Brief Announcement: Crystalline: Fast and Memory Efficient Wait-Free Reclamation

Ruslan Nikolaev
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA

Binoy Ravindran

Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA

Abstract

We present a new wait-free memory reclamation scheme, Crystalline, that simultaneously addresses the challenges of high performance, high memory efficiency, and wait-freedom. Crystalline guarantees complete wait-freedom even when threads are dynamically recycled, asynchronously reclaims memory in the sense that any thread can reclaim memory retired by any other thread, and ensures (an almost) balanced reclamation workload across all threads. The latter two properties result in Crystalline's high performance and high memory efficiency, a difficult trade-off for most existing schemes. Our evaluations show that Crystalline exhibits outstanding scalability and memory efficiency, and achieves superior throughput than state-of-the-art reclamation schemes as the number of threads grows.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases memory reclamation, wait-free, reference counting, hazard pointers

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.60

Related Version *Full Version*: https://arxiv.org/abs/2108.02763

Supplementary Material *Software (Source Code)*: https://github.com/rusnikola/wfsmr

Funding ONR grants: N00014-18-1-2022, N00014-19-1-2493, and AFOSR grant: FA9550-16-1-0371.

Acknowledgements We thank the anonymous reviewers for their invaluable feedback.

1 Introduction

Non-blocking data structures do not use simple mutual exclusion: a concurrent thread may hold an obsolete pointer to an object which is about to be freed by another thread. Responding to this challenge, *safe memory reclamation* (SMR) schemes for unmanaged C/C++ code have been proposed in the literature (e.g., [1, 2, 3, 4, 5, 6, 9, 11, 12, 13, 14, 16]). However, they typically involve major trade-offs, e.g., memory efficiency vs. throughput.

The significance of balancing the reclamation workload – the task of reclaiming deleted memory objects – across all threads have not received adequate attention in the literature. Consider the common scenario when read operations dominate, but data is still modified. If the reclamation workload is unbalanced, as in most existing reclamation schemes [4, 6, 12, 16], then most threads are not actively reclaiming memory, which can cause memory waste.

Vast majority of SMR schemes (e.g., [6, 12, 16]) are inherently *synchronous*, i.e., they need to periodically examine which objects marked for deletion can be safely freed. In contrast, reference counting [7, 15] is *asynchronous*: an *arbitrary* thread with the last reference frees an object. Unfortunately, reference counting is often impractical due to very high overheads when accessing objects. Hyaline [8, 11], an approach where reference counters are only used when objects are retired, is still *asynchronous* and exhibits high performance. However, Hyaline can be blocking since its memory usage is unbounded when threads starve.

© Ruslan Nikolaev and Binoy Ravindran;
licensed under Creative Commons License CC-BY 4.0
35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 60; pp. 60:1–60:4

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

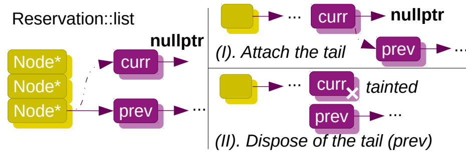


Figure 1 Crystalline-W: list tainting.

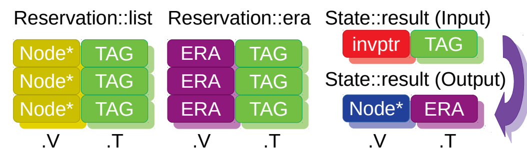


Figure 2 Crystalline-W: reservations and state.

We propose Crystalline-W, which has the following properties: wait-freedom, asynchronous reclamation, and balanced reclamation workload. Crystalline-W extends a lock-free algorithm, Crystalline-L, which we also present in the paper (Section 2). Crystalline-L, in turn, is based on Hyaline-1S [8, 11], a prior scheme. Making Crystalline wait-free involves overcoming a set of unique challenges, caused in part due to asynchronous memory reclamation (Section 3).

2 Lock-Free Reclamation with Crystalline-L

In Hyaline [8, 11], threads explicitly annotate each operation. When objects are detached from a data structure, they are first *retired* and then freed when it is safe to do so. Hyaline-1S is a variant that bounds memory usage for *stalled* threads by explicitly tracking local pointers via a special `protect` method using the global era clock [12, 16]. However, Hyaline-1S is still not lock-free unless operations are periodically restarted for *starving* threads. The problem arises when a thread reserves an increasing number of local pointers in an unbounded loop (e.g., one "unlucky" thread is stuck traversing a list because it keeps growing).

The culprit is a simplistic API [16] which enables retrieving an unbounded number of local pointers. To avoid this issue, alternative APIs [6, 12] explicitly differentiate each local pointer reservation in `protect`. Crystalline-L modifies Hyaline-1S so that each thread has several (rather than just one) reservations. Each reservation has its own list of retired objects.

Hyaline-1S retires an entire *batch* of objects, and each batch is attached to every active reservation. Each object reserves space for a list of retired objects. Whereas Hyaline-1S only needs MAX_THREADS+1 objects to successfully retire a batch, Crystalline-L handles MAX_IDX local pointers and consequently needs MAX_THREADS×MAX_IDX+1 objects.

One problem with Hyaline-1S is that a batch must aggregate *all* objects before retirement can even start, which is further aggravated in Crystalline-L. In practice, the required number of objects is much lower as each object is appended to the respective list only if the list's era overlaps with the batch's minimum birth era. Crystalline-L uses *dynamic* batches to avoid their excessive growth. The `retire` method first checks how many lists are to be changed for the batch to be fully retired and records the location of the corresponding reservations. If the number of objects in the batch suffices, `retire` completes by appending the objects to their corresponding lists. Otherwise, `retire` is repeated later when more objects are available.

3 Wait-Free Reclamation with Crystalline-W

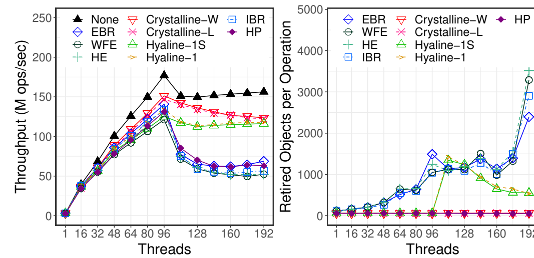
The Crystalline-L scheme is still only lock-free because of two fundamental challenges:

- `retire` has an unbounded loop: `protect` or another `retire` contends on the same list.
- `protect` has an unbounded loop which must converge on the era value; however, the era clock unconditionally increments when a new object is allocated.

To solve the first problem, we use unconditional SWAP in `retire` as shown in Figure 1. We initialize the *next* field of a retired object with `nullptr`, and swap the current list head with the pointer to the retired object. Putting corner cases aside, two major cases exist. If

None	no reclamation (leak memory)
Hyaline-1	basic variant [8, 11]
Hyaline-1S	supports stalled threads [8, 11]
HP	the hazard pointers scheme [6]
HE	the hazard eras scheme [12]
IBR	2GEIBR (interval-based) [16]
WFE	the wait-free eras scheme [9]
EBR	epoch-based reclamation

Figure 3 Evaluated reclamation schemes.



(a) Throughput. **(b)** Retired objs.

Figure 4 Lock-free hash map (read-dominated).

the retired object still has its next field intact, it simply attaches the previous list as its tail (part I). It is also possible that a thread associated with the reservation already started traversing and dereferencing retired objects. Crystalline-W additionally taints the retrieved *next* pointer of all traversed objects by using SWAP. `retire` finds that the next field is now tainted (part II). Thus, `retire` traverses the tail on behalf of the thread that tainted *next*.

The second problem was already considered by WFE [9] for hazard eras [12]. The approach, which we also adopt in Crystalline-W, is based on a fast-path-slow-path idea to coordinate global era clock increments. Each thread maintains its *state* for the slow path, which is taken when `protect` fails to retrieve a pointer after a small number of steps. The *result* field of *state* is used for both input and output. On input, a current slow path cycle is advertised. Output contains a retrieved pointer with the corresponding era. In Figure 2, we demonstrate how input and output values must be aligned. Reservations need to be extended with extra tags which identify slow path cycles and prevent spurious updates. Despite similarities, Crystalline-W substantially diverges from the original WFE's idea due to its unique retirement mechanism. Complete details can be found in [10].

4 Evaluation

Crystalline's implementation extends the benchmark of [9, 11, 16]. We evaluated all schemes (Figure 3) from 1 to 192 threads on a 96-core machine consisting of four Intel Xeon E7-8890 v4 2.20 GHz CPUs, 256GB of RAM. We present results for hash map [6] under (90% get, 10% put) workload. Complete details of our experiments and additional results are in [10].

Crystalline-L/-W achieve superior throughput (Figure 4a), which is especially evident under oversubscription, where the gap with other algorithms is as large as 2x. Hyaline-1/-1S's throughput is worse, which is due to a smaller granularity of reservations. WFE has the worst throughput. Crystalline-L/-W achieve exceptional memory efficiency which is on par with HP (Figure 4b). Even Hyaline-1/1S are visibly less memory efficient.

5 Conclusions

We presented a new wait-free SMR scheme, Crystalline-W, which guarantees complete wait-freedom with bounded memory usage. Crystalline-W is based on a lock-free algorithm, Crystalline-L, which is also presented in the paper. Crystalline-L is an improved version of Hyaline-1S that additionally guarantees bounded memory usage even when threads starve. Both Crystalline-L/-W show very high throughput and high memory efficiency, unparalleled among existing schemes, which is especially evident in read-dominated workloads.

References

1. Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. Concurrent Deferred Reference Counting with Constant-Time Overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21, pages 526–541. ACM, 2021. doi:10.1145/3453483.3454060.
2. Nachshon Cohen. Every Data Structure Deserves Lock-free Memory Reclamation. *Proc. ACM Program. Lang.*, 2(OOPSLA):143:1–143:24, 2018. doi:10.1145/3276513.
3. Andreia Correia, Pedro Ramalhete, and Pascal Felber. OrcGC: Automatic Lock-Free Memory Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 205–218. ACM, 2021. doi:10.1145/3437801.3441596.
4. Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.
5. Jeehoon Kang and Jaehwang Jung. A Marriage of Pointer- and Epoch-Based Reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '20, pages 314–328. ACM, 2020. doi:10.1145/3385412.3385978.
6. Maged M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004. doi:10.1109/TPDS.2004.8.
7. Maged M. Michael and Michael L. Scott. Correction of a Memory Management Method for Lock-Free Data Structures. Technical report, University of Rochester, USA, 1995. URL: https://www.cs.rochester.edu/u/scott/papers/1995_TR599.pdf.
8. Ruslan Nikolaev and Binoy Ravindran. Brief Announcement: Hyaline: Fast and Transparent Lock-Free Memory Reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 419–421. ACM, 2019. doi:10.1145/3293611.3331575.
9. Ruslan Nikolaev and Binoy Ravindran. Universal Wait-Free Memory Reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 130–143. ACM, 2020. doi:10.1145/3332466.3374540.
10. Ruslan Nikolaev and Binoy Ravindran. Crystalline: Fast and Memory Efficient Wait-Free Reclamation (full paper, arXiv), 2021. arXiv:2108.02763.
11. Ruslan Nikolaev and Binoy Ravindran. Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21, pages 987–1002. ACM, 2021. doi:10.1145/3453483.3454090.
12. Pedro Ramalhete and Andreia Correia. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 367–369. ACM, 2017. doi:10.1145/3087556.3087588.
13. Gali Sheffi, Maurice Herlihy, and Erez Petrank. VBR: Version Based Reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, pages 443–445. ACM, 2021. doi:10.1145/3409964.3461817.
14. Ajay Singh, Trevor Brown, and Ali Mashtizadeh. NBR: Neutralization Based Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 175–190. ACM, 2021. doi:10.1145/3437801.3441625.
15. John D. Valois. Lock-free Linked Lists Using Compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222. ACM, 1995. doi:10.1145/224964.224988.
16. Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based Memory Reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 1–13. ACM, 2018. doi:10.1145/3178487.3178488.