Smarter Atomic Smart Pointers
Safe and Efficient Concurrent Memory Management (Abstract)*

Daniel Anderson
Carnegie Mellon University
Pittsburgh, USA
dlanders@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, USA
guyb@cs.cmu.edu

Yuanhao Wei
Carnegie Mellon University
Pittsburgh, USA
yuanhao1@cs.cmu.edu

ABSTRACT

We present a technique for concurrent memory management that combines the ease-of-use of automatic memory reclamation, and the efficiency of state-of-the-art deferred reclamation algorithms.

First, we combine ideas from referencing counting and hazard pointers in a novel way to implement automatic concurrent reference counting with wait-free, constant-time overhead. Second, we generalize our previous algorithm to obtain a method for converting *any* standard manual SMR technique into an automatic reference counting technique with a similar performance profile.

We have implemented the approach as a C++ library and compared it experimentally to existing atomic reference-counting libraries and state-of-the-art manual techniques. Our results indicate that our technique is faster than existing reference-counting implementations, and competitive with manual memory reclamation techniques. More importantly, it is significantly safer than manual techniques since objects are reclaimed automatically.

ACM Reference Format:
Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2023. Smarter Atomic Smart Pointers Safe and Efficient Concurrent Memory Management (Abstract). In *Proceedings of the 2023 ACM Workshop on Highlights of Parallel Computing (HOPC '23), June 16, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3597635.3598027

1 INTRODUCTION

Memory reclamation, the problem of freeing allocated memory in a safe manner, is essential in any program that uses dynamic memory allocation. A block of memory is safe to reclaim only when it can not be subsequently accessed by any thread of the program. Determining exactly when this is the case is, however, a difficult problem, and even more so for mutlithreaded programs which could be sharing, copying, or modifying references to the same memory blocks concurrently. Traditional solutions for safe memory management are filled with trade-offs. One choice is to use a language with automatic garbage collection, but this can inhibit performance and make it more difficult for programmers to implement certain data structures.

Researchers have developed a broad set of techniques to implement safe concurrent memory management. The goal is to delay the destruction and reclamation on an object until it can be ensured that no thread can still access it. These techniques are generally referred to as *safe memory reclamation* (SMR), and include approaches such as read-copy-update (RCU) [9], epoch-based-reclamation (EBR) [7], hazard-pointers (HP) [11], pass-the-buck [10], pass-the-pointer [4], interval-based reclamation (IBR) [16], Hyaline [13].These techniques, however, are difficult to use and can lead to subtle and hard to reproduce bugs. As evidence, we [1] noted several instances where these techniques were used incorrectly.

An alternative approach for memory management in languages without built-in garbage collection (or even with) is to use reference counting. Reference counting requires very few modifications for programmers to integrate into their code, and provides memory safety and leak freedom automatically as long as the programmer does not create reference cycles. Owing to the ease of use of automatic reference counting, there has been increasing interest in concurrent (atomic) reference-counted pointers (both strong and weak), as evidenced by their inclusion in the most recent C++ standard (C++20), and recent papers on the topic [4, 15]. Early approaches [5, 10] suffered severe performance issues due to contention on the reference counts, but more recent approaches, such as FRC [15] and OrcGC [4] have developed scalable approaches for concurrent reference counting.

2 OUR FIRST CONTRIBUTION

In our first paper [1], we propose a theoretically and practically efficient approach to automatic memory reclamation based on a novel combination of reference counting and HP. Theoretically, we show the first solution with constant expected time overhead using only single word compare-and-swap (CAS) and only delaying $O(P^2)$ decrements. Previous approaches are either only lock-free, wait-free with $O(P)$ time per operation, or use double-word fetch-and-add, which is not available on modern machines.

Our approach is based on a new algorithm that generalizes HP to allow for multiple retires on the same object. Standard HP [11] would not be efficient with multiple retires, requiring potentially much more space. Our generalization allows us to implement *deferred decrements* that protect an object's reference count, delaying decrements (and hence reclamation) while an increment is in progress. This contrasts with previous reference counting techniques [8, 10] that use HP to delay memory reclamation *after* a reference count hits zero. This is a subtle difference, but it has ramifications both in theory and practice. We further extend the approach by borrowing the idea of *deferred increments* from reference-counted garbage collectors [3]. When a reference to an object is

*This work appeared in PLDI'21 as "Concurrent Deferred Reference Counting with Constant-Time Overhead" [1] and in PLDI'22 as "Turning Manual Concurrent Memory Reclamation into Automatic Reference Counting" [2]

HOPC '23, June 16, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0218-1/23/06.
https://doi.org/10.1145/3597635.3598027

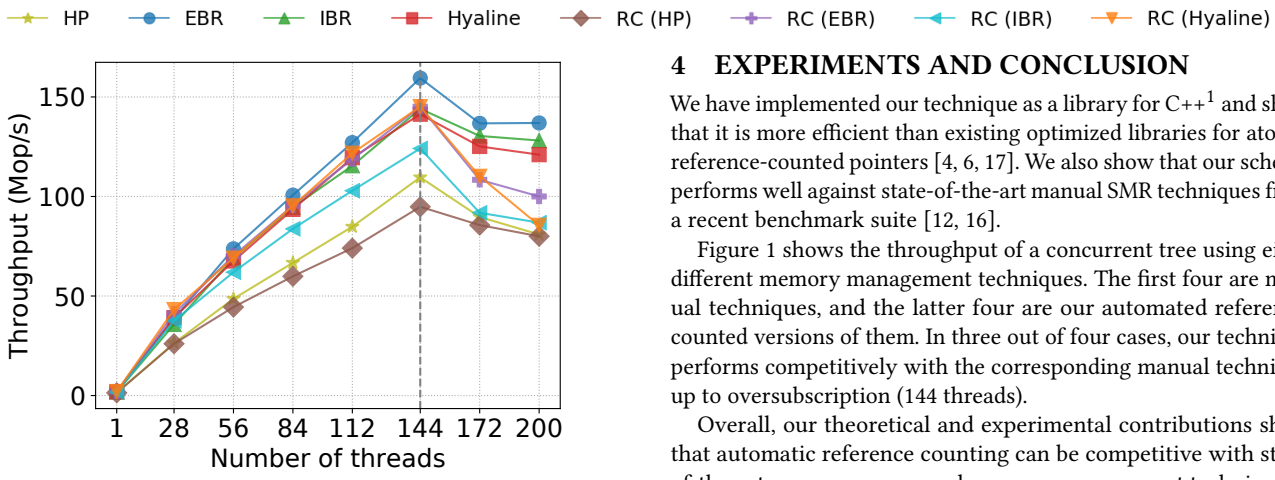

Figure 1: Throughput of a concurrent binary search tree with 100K keys, undergoing 90% reads and 10% updates.

short lived, it almost certainly does not need to modify the reference count. Our technique protects the reference count during the reference's short lifetime. In the common case, this avoids both the increment and the decrement.

3 OUR SECOND CONTRIBUTION

In our second paper [2], we show that reference counting can be nearly as fast as *any* manual technique while using a similar amount of memory (in most cases), thus showing that the ease-of-use of automatic approaches comes at no significant cost to practical performance. The technique in our first paper was based on HP and exhibits similar performance to manual application of HP. HP can be up to twice as slow as techniques such as EBR, though they also use substantially less memory.

In this work, we show that the technique can in fact be applied to any manual SMR scheme to yield an automatic version with similar performance. We apply this to three (very different) state-of-the-art manual techniques, EBR, IBR and Hyaline, to yield automatic versions of all three. To the best of our knowledge, this is the first time reference counting has been combined with any manual technique outside of variations of HP. The resulting algorithms are lock-free, assuming the SMR scheme being automated is lock-free.

In addition, we show how this framework can be extended even further to support lock-free atomic weak pointers. We use them to implement a concurrent doubly-linked-list based queue [14], and show that our implementation is several times faster than the only other lock-free atomic weak pointer that we are aware of [17].

A key challenge with weak pointers is supporting the upgrade to strong pointers efficiently. This requires being able to atomically increment the reference count only if it is not already zero. This operation is typically implemented using a CAS-loop which takes up to $O(P)$ amortized time per process if P processes perform this upgrade at the same time. Instead, we show how to implement a wait-free *increment-if-not-zero* operation so that reading and incrementing/decrementing take only $O(1)$ time in the worst case.

4 EXPERIMENTS AND CONCLUSION

We have implemented our technique as a library for C++[1] and show that it is more efficient than existing optimized libraries for atomic reference-counted pointers [4, 6, 17]. We also show that our scheme performs well against state-of-the-art manual SMR techniques from a recent benchmark suite [12, 16].

Figure 1 shows the throughput of a concurrent tree using eight different memory management techniques. The first four are manual techniques, and the latter four are our automated reference-counted versions of them. In three out of four cases, our technique performs competitively with the corresponding manual technique up to oversubscription (144 threads).

Overall, our theoretical and experimental contributions show that automatic reference counting can be competitive with state-of-the-art error-prone manual memory management techniques.

REFERENCES

[1] Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. 2021. Concurrent deferred reference counting with constant-time overhead. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 526–541.
[2] Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. 2022. Turning Manual Concurrent Memory Reclamation into Automatic Reference Counting. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 526–541.
[3] Henry G Baker. 1994. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *ACM Sigplan Notices* 29, 9 (1994), 38–43.
[4] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2021. OrcGC: automatic lock-free memory reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 205–218.
[5] David Detlefs, Paul Alan Martin, Mark Moir, and Guy L. Steele Jr. 2002. Lock-free reference counting. *Distributed Computing* 15, 4 (2002), 255–271.
[6] Facebook. 2020 (accessed June 5, 2020). *Facebook Open Source Library.* https://github.com/facebook/folly
[7] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.
[8] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. 2009. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. *IEEE Trans. Parallel Distrib. Syst.* 20, 8 (2009).
[9] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. 2008. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal* 47, 2 (2008), 221–236.
[10] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking Memory Management Support for Dynamic-sized Data Structures. *ACM Trans. Comput. Syst.* 23, 2 (May 2005).
[11] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504.
[12] Ruslan Nikolaev and Binoy Ravindran. 2020. Universal Wait-Free Memory Reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 130–143.
[13] Ruslan Nikolaev and Binoy Ravindran. 2021. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 987–1002.
[14] Pedro Ramalhete and Andreia Correia. [n. d.]. DoubleLink - A Low-Overhead Lock-Free Queue. http://concurrencyfreaks.blogspot.com/2017/01/doublelink-low-overhead-lock-free-queue.html.
[15] Charles Tripp, David Hyde, and Benjamin Grossman-Ponemon. 2018. FRC: a high-performance concurrent parallel deferred reference counter for C++. *Acm Sigplan Notices* 53, 5 (2018), 14–28.
[16] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. 2018. Interval-based memory reclamation. *ACM SIGPLAN Notices* 53, 1 (2018), 1–13.
[17] Anthony Williams. 2019 (accessed November 5, 2019). *just::thread Concurrency Library.* https://www.stdthread.co.uk.

[1] Available at https://github.com/cmuparlay/concurrent_deferred_rc