Document number: P0232R0
Date: 2016-02-12
Project: Programming Language C++, SG14, SG1, EWG
Authors: Paul McKenney, Michael Wong, Maged Michael
Reply to: paulmckrcu@gmail.com, fraggamuffin@gmail.com, maged.michael@acm.org

A Concurrency ToolKit for Structured Deferral or Optimistic Speculation

1. Introduction

This papers introduce the concept of a Concurrency ToolKit that contains the recently added atomic smart pointer [Sut15] along with the proposed lock-free algorithms on Hazard Pointers [Mic16] and Read-Copy-Update [McK15] and analyzes their motivation in a humorous way, while showing where they can be useful, and their performance differences.

It is also an excerpt from *"Is Parallel Programming Hard, And, If So, What Can You Do About It?" [Perf]* along with analysis of *"Reference Counting through Atomic Smart Pointers"* [Sut15] to show the advantages and disadvantages of each technique and to offer motivation supporting why we would want each of these techniques to be standardized in C++.

2. Schrödinger's Zoo

We will use the Schrödinger's Zoo application to evaluate performance [McK13]. Schrödinger has a zoo containing a large number of animals, and he would like to track

them using an in-memory database with each animal in the zoo represented by a data item in this database. Each animal has a unique name that is used as a key, with a variety of data tracked for each animal.

Births, captures, and purchases result in insertions, while deaths, releases, and sales result in deletions. Because Schrödinger's zoo contains a large quantity of short-lived animals, including mice and insects, the database must be able to support a high update rate.

Those interested in Schrödinger's animals can query them, however, Schrödinger has noted extremely high rates of queries for his cat, so much so that he suspects that his mice might be using the database to check up on their nemesis. This means that Schrödinger's application must be able to support a high rate of queries to a single data element.

On the other hand, it is impossible to say exactly when a given birth or death occurs. For example, a cat's heart rate will normally be between 140 and 220 beats per minute [Wiki16], so that it is necessary to wait many seconds after the last heartbeat before death can be pronounced. In the meantime, different observers might disagree as to whether the cat is living or dead. Schrödinger's own work [Sch35] and that of his colleague Heisenberg [Hei27] has helped Schrödinger to cope with this sort of ambiguity and uncertainty. In fact, Schrödinger's design goes further by actually exploiting ambiguity and uncertainty. He does this by refusing to require that queries, insertions, and deletions be fully ordered. As we will see, this design choice will enable extremely high-speed queries.

Nor is this situation limited to Schrödinger's zoo's database. Any situation in which data within the computer is a function of events and entities outside the computer, similar uncertainties will be forced by speed-of-light delays if by nothing else. By the time a given change has been committed to the system's memory, it might well have been superseded by some other change. It might not even be possible to determine the time order of several external events. Worse yet, there are situations where protocols introduce additional delays, for example, such delays are common practice in Internet routing protocols. These routing-protocol delays are absolutely necessary in order to preserve Internet stability, however, they introduce long periods during which a given system connected to Internet might be uncertain of Internet's topology. The applicability of unordered lookups is thus quite broad.

3. Trading Certainty for Performance and Scalability

Clinging to certainty is all too human, but it can also be all too bad for your software's performance and scalability. For example, the following pattern is quite common:

1. Acquire a lock
2. While holding the lock, compute some property of data protected by that lock
3. Release the lock
4. Use the computed property

In step 4, it is uncertain whether or not the computed property still holds. As soon as the lock was dropped in step 3, some other thread might have acquired that lock and carried out arbitrarily large changes to the protected data.

If the computation of the property did not update the data structure, it might be possible to substitute lightweight synchronization for the lock acquisition and release in steps 1 and 3. Two examples of such lightweight synchronization are hazard pointers (see P0233R0 [Mic16]) and RCU (see P0279R0 [McK16]). Both of these synchronization mechanisms defer the destructive portion of each update (for example, the memory-reclamation portion) in order to allow lightweight readers. Such readers avoid atomic read-modify-write operations entirely and avoid cache misses in the common case.

4. Performance Comparison

Schrödinger has wisely selected a few benchmarks for the database tracking his zoo, which run on a four-socket 32-core (64 hardware threads) Westmere-EX x86 system. The first benchmark is read-only, testing high query loads. The results for a 1024-bucket hash table are shown in the next figure. This benchmark in Figure 1 shows that global locking is a spectacularly bad choice (which should be no surprise), but that per-bucket locking is also disastrous above 8 CPUs (which might be more of a surprise). The problem with per-bucket locking is that there is no read-side locality, which results in cache thrashing on the lock data structure. This cache thrashing become quite expensive across socket boundaries, hence the cliff beyond 8 CPUs (note the log-log nature of the plot) in Figure 1. In contrast, both hazard pointers and RCU do quite well, both offering not only near-linear scalability, but near-optimal performance.

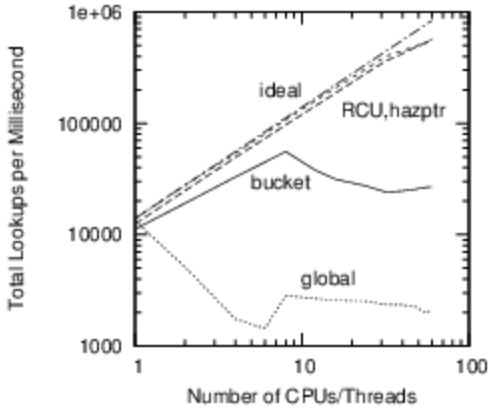

Figure 1. This shows RCU and hazard Pointer compared to global locking which is a spectacularly bad choice (which should be no surprise), but that per-bucket locking is also disastrous above 8 CPUs, when compared to ideal

These results are quite encouraging for RCU and for hazard pointers, but Schrödinger is well aware of the extreme levels of interest in the well-being of his cat, so much so that he suspects that his rats are using his database to track the whereabouts of their potential predator. This calls for a benchmark in Figure 2 testing queries weighted heavily to the cat, for example, running queries on 60 CPUs and varying the number of CPUs that are repeatedly looking up only the cat. The results are shown on the next plot. As can be seen, bucket-locking's performance converges on that of global locking as the number of cat-querying CPUs increases. The performance of global locking increases with increasing cat intensity, and this is due to cache locality. Please note that even with this increase, the performance of global locking remains abysmal at more than two orders of magnitude slower than that of either hazard pointers or RCU.

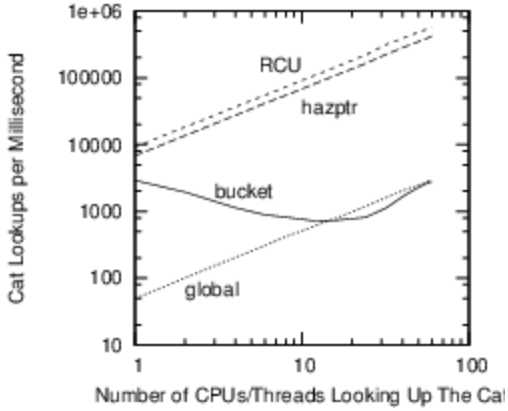

Figure 2: benchmark testing queries weighted heavily to the cat

Finally, Schrödinger must look at updates, running a benchmark that varies the number of CPUs doing updates out of a total of 60 CPUs. This results in the two plots shown below in Figure 3 and 4, with the left-hand plot showing number of lookups (which of course drops to zero when all 60 CPUs are doing updates) and with the right-hand plot showing number of updates. This splits into three regimes: RCU does better at update rate, hazard pointers does better at moderate update rates, and bucket locking does better at very high update rates. This last should not be surprising: Both RCU and hazard pointers incur additional reclamation overhead on updates, and at high update rates there are not enough readers to offset this added cost.

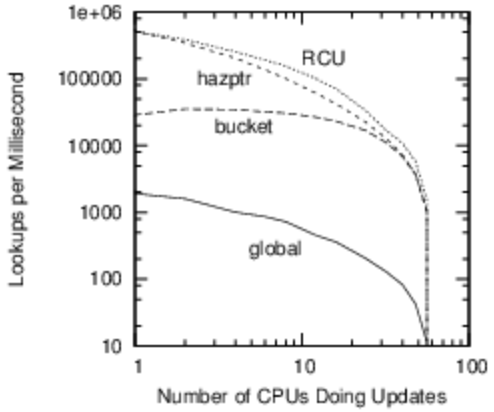

Figure 3: Number of lookups/ms vs CPUs doing updates

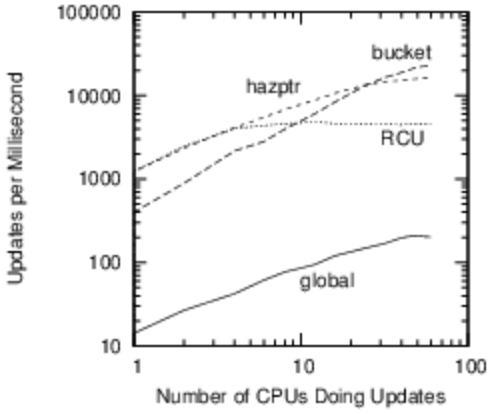

Figure 4: Number of Updates/ms vs CPUs doing updates

5. Which to Choose?

Bucket locking is very clearly only useful at very high update rates. As a rough rule of thumb, from a performance and scalability viewpoint, you should use RCU for read-intensive workloads and hazard pointers for workloads that have significant update rates (more than 10% updates based on the above benchmarks). However, there are other considerations, and these are summarized in the following table taken from Hazard Pointer P0233R0 [Mic16] comparing Reference Counting (as proposed in Smart Pointer and Atomic Smart Pointer), RCU and Hazard Pointers.

	Reference Counting	RCU	Hazard Pointers
Unreclaimed objects	**Bounded**	Unbounded	**Bounded**
Non-blocking traversal	**Lock-free**	**Wait-free**	**Lock-free.**
Non-blocking reclamation	**Lock-free**	Blocking	**Lock-free**
Contention among readers	Can be very high	**No contention**	**No contention**
Traversal speed	Atomic updates	**No or low overhead**	Store-load fence
Reference acquisition	Conditional	**Unconditional**	Conditional
Automatic reclamation	**Yes**	No	No

Advantages

This leaves open the question of reference counting, such as that used by smart pointers. This question can be answered by considering the cache-thrashing effects of per-access reference-count manipulations [McK13, Figure 2]. Our experience leads us to believe that a successful optimization of reference counting would result in something like either hazard pointers or RCU, depending on the design choices. You are of course welcome to try to prove us wrong!

6. When to Procrastinate?

Although large read-side performance benefits can be obtained from both hazard pointers and RCU, both are specialized mechanisms that are almost always used in conjunction with other mechanisms. This raises the question as to when hazard pointers and RCU should be used. Extensive use of RCU in the Linux kernel has resulted in the following rules of thumb, which may also apply to hazard pointers:

1. Procrastination works extremely well in read-mostly situations where disagreement among readers is permissible. What constitutes "read-mostly"

depends on the workload, but 90 percent reads to 10 percent updates is a good starting point.
2. Procrastination works reasonably well in read-mostly situations where readers must agree.
3. Procrastination sometimes works well in cases where the numbers of reads and updates are roughly equal and where readers must agree.
4. Procrastination rarely works well in update-mostly situations where readers must agree. There are currently two known exceptions to this rule: (1) providing existence guarantees for update-friendly mechanisms, and (2) providing low-overhead wait-free read-side access for realtime use.

Note that the traditional definition of *read* may be generalized to include writes. An example within Schrödinger's application would be if each animal's data structure included an array of per-thread cache-aligned variables that count queries involving that animal, thus measuring Schrödinger's cat's popularity. This works because these read-side writes do not conflict. Further generalization is not only possible, but also heavily used in practice—for example, some Linux-kernel RCU read-side critical sections contain conflicting writes that are protected by locking, with the lock contained in the RCU-protected data structure [Arc03]. These use cases work because reference counters, hazard pointers, and RCU all permit a wide variety of code in their read-side critical sections, including atomic operations, memory barriers, time delays, transactions, and lock operations.

Schrödinger's application requires only the traditional definition of *read* and is thus covered by the first rule of thumb. This application is therefore eminently suitable for synchronization via procrastination. These rules will continue to be refined with further experience. In particular, better mechanisms are needed for update-heavy situations, and there is some promising work in progress in this area, such as Transactional Memory. Finally, the above table along with Table 1 in [McK13] might be a first step toward rules of thumb for choosing between Reference Counting, hazard pointers and RCU.

6.1 Procrastination Implementations

The following is a partial list of RCU and hazard-pointer implementations:

- Userspace RCU (liburcu) - C implementation of RCU and several RCU-protected data structures, including linked lists and hash tables. Available in a number of Linux distributions.

- Concurrent Building Blocks - C++ implementation of Hazard Pointer (called "SMR") and other lock-free data structures. Also has Java interfaces.
- Concurrency Kit - C implementation of Hazard Pointer and lock-free data structures Atomic Ptr Plus - C/C++ library that has a Hazard Pointer implementation.
- The parallelism shift and C++'s memory model - Contains C++ implementation for Windows in appendices.
- libcds - C++ library of lock-free containers and Hazard Pointer implementation
- Predicate RCU: An RCU for Scalable Concurrent Updates - Contains an RCU implementation optimized for moderate-sized systems. Includes logic allowing updates to ignore irrelevant RCU read-side critical sections.
- Concurrent Memory Deallocation in the Objective-C Runtime - Describes an RCU-like implementation that uses program-counter ranges to describe RCU read-side critical sections.
- Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks - Includes an RCU-like implementation that is specialized for reader-writer locking.
- Speedy Transactions in Multicore In-memory Databases - In-memory database that uses an RCU-like mechanism.
- Mindicators: A Scalable Approach to Quiescence - RCU-like mechanism specialized for use within transactional memory implementations.
- Poor Man's RCU - Trivial RCU-like implementation that provides lock-free readers despite being implemented using locking.

7. Conclusion and goals

This paper was written with the mind of helping the audience understand the tradeoffs between the RCU proposal and Hazard Pointer proposal, how they compare with the current interfaces in Shared Pointer and Atomic Shared Pointer and when is it better to be lazy and procrastinate.

All three form interesting tools in a Concurrency ToolKit for Lock-free programming that is emerging in the C++ Standard. The result will make C++ more applicable in applications with sensitivity towards real-time constraints and low-latency such as Games, Finance, Banking, and image processing applications in VR/AR, automobile vision, or other simulation environment in embedded processing.

8. References

[Arc03] Arcangeli, A., Cao, M., McKenney, P. E., Sarma, D. 2003. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 Usenix Annual Technical Conference*: 297-310.

[Hei27] W. Heisenberg. Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. Zeitschrift für Physik, 43(3-4):172–198, 1927. English translation in “Quantum theory and measurement” by Wheeler and Zurek.

[McK13] Paul E. McKenney. Structured deferral: synchronization via procrastination. Commun. ACM, 56(7):40–49, July 2013.

[McK15] N4483, P. McKenney, Read-Copy-Update, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4483.pdf

[McK16] P0279R0, P. McKenney, RCU

[Mic16] P0233R0, M. Michael et al, Hazard Pointers, Safe Resource Reclamation for Optimistic Concurrency

[Perf] P. McKenney, *“Is Parallel Programming Hard, And, If So, What Can You Do About It?”* , http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

[Sch35] E. Schrödinger. Die gegenwärtige Situation in der Quantenmechanik. Naturwissenschaften, 23:807–812; 823–828; 844–949, November 1935. English translation: http://www.tuhh.de/rzt/rzt/it/QM/cat.html.

[Sut15] H. Sutter, Atomic Smart Pointers, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4162.pdf

[Wiki16] Wikipedia. Cat anatomy. https://en.wikipedia.org/wiki/Cat_anatomy.