section 2, we provide a brief overview on related work. We present the Userspace RCU implementation and our static analysis methods in Section 3. Section 4 provides a brief explanation how the backend analysis techniques are defined in our earlier work. We present the approach of visualization in Section 5, and finally, this paper concludes in Section 6.

2. Related work

Visualizing concurrency aspects of programs can have the goal of assessing performance aspects of a particular solution [17]. One category of tools used to measure performance is sampling- and instrumenting profilers which are for both single- and multithreaded programs. These profilers produce aggregated performance statistics and/or traces of events which can be used for detailed performance analysis [1]. These statistics are consequently converted into visual representations like barcharts and flamegraphs to provide an overview and highlight the proportions of each program parts contribution to a given metric. Compared to these visualizations, we propose a technique based on static analysis instead of dynamic profiling to reason about the structure of the RCU implementation. Another important aspect

is that the analysis done by the RCU visualization technique is more qualitative in nature.

3. Userspace RCU implementation

3.1. RCU overview

RCU is implemented in program code as a set of API calls (free function calls in case of the *liburcu* C library), which implements concurrent publication of modifications on shared data, subscription for insertion into shared data structures, waiting for readers to complete their executions and finally to maintain different versions of the same data [4]. This API is geared towards read-heavy uses, where updates of values and structured data are relatively less frequent, and where consistency guarantees are not critical. Memory usage is another concern, as multiple version of the same data can lead to overuse.

Concurrent access to variables is done by associating regions of code with parts of programs executions, which read shared values (readers) [12]. These sections are called read critical sections. Read critical sections interact with synchronization points, which are usually used as part of the update part of the program executions (updaters). Readers subscribe to a specific version of the data they are reading, which is the one available at the beginning of the critical section. The end of a critical section is explicit in the code, which is needed for the updaters to detect if there is no more reading activity for a specific piece of data. Read critical sections does not enforce ordering inside a single section, nor do multiple sections between each other.

3.2. RCU contexts in liburcu

Userspace-implemented RCU library librcu is a compile- and link-time solution for using RCU primitives in arbitrary C software without depending on kernel features of the operating system (OS) [12]. The library supports multiple implementations of the RCU semantics, the API consists of free functions with prefixes corresponding to the name of the technique (urcu) and the implementation technique (i.e. mb for memory barrier, qsrb for quiescent state-based reclamation or signal for using posix signals). By default, API calls are implemented as external linkage functions, and the generated IR code therefore contains explicit references to the mentioned free functions. Using optimizations which cause the functions to be inlined will render the solution described here unusable. Inlining small functions can be the result of link-time optimization or by defining the `URCU_INLINE_SMALL_FUNCTIONS` preprocessor symbol before including the library headers.

Another limitation is that debug information must be generated alongside with the IR code. An example of an API function which is used for opening a read-side critical section by using memory barriers as implementation is `urcu_mb_read_lock()`.

There are two API functionalities, which must be used in pairs. For registering threads, one would used the `urcu_<flavor>_register_thread()` and

`urcu_<flavor>_unregister_thread()`. These are used in a non-nested way (calling register while already registered is an error), but the other pair of API functions signifying the read-side critical sections can be nested indefinitely. These are the `urcu_<flavor>_read_lock()` and `urcu_<flavor>_read_unlock()` functions. The solution presented here is tailored towards the nestable usage, and can be extended to consider the non-nestable case. There are API calls which can only be safely used inside the context of registered threads (the majority of the librcu API) and there are API calls, which have special meaning when inside a read-side critical section (like `defer_rcu()` or `synchronize_rcu()`). The intended usage of the solution presented here is to provide information about the potential execution paths that are potentially enclosed in the mentioned API calls. It would help the software's discoverability, changeability, and maintainability to know which part of the code potentially contributes to the synchronization structure.

4. Code comprehension framework

Our earlied work proposed a code comprehension framework for the RCU synchronization contexts [7]. We have developed a static analysis solution based on the Clang compiler. Our static analysis tool takes advantage of the LLVM IR (Intermediate Representation) which is generated from the source code.

For context detection, the iterative algorithm of forward dataflow analyses uses reverse postorder traversal of the control-flow graph (CFG) elements in case of forward analysis in order for performance reasons. This results in a scalable method for gaining an overview about the synchronization aspects of the software. The modular nature of the approach lends itself to distributed use.

The transfer function saves the interesting locations (the instructions that can be used to get the locations), by appending them to the basic block level global fact, but only if this global fact is does not already contain them. In addition, if a context ending API call is detected, the exit state of the instruction set to the current global state of the basic block. The reverse postorder visitation guarantees, if a context starting instruction then happens to precede a context ending one, there is path in the CFG from the starter to the ending one. The set-like nature of the list in turn allows for the halting of the fixed-point algorithm in finite steps, as there are a finite amount of interesting locations inside a program.

The meet function is responsible for merging the exit states of multiple incoming dataflow facts. This is defined as the concatenation of the dataflow fact lists in a manner, that guarantees uniqueness of elements inside the resulting list, and the preservation of relative ordering among the interesting locations.

5. Visualization of the contexts

Monaco Editor is maintained by Microsoft and available worldwide for free [14]. It has a playground with full of interactive examples and provides wide access to

the editor and it supports feature like colorize the editor line-by-line, add different error and warning messages or add a hover message when the cursor is hovered over the text. Doing all this with JavaScript programming language for the dynamic parts, CSS for styling and HTML to build the raw frame [8]. It gives full access to the Document Object Model (DOM) supplemented by its own special elements. However, it sets up some limitations.

The figures below show the four aspects of Monaco Editor that we consider to be the most important. We would like to note that this is our implemented version of the code parser and the Monaco Editor. The C++ code is approximately the same in all four figures, with minimal changes in place, which were necessary in order to be able to present the different possible appearance methods.

```
    /*
     * Each thread need using RCU read-side need to be explicitly
     * registered.
     */
    urcu_memb_register_thread();

    /*
     * Adding nodes to the linked-list. Safe against concurrent
     * RCU traversals, require mutual exclusion with list updates.
     */
    for (i = 0; i < CAA_ARRAY_SIZE(values); i++) {
        struct mynode *node;

        node = malloc(sizeof(*node));
        if (!node) {
            ret = -1;
            goto end;
        }
        node->value = values[i];
        cds_list_add_tail_rcu(&node->node, &mylist);
    }

    /*
     * Surround the RCU read-side critical section with urcu_memb_read_lock()
     * and urcu_memb_read_unlock().
     */
    urcu_memb_read_lock();

    /*
     * This traversal can be performed concurrently with RCU updates.
     */
    cds_list_for_each_rcu(pos, &mylist) {
        struct mynode *node = cds_list_entry(pos, struct mynode, node);
        printf(" %d", node->value);
    }

    urcu_memb_read_unlock();
end:
    urcu_memb_unregister_thread();
```

Figure 1. Visualization of an RCU thread registration.

In order to make it easier to distinguish different visualization parts, we used

separate colors to display the individual methods and code parts. Figure 1 shows a thread registration process. The editor highlights precisely the part of the code where an `urcu_memb_register_thread()` registration takes place, the end of which is indicated by `urcu_memb_unregister_thread()`. The editor highlights this part of the code sections in yellowish color.

```
    /*
     * Each thread need using RCU read-side need to be explicitly
     * registered.
     */
    urcu_memb_register_thread();

    /*
     * Adding nodes to the linked-list. Safe against concurrent
     * RCU traversals, require mutual exclusion with list updates.
     */
    for (i = 0; i < CAA_ARRAY_SIZE(values); i++) {
      struct mynode *node;

      node = malloc(sizeof(*node));
      if (!node) {
        ret = -1;
        goto end;
      }
      node->value = values[i];
      cds_list_add_tail_rcu(&node->node, &mylist);
    }

    /*
     * Surround the RCU read-side critical section with urcu_memb_read_lock()
     * and urcu_memb_read_unlock().
     */
    urcu_memb_read_lock();

    /*
     * This traversal can be performed concurrently with RCU updates.
     */
    cds_list_for_each_rcu(pos, &mylist) {
      struct mynode *node = cds_list_entry(pos, struct mynode, node);
      printf(" %d", node->value);
    }

    urcu_memb_read_unlock();
end:
    urcu_memb_unregister_thread();
```

Figure 2. Visualization of an RCU lock snippet.

In Figure 2, we highlight another part of the previous code snippet where a read lock was created. Its registration starts at the `urcu_memb_read_lock()` line and ends with the `urcu_memb_read_unlock()` line. It is important to note that we can set the highlighting of these blocks ourselves, which should be in focus, as shown in Figure 1 and Figure 2 separately. We also have the option to display them at the same time. In this case, the different layers in the editor will be aligned.

The algorithm detects deficiencies that can cause problems at the code level,

such as the unregistration of registered threads or locks. The editor also draws the user's attention to such cases, as shown in the Figure 3. It also reveals which part of the code is missing, and if there are several errors in the code, it shows how many errors are in the editor in total. We can use the up and down arrows on the right side of the error bar to jump back and forth between errors. With this feature, real-time errors can be displayed to users, thereby avoiding the occurrence of runtime problems.

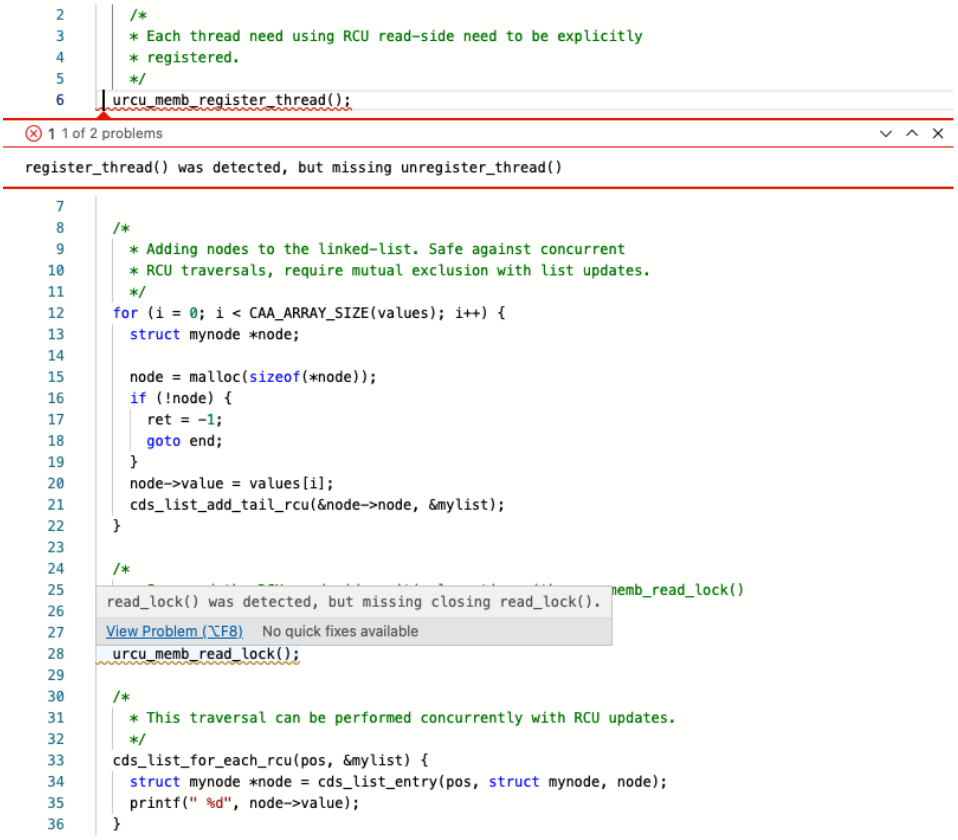

Figure 3. Visualization of an alert.

In Figure 4, one can see the highlighting of a code fragment that uses a RCU function that uses a shared variable outside the locking code snippet at runtime, potentially causing an error that could arise due to shared memory. By highlighting this, the user can better check whether the given piece of code has been provided with the appropriate error handling or threading methods, which can be used to avoid runtime problems due to shared memory space.

Figure 5 presents the comprehensive visualization of an RCU-based code snippet in the Monaco Editor. This approach makes many aspects of the RCU usage more

comprehensible. Our solution makes the debugging procedure, and bug fixes easier.

In addition to these, the Monaco Editor visualization implementation we created is able to highlight potential runtime problems, such as over-indexing on the array or highlighting different ranges and displaying hover messages.

```
    /*
     * Each thread need using RCU read-side need to be explicitly
     * registered.
     */
  urcu_memb_register_thread();

  /*
    * Adding nodes to the linked-list. Safe against concurrent
    * RCU traversals, require mutual exclusion with list updates.
    */
  for (i = 0; i < CAA_ARRAY_SIZE(values); i++) {
    struct mynode *node;

    node = malloc(sizeof(*node));
    if (!node) {
      ret = -1;
      goto end;
    }
    node->value = values[i];
    cds_list_add_tail_rcu(&node->node, &mylist);
  }

  /*
    * Surround the RCU read-side critical section with urcu_memb_read_lock()
    * and urcu_memb_read_unlock().
    */
  urcu_memb_read_lock();

  /*
    * This traversal can be performed concurrently with RCU updates.
    */
  cds_list_for_each_rcu(pos, &mylist) {
    struct mynode *node = cds_list_entry(pos, struct mynode, node);
    printf(" %d", node->value);
  }

  urcu_memb_read_unlock();
end:
  urcu_memb_unregister_thread();
```

Figure 4. Visualization of shared data's usage outside the locking snippet.

6. Conclusion

Despite RCU is a very powerful mechanism and in a sense simplifies thread handling in order for someone to understand what is going on in the background, a deeper understanding of the topic is required. The visualization tool does not answer

```
    /*
     * Each thread need using RCU read-side need to be explicitly
     * registered.
     */
  urcu_memb_register_thread();

  /*
    * Adding nodes to the linked-list. Safe against concurrent
    * RCU traversals, require mutual exclusion with list updates.
    */
  for (i = 0; i < CAA_ARRAY_SIZE(values); i++) {
    struct mynode *node;

    node = malloc(sizeof(*node));
    if (!node) {
      ret = -1;
      goto end;
    }
    node->value = values[i];
    cds_list_add_tail_rcu(&node->node, &mylist);
  }

  /*
    * Surround the RCU read-side critical section with urcu_memb_read_lock()
    * and urcu_memb_read_unlock().
    */
  urcu_memb_read_lock();

  /*
    * This traversal can be performed concurrently with RCU updates.
    */
  cds_list_for_each_rcu(pos, &mylist) {
    struct mynode *node = cds_list_entry(pos, struct mynode, node);
    printf(" %d", node->value);
  }

  urcu_memb_read_unlock();
end:
  urcu_memb_unregister_thread();
```

Figure 5. Comprehensive visualization in the Monaco Editor.

all questions, but it helps to comprehend the background processes better. Our previous work includes a code comprehension framework for RCU. This paper presents the visualization approach based on the framework. The visualization is implemented in the Microsoft's Monaco Editor that is a modern, customizable solution for high-level code comprehension.

References

[1] R. Bell, A. D. Malony, S. Shende: *ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis*, in: Euro-Par 2003 Parallel Processing, ed. by H. Kosch, L. Böszörményi, H. Hellwagner, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 17–26, isbn: 978-3-540-45209-6.

[2] M. Danisovszky, T. Nagy, K. Répás, G. Kusper: *Western Canon of Software Engineering: The Abstract Principles*, in: 2019 10th IEEE International Conference on Cognitive Infocommunications (CogInfoCom), 2019, pp. 153–156, doi: https://doi.org/10.1109/CogInfoCom47531.2019.9089999.

[3] M. Desnoyers, P. E. McKenney: *Userspace RCU*, https://liburcu.org/.

[4] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, J. Walpole: *User-Level Implementations of Read-Copy Update*, IEEE Transactions on Parallel and Distributed Systems 23.2 (2012), pp. 375–382, doi: https://doi.org/10.1109/TPDS.2011.159.

[5] M. Drocco, V. G. Castellana, M. Minutoli: *Practical Distributed Programming in C++*, in: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '20, Stockholm, Sweden: Association for Computing Machinery, 2020, pp. 35–39, isbn: 9781450370523, doi: https://doi.org/10.1145/3369583.3392680.

[6] E. Fülöp, A. Gyén, N. Pataki: *A Framework for C++ Exception Handling Assistance*, in: Proceedings of the Ninth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, ed. by Z. Budimac, CEUR Workshop Proceedings 3237, 2022, 4:1–4:13, url: http://ceur-ws.org/Vol-3237/paper-ful.pdf.

[7] E. Fülöp, A. Gyén, N. Pataki: *Code Comprehension for Read-Copy-Update Synchronization Contexts in C Code*, in: Geoinformatics and Data Analysis, ed. by S. Bourennane, P. Kubicek, Cham: Springer International Publishing, 2022, pp. 187–200, isbn: 978-3-031-08017-3, doi: https://doi.org/10.1007/978-3-031-08017-3_17.

[8] E. Fülöp, A. Gyén, N. Pataki: *Monaco Support for an Improved Exception Specification in C++*, IPSI Transactions on Internet Research 19.1 (Jan. 2023), pp. 24–31, doi: https://doi.org/10.58245/ipsi.tir.2301.05, url: http://ipsitransactions.org/journals/papers/tir/2023jan/p5.pdf.

[9] T. E. Hart, P. E. McKenney, A. D. Brown, J. Walpole: *Performance of memory reclamation for lockless synchronization*, Journal of Parallel and Distributed Computing 67.12 (2007), Best Paper Awards: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), pp. 1270–1285, issn: 0743-7315, doi: https://doi.org/10.1016/j.jpdc.2007.04.010, url: https://www.sciencedirect.com/science/article/pii/S074373150700069X.

[10] B. Kot, B. Wuensche, J. Grundy, J. Hosking: *Information Visualisation Utilising 3D Computer Game Engines Case Study: A Source Code Comprehension Tool*, in: Proceedings of the 6th ACM SIGCHI New Zealand Chapter's International Conference on Computer-Human Interaction: Making CHI Natural, CHINZ '05, Auckland, New Zealand: Association for Computing Machinery, 2005, pp. 53–60, isbn: 1595930361, doi: https://doi.org/10.1145/1073943.1073954.

[11] G. Márton, I. Szekeres, Z. Porkoláb: *Towards a High-level C++ Abstraction To Utilize The Read-Copy-Update Pattern*, Acta Electrotechnica et Informatica 18.3 (2018), pp. 18–26, doi: https://doi.org/0.15546/aeei-2018-0021.

[12] P. E. McKenney: *Is Parallel Programming Hard, And, If So, What Can You Do About It? (Release v2021.12.22a)*, 2021, arXiv: 1701.00854 [cs.DC], url: https://arxiv.org/abs/1701.00854.

[13] P. E. McKenney, J. Walpole: *What is RCU, fundamentally?*, 2007, url: https://lwn.net/Articles/262464/.

[14] Microsoft: *Monaco Editor*, https://microsoft.github.io/monaco-editor/.

[15] G. Nagy, Z. Porkoláb: *Read-Copy-Update as a Possible Locking Strategy in Scala*, in: Proceedings of the Seventh Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, ed. by Z. Budimac, CEUR Workshop Proceedings 2217, 2018, 12:1–12:8, url: http://ceur-ws.org/Vol-2217/paper-nag.pdf.

[16] Z. Porkoláb, T. Brunner: *Advanced Code Comprehension using Version Control Information*, IPSI Transactions on Internet Research 16.2 (July 2020), pp. 47–54.

[17] Z. Porkoláb, T. Brunner: *The CodeCompass Comprehension Framework*, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 393–396, isbn: 9781450357142, doi: https://doi.org/10.1145/3196321.3196352.

[18] W. Steingartner, M. Haratim, J. Dostál: *Software visualization of natural semantics of imperative languages - a teaching tool*, in: 2019 IEEE 15th International Scientific Conference on Informatics, 2019, pp. 000509–000514, doi: https://doi.org/10.1109/Informatics47936.2019.9119290.