Wait-Free Weak Reference Counting

Matthew J. Parkinson
Azure Research, Microsoft
Cambridge, UK
mattpark@microsoft.com

Sylvan Clebsch
Azure Research, Microsoft
Cambridge, UK
sylvan.clebsch@microsoft.com

Ben Simner
University of Cambridge
Cambridge, UK
ben.simner@cl.cam.ac.uk

Abstract

Reference counting is a common approach to memory management. One challenge with reference counting is cycles that prevent objects from being deallocated. Systems such as the C++ and Rust standard libraries introduce two types of reference: strong and weak. A strong reference allows access to the object and prevents the object from being deallocated, while a weak reference only prevents deallocation. A weak reference can be *upgraded* to provide a strong reference provided there are other strong references to the object. Hence, the upgrade operation is partial, and may fail dynamically. The classic implementation of this upgrade operation is not wait-free, that is, it can take arbitrarily long to complete if there is contention on the reference count.

In this paper, we propose a wait-free algorithm for weak reference counting. The algorithm requires primitive wait-free atomic operations of "compare and swap", and "fetch and add". We provide a correctness proof of the algorithm using the Starling verification tool. We provide a full implementation in C++ and demonstrate the best and worst case performance using micro-benchmarks. For our best case scenario with high contention, our new algorithm provides ~3x more throughput, while for the worst case the new algorithm is ~85% slower. Based on a worst case analysis, we provide a second algorithm that combines the strengths of the two algorithms, and has a significantly improved worst case with ~30% overhead, while maintaining the ~3x speedup for the best case.

1 Introduction

Reference counting is a popular approach to memory management. The core idea is to track the number of incoming edges to an object, and when this number reaches zero, the object can be deallocated. This is an approximation of reachability, but when the object graph contains cycles, it is possible for objects to be unreachable but not deallocated.

One approach to reducing this problem is to count two types of references: strong and weak [1, 2]. The strong references are used to access the object, and the weak references are used to prevent the object from being deallocated. Additionally, a weak reference can be *upgraded* to a strong reference, but this upgrade will fail if there are no strong references to the object: destruction and deallocation. Strong and weak references lead to having a two stage process to collecting an object. When there are no more strong references, the object can be destructed: its out going references (both strong and weak) can be removed. When there are no more strong or weak references, then the object can actually be deallocated.

Given a cyclic object graph such as a parent pointing tree, then the parent pointers can be weak, and the child pointers can be strong. If there are no external references to the tree, then the tree can be deallocated. Although the root will have weak references to it, there are no strong references, so its destructor can run. This removes the strong references from the children, which in turn will remove the weak references from the parent, allowing the root to be collected.

The classic implementation of upgrading a counted weak reference to a strong reference is not wait-free. The implementations in the standard libraries for both C++ and Rust involve a loop that checks the current reference count value and, if it is non-zero, attempts to increment it with a compare and swap operation. Under contention this can take an arbitrarily long time to complete.

In this paper, we propose a wait-free algorithm for weak reference counting, where all operations are guaranteed to terminate in a bounded number of instructions. We assume the machine supports primitive operations for "compare and swap", and "fetch and add". The new algorithm uses the same amount of state as the classic algorithm, but uses a different state machine that allows us to avoid the loop in the upgrade operation.

To validate the algorithm, first we verify its correctness. We verify the algorithm's correctness by encoding the algorithm into the Starling verification tool [15]. Starling is based on separation logic [4, 12], which allows the proof to naturally capture the notion of owning a reference count, which is an important intuition in the correctness of the algorithm.

We provide a second validation of the algorithm, by evaluating the performance of the algorithm. We provide a C++ implementation, which we compare to the classic algorithm using two micro-benchmarks. The first is designed to exhibit the best case performance of the wait-free algorithm and the second is designed to exhibit the worst case overhead of the wait-free algorithm. The micro-benchmarks show an improvement of ~3x in the best case, and a ~85% overhead in the worst case. These results are based on micro-benchmarks and hence are not representative of real world performance gains and loses, which would be considerably smaller.

Finally, we provide an optimised algorithm that addresses the worst case performance of the wait-free algorithm. The

$$\begin{tikzcd}
\mathtt{strong}(0) & \mathtt{strong}(1) \arrow[r, bend left, "\mathtt{incref}"] \arrow[l, bend left, "\mathtt{decref}\ \{\text{+WeakRef}\}"] & \mathtt{strong}(2) \arrow[l, bend left, "\mathtt{decref}"] \arrow[r, bend left, "\mathtt{incref}"] & \cdots \arrow[l, bend left, "\mathtt{decref}"]
\end{tikzcd}$$

Figure 1. State machine for the classic algorithm

$$\begin{tikzcd}
\mathtt{closed} \arrow[loop above, "\mathtt{incref}"] & \mathtt{strong}(0) \arrow[l, "\mathtt{close}"] \arrow[r, bend left, "\{\text{-WeakRef}\}\ \mathtt{incref}"] & \mathtt{strong}(1) \arrow[l, bend left, "\mathtt{decref}\ \{\text{+WeakRef}\}"] \arrow[r, bend left, "\mathtt{incref}"] & \cdots \arrow[l, bend left, "\mathtt{decref}"]
\end{tikzcd}$$

Figure 2. State machine for the wait-free algorithm

optimised algorithm detects if there has ever been a weak-reference, if there has not been, then the algorithm is equivalent to the classic algorithm. Once a weak-reference has been created, the algorithm is equivalent to the wait-free algorithm. This combines the strengths of the two algorithms, and has a significantly improved worst case with ~30% overhead, while maintaining the ~3x speedup for the best case.

In the rest of the paper, we provide a brief overview of the classic algorithm (§2). We then describe our new algorithm (§3), and give an implementation in C++ (§4). We then verify the algorithm's correctness (§5) and evaluate the performance of the algorithm (§6). Finally, based on the worst case performance evaluation, we provide a second algorithm that combines the strengths of the classic and the wait-free algorithms (§7).

2 Classic algorithm

The classic algorithm can be modelled by the state machine in Figure 1. Effectively, the state machine enforces that once the reference count reaches zero, it cannot be incremented again.

In the classic algorithm, the thread that reaches `strong`(0) logically receives a weak reference, `WeakRef`. Effectively, while the strong reference count is non-zero, there is an additional weak reference that is owned by the object itself. When moving to `strong`(0), this weak reference is transferred to the thread that decremented the strong reference count. This weak reference is used to keep the object alive while the destructor is run. This prevents other threads from deleting the object while it is being destructed.

The state machine is normally implemented with a single machine word for the strong reference count, which reflects the current state, i.e. `strong`(5) is represented as the integer 5. The weak reference count is stored in a separate machine word. If a thread owns a strong reference, then it is not possible to be in the `strong`(0) state, and hence it is safe to atomically increment the strong reference count.

Promoting a weak-reference to a strong only requires the thread to own a weak reference. If a thread only owns a weak reference, then it is possible that the current state is `strong`(0); the object has (or is being) destructed but is not yet deallocated. It would, therefore, be unsafe to perform an atomic increment, lest this resurrects a dead or dying object by transitioning from `strong`(0) to `strong`(1). Weak reference promotion is thus implemented using a read, a conditional and then a compare and swap (CAS) instruction. This CAS may fail and need to be retried. This means it is not wait-free as it can be delayed by other threads performing operations on the strong reference count. It is, however, lock-free, as each iteration of retry must be caused by another successful operation on the reference count.

We provide a C++ implementation in Figure 3a. The CAS loop can be seen in `acquire_strong_from_weak` on lines 38-42. This is effectively the C++ standard library definition, but condensed to fit in the paper.

3 Wait-free algorithm

Our implementation differs from the classic one by using an additional state:

- `strong`(n) — the strong reference count is n (note that n can be zero).
- `closed` — there are no strong references to the object, and all future promotions will fail.

By separating `strong`(0) and `closed`, we are able to implement wait-free versions of all the standard operations. We give the state machine for this implementation in Figure 2. The key change is to make closing the strong reference count a two-step operation.

We encode the state of the strong reference count by using the low bit to represent which case we are in, and the upper 63 bits to represent the strong reference count. By packing into a single word, we can use operations that can manipulate both the bit and the count in a single atomic operation. Note that if the reference count is closed, then the value in the count is irrelevant. Thus, `closed` is represented by any odd number, and `strong`(n) is represented by $2 \times n$.

This representation lets us build the state machine in Figure 2 using just atomic `fetch_add` and `compare_exchange` operations. The `incref` operations are built using `fetch_add` of 2, and the `decref` operations are built using `fetch_add` of -2. By using `fetch_add`, we know the starting state of the `incref` and `decref` transitions.

When a thread has performed a `decref` operation, resulting in zero, then it will attempt to `close` the strong reference count to stop future upgrades creating new strong references. The `close` transition is implemented using an atomic `compare_exchange` operation, which will only succeed if the current value is `strong`(0). If it fails, then an upgrade must have successfully happened and the `close` operation is not required.

Recall with the classic algorithm, the transition from `strong`(1) to `strong`(0) provides the thread with a weak reference count. The same is true of the wait-free algorithm. However, as the state machine allows the reverse transition, we also require that a transition from `strong`(0) to `strong`(1) removes ownership of a weak reference count from the thread that performs the transition. This transition is only performed by a thread that is attempting to acquire a strong reference count from a weak one. Hence, it will lose this weak reference, which it must then re-establish by incrementing the weak reference count. This ownership transfer is formalised in the Starling proof in Section 5.

4 Implementation

In Figure 3b, we present the core C++ class for implementing wait-free weak reference counts. This can be used to build C++ smart pointers for managing memory in the standard way.[1] The full C++ implementation can be found on GitHub [10].

The implementation uses two atomic variables to store the state of the reference counts: `strong` and `weak`. It additionally holds the state of the object in body.

The two most important functions are `release_strong` and `acquire_strong_from_weak`. The other functions are identical to the classic algorithm.

The `release_strong` function is used to release a strong reference count. On line 23, due to having just performed the state machine transition from `strong`(1) to `strong`(0), we know that we have acquired ownership of a weak reference count. This is required to continue accessing the state safely. On line 25, we attempt to close the strong reference count. If this succeeds, we can destruct the underlying object. Whether or not the strong reference count was closed, we can and must release the weak reference count.

In `acquire_strong_from_weak`, we attempt to acquire a strong reference count from a weak reference count. The code checks if the strong reference count is closed on line 40-41. The lines 42-45 implement the logic to recreate the weak reference that is required by the state machine if this promotion caused the transition from `strong`(0) to `strong`(1).

In `release_weak` we apply the common optimisation of checking for the last reference count before performing the decrement. If this is observed to be the last decrement, then the update is not actually required. This is implemented on line 16.

[1]The wrapper classes were written by GitHub CoPilot, with minor human edits.

5 Correctness

In this section, we prove the wait-free algorithm is correct and wait-free.

First, we show that the algorithm is wait-free. Assuming `fetch_add` and `compare_exchange_strong` are wait-free, then proving the algorithm is actually wait-free is trivial as there are no loops.

Second, we verify the specification of the wait-free algorithm using the `Starling` tool [15]. Starling is based on ideas from separation logic [4, 8, 12] and naturally models the notions of owning a reference count.

Starling operates on its own language. The Starling syntax is sufficiently different from the C++ implementation that we present a hybrid form here to help the reader understand the proof. The full Starling proof is available on GitHub [10]. We discuss the limitations and encodings we use in the Starling proof in §5.1.

Starling effectively builds a custom separation logic using the Views Framework [3]. For the reference counting structure, we have two permissions that are used in specifications: `StrongRef` and `WeakRef`. These correspond to the caller owning a strong reference count or a weak reference count, respectively. We specify `acquire_strong` as:

$$\{\texttt{StrongRef}\}\texttt{acquire\_strong()}\{\texttt{StrongRef} * \texttt{StrongRef}\}$$

This uses the $*$ operation from separation logic to mean distinct permissions. Hence, this specification can be read as: the caller must own a strong reference count before the call and will own two after the call.

We specify `release_strong` as:

$$\{\texttt{StrongRef}\}\texttt{release\_strong()}\{\texttt{emp}\}$$

This uses the `emp` assertion to mean no permissions. Hence, this specification can be read as: the caller must own a strong reference count before the call and will own none after the call.

We specify `acquire_weak` in two ways as:

$$\{\texttt{StrongRef}\}\ \texttt{acquire\_weak()}\ \{\texttt{StrongRef} * \texttt{WeakRef}\}$$
$$\{\texttt{WeakRef}\}\ \texttt{acquire\_weak()}\ \{\texttt{WeakRef} * \texttt{WeakRef}\}$$

The first specification is for when the caller owns a strong reference count, and the second is for when the caller owns a weak reference count. In both cases, the caller owns an additional weak reference count after the call.

The final operation, `acquire_strong_from_weak`, is specified as:

$$\begin{array}{l}\{\texttt{WeakRef}\}\\ \quad\texttt{acquire\_strong\_from\_weak()}\\ \{\texttt{WeakRef} * (\texttt{return} \Rightarrow \texttt{StrongRef})\}\end{array}$$

Here we use $\Rightarrow$ to get a `StrongRef` conditionally on the `return` value of the function. The call preserves the weak reference count.

Starling proofs consist of two components: (1) a set of constraints that give a set of separation logic-like predicates

```cpp
template <typename T>
class RCObjectClassic
{
  static constexpr size_t SINGLE_RC = 1;
  std::atomic<size_t> strong{SINGLE_RC};
  std::atomic<size_t> weak{1};
  T body;


public:
  template <typename... Args>
  RCObjectClassic(Args &&...args)
      : body(std::forward<Args>(args)...) {}

  void release_weak() {
    if ((weak == 1) || (--weak == 0))
      free(this);
  }

  void release_strong() {
    if (strong.fetch_sub(SINGLE_RC) != SINGLE_RC)
      return;
    body.~T();
    release_weak();
  }

  void acquire_weak() {
    ++weak;
  }

  void acquire_strong() {
    strong.fetch_add(SINGLE_RC);
  }

  bool acquire_strong_from_weak() {
    auto curr =
      strong.load(std::memory_order_relaxed);
    while (curr != 0) {
      auto next = curr + SINGLE_RC;
      if (strong.compare_exchange_weak(curr, next))
        return true;
    }
    return false;
  }
};
```

(a) Classic algorithm

```cpp
template <typename T>
class RCWaitFree
{
  static constexpr size_t SINGLE_RC = 2;
  static constexpr size_t CLOSED = 1;
  std::atomic<size_t> strong{SINGLE_RC};
  std::atomic<size_t> weak{1};
  T body;

public:
  template <typename... Args>
  RCWaitFree(Args &&...args)
      : body(std::forward<Args>(args)...) {}

  void release_weak() {
    if ((weak == 1) || (--weak == 0))
      free(this);
  }

  void release_strong() {
    if (strong.fetch_sub(SINGLE_RC) != SINGLE_RC)
      return;
    // Implicitly holds a weak self-reference here.
    size_t old = 0;
    if (strong.compare_exchange_strong(old, CLOSED))
      body.~T();
    release_weak();
  }

  void acquire_weak() {
    ++weak;
  }

  void acquire_strong() {
    strong.fetch_add(SINGLE_RC);
  }

  bool acquire_strong_from_weak() {
    auto old = strong.fetch_add(SINGLE_RC);
    if ((old & CLOSED) != 0)
      return false;
    if (old == 0)
      // Blocked closing; restore implict weak
      // self-reference
      acquire_weak();
    return true;
  }
};
```

(b) Wait-free algorithm

Figure 3. Implementation of algorithms

```
1   void release_strong() {
2     // StrongRef
3     if (strong.fetch_sub(2) != 2)
4       return;
5     // WeakRef
6     last = strong.compare_exchange(0, 1);
7     // if last {Destruct} else {WeakRef}
8     if (last)
9       // Destruct
10      body.~T();
11      // WeakRef
12    // WeakRef
13    release_weak();
14  }
15
16  bool acquire_strong_from_weak() {
17    // WeakRef
18    auto old = strong.fetch_add(2);
19
20    // if (old % 2 = 0) {StrongRef} *
21    //   if (old ≠ 0) {WeakRef}
22    if (old % 2 != 0)
23      // WeakRef
24      return false;
25
26    // StrongRef * if (old ≠ 0) {WeakRef}
27    if (old == 0)
28      // StrongRef
29      acquire_weak();
30      // StrongRef * WeakRef
31
32    // StrongRef * WeakRef
33    return true;
34  }
```

Figure 4. Overlaid starling proof onto original C++ code.

a semantics and (2) a set of proof outlines for the code. The system behaves similarly to the Owicki-Gries method [9], but uses a logic for proof outlines that looks like separation logic.

In addition to the two permissions `StrongRef` and `WeakRef`, we also have `Destruct` and `Dealloc` permissions. These correspond to the permission to call the destructor and to deallocate the underlying object, respectively. These are not part of the specification but are needed in the proof outline.

In Figure 4, we present the proof outlines of the two most interesting methods: `release_strong` and `acquire_strong_from_weak`. We overlay the Starling proof outline onto the original C++ code.

For the proof outline of `release_strong`, on line 5 we have the permission `WeakRef`. This is modelling the state machine in Figure 2, where we move from `strong`(1) to `strong`(0). We see the reverse of this on line 21, where the `WeakRef` is removed if the `old` value was 0. The `WeakRef` is re-established by the call to `acquire_weak` on line 29.

On line 7, we see that the `Destruct` permission is only present if the reference count has been closed and hence `last` is true.

The second aspect of a Starling proof is a set of constraints that provide a first-order logic interpretation of the permissions. In the interpretation, we introduce two variables to represent if the object has been `destructed` and `deallocated`. The first constraint provides a basic invariant that must always be true:

```
constraint emp →
    (strong%2 = 0 ⇒ ¬destructed) ∧
    (weak > 0 ⇒ ¬deallocated) ∧
    (strong > 0 ∧ strong%2 = 0 ⇒ weak > 0)
```

This gives some basic correctness and enforces that the program does not break the assumptions about when the object can be destructed and deallocated.

We provide the interpretation of the `Destruct` and `Dealloc` permissions in the following two constraints:

```
constraint Destruct →
    ¬destructed ∧ strong%2 = 1 ∧ weak > 0
constraint Dealloc →
    ¬deallocated ∧ weak = 0
```

The interpretation in Starling must also account for the combination of permissions using ∗. For the `Destruct` and `Dealloc` permissions, we have the following constraints:

```
constraint Destruct * Destruct → false
constraint Dealloc * Dealloc → false
```

This ensures that the Starling tool must show it is never possible for two `Destruct` or two `Dealloc` permissions to be held at the same time by any threads. Due to the underlying theory being intuitionistic (adding permissions only reduces the set of possible states), this also means there cannot be more than two held at the same time.

The interpretation of the `StrongRef` permissions must account for having an arbitrary number of them:

```
constraint iter[n] StrongRef →
    n > 0 ⇒ (strong ≥ n × 2 ∧ strong%2 = 0)
```

This states that if you are interpreting n `StrongRef` permissions where $n > 0$, then the state cannot be closed, and the reference count must contain at least the n. That is, we are in the state `strong`(m) where $m \geq n$.

The final and most complex interpretation is for the WeakRef permission:

$$\begin{array}{l}\texttt{constraint iter}[n]\ \texttt{WeakRef} \longrightarrow \\ \quad n > 0 \Rightarrow \\ \qquad (\neg\texttt{destructed} \land \texttt{strong\%2} = 1 \land \texttt{weak} \geq n+1) \lor \\ \qquad (\texttt{strong\%2} = 0 \land \texttt{strong} > 0 \land \texttt{weak} \geq n+1) \lor \\ \qquad (\texttt{destructed} \land \texttt{weak} \geq n) \lor \\ \qquad (\texttt{strong} = 0 \land \texttt{weak} \geq n)\end{array}$$

This must account for the giving and taking of weak reference permissions on the transition into and out of the strong(0) state. It also accounts for the Destruct permission effectively holding a single weak reference count permission.

Although these logical statements are tricky, the Starling tool can quickly prove that each step in the proof outline respects the interpretation of the permissions, and moreover, that it preserves any possible concurrent context. Hence, it proves that the operations satisfy the specifications given earlier.

The starling proof indirectly shows that any access cannot occur once the destructor has run. If we assume that any thread accessing the protected object has a strong reference count, then it must have a StrongRef predicate. By the definition of the StrongRef predicate, we know $\texttt{strong\%2} = 0$, and thus by the definition of Emp, we know the destructor cannot have run.

5.1 Limitations of proof

The Starling proof tool is naturally suited to this form of problem. However, there are a few places where we were not able to prove as strong a property as we wanted.

The first is dynamic allocation. The full API allows the allocation of objects at runtime. Starling does support this with an extension to use Grasshopper [11]. However, we stayed in the more stable fragment, and just represented a single object, by using global variables for its fields. This keeps the complexity down, while covering the majority of the cases.

We did not model the actual destruction and deallocation of objects, but instead used additional global variables to encode the allocation status of the single object that the proof considers.

The Starling proof logic is intuitionistic. As such, it allows permissions to be leaked. Thus, the proof does not guarantee that the object is not leaked. We are confident the code does not leak references, but it is not shown by the tool.

The proof we discuss in the paper does not guarantee that the destructor is called before deallocation occurs. Given the specification of release_weak, it is a perfectly valid implementation to call it directly after line 5 of Figure 4. We can fix this deficiency in the proof using an alternative type of WeakRef permission on line 5. However, we were not able to encode that into Starling without using an auxiliary variable to the proof that tracks the number of threads on line 5 of Figure 4. The extended Starling proof is available online [10]. The C++ implementation contains an optimised reference release. We do not encode this into the Starling proof, as it would also require an auxiliary variable to model taking the non-decrementing path. As this is not the aspect of our algorithm that is different from the classic algorithm, we have not modelled it.

Starling does not provide sufficient modulo arithmetic to use the bottom bit trick. We instead encode the invariants in terms of two variables, the bottom bit and the actual strong reference count. Starling supports elaborate atomic operations over multiple variables, so we are able to encode the correct behaviour without the bottom bit trick.

6 Evaluation

We evaluate our implementations using two micro-benchmarks that exhibit the best and worst case behaviour of the wait-free algorithm (Figure 3b) with respect to the classic algorithm (Figure 3a). All our experiments were run on an Azure F72s v2 instance running Ubuntu, which has 72 hardware threads. As one of our micro-benchmarks performs allocation, we use snmalloc [5] as the system allocator to minimise the effect of the allocator. All the benchmarking code, scripts for generating graphs, and unprocessed data are available on GitHub [10].

The first benchmark is designed to illustrate the base case of the algorithm. It repeatedly acquires a strong reference from a weak reference, and then immediately drops the strong reference. We perform this operation a million times, and divide the operation across a set of threads. The results are shown in Figure 5, where we plot the time taken to perform the upgrade and release operations a million times as a function of the number of threads. The results demonstrate that the wait-free algorithm performs better in this scenario across all core counts. With a single thread the classic algorithm has ~40% overhead as compared to the wait-free algorithm. As the core count increases, so does the overhead. At the limit of the machine with 72 hardware threads, the wait-free algorithm is ~3x faster than the classic algorithm. We also note that the performance worsens once the number of threads exceeds the number of physical cores, i.e. 36 hardware threads.

To further refine this benchmark, we measured the individual cycle count of a single upgrade operation and the subsequent strong release. The results are shown in Figure 6. We present a cumulative distribution function to illustrate that the cycle counts for the wait-free are generally much lower than the classic algorithm. The graphs show that, in almost all circumstances, the wait-free algorithm has a lower cycle count cost than the classic algorithm. This leads to lower latency in upgrading a reference count. For instance, for 36 threads the wait free algorithm completed over 50%

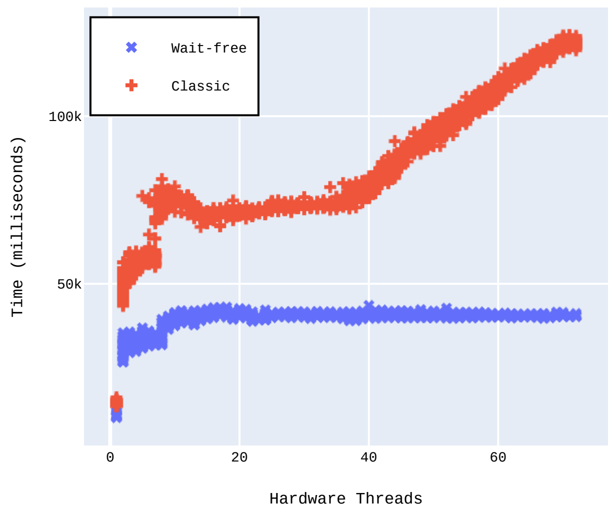

Figure 5. Throughput of upgrading a weak reference to a strong reference

of the operations within 2900 cycles, whereas the classic algorithm took over 9900 cycles to reach the same point.

To get accurate cycle counts, we used instructions to prevent speculative execution around the operations.[2] This has the effect of increasing the running cost, but allows for a more accurate representation of the cost of contention. We distinguished a sampling thread from interfering threads. The n-1 interfering threads all performed the upgrade and release operations until the sampling thread had 1000 samples, and repeated the experiment 600 times. We ignored the first five runs to avoid any warm-up effects.

The second micro-benchmark is designed to exhibit the worst case behaviour of the wait-free algorithm. It repeatedly creates a new object, and then releases it. This causes the more complex path of the wait-free algorithm to be taken, where it must perform both the `fetch_add` and the `compare_exchange` to close the strong reference count. This benchmark shows that the wait-free algorithm has ~85% overhead in the worst case when hyper-threading is used. This is expected, as the wait-free algorithm has two atomic operations instead of one.

As each operation actually allocates a new object and then frees it, we also measure the `snmalloc` costs for those operations. We can see that the `snmalloc` costs are small but grow when the machine reaches its physical core count. This is expected, as at least one pair of threads are sharing a core, and hence the wall-clock time approximately doubles.

The microbenchmarking shows very promising results for the wait-free algorithm. These benchmarks have been presented to show the extremes in performance of the wait-free algorithm. As with all micro-benchmarks the performance of real systems will be affected far less than shown here. We address the worst case in the next section.

7 Optimisation

The wait-free algorithm has a worst case cost that is high in what could be a common scenario. With the algorithm given earlier, we are forced to pay the cost of two atomic instructions even if we do not use weak references. Next, we present an algorithm that effectively combines the best of both algorithms.

The key insight in the optimised wait-free algorithm is to detect if there has ever been a weak reference to an object. If there has not been a weak-reference, then we use the classic algorithm for strong reference counts, and if there has been a weak reference, then we use the wait-free algorithm. This provides an algorithm that performs close to the classic algorithm when there are no weak references, and performs close to the wait-free algorithm when there are weak references.

To track if there has ever been a weak reference to an object, we borrow a second bit from the representation of the reference count to be the weak reference bit. This bit is set to 1 when a weak reference is created, and is never reset. This bit is then used to allow the state machines for the classic algorithm, and the wait-free algorithm to be combined into a single state machine (Figure 8).

The first thread that creates a weak reference is responsible for setting the weak reference bit. The transition labelled `weak` in Figure 8 is the transition that sets the weak reference bit. We use `strong` for counts without the bit set, and $\texttt{strong}^W$ for ones where there is potentially a weak reference.

With this more refined state machine, the state `strong`(0) does not need to wait for weak references to be removed before deallocating the object. We do not need to delay the deallocation as there cannot be any weak references if we are in this state.

We present the code in Figure 9. The `release_strong` method first determines if decrement has led to `strong`(0) or $\texttt{strong}^W(0)$ state. If it has not then it returns immediately. Otherwise, it performs a second check to determine if the weak reference bit is set. Based on that it either runs the destructor and deallocates the object, or attempts to `close` the strong reference count.

The `acquire_weak` code, first attempts to detect if this is the first weak reference. If it is, then attempts to increase it from 0 to 2 using a CAS. It attempts to add two weak-references one for the current thread, and one owned by the reference count object that is used to protect the threads attempting to close the strong reference count.

If the CAS succeeds, then this thread is responsible for setting the weak reference bit. It is possible for other threads to successfully create a weak-reference between the CAS and the setting of the weak reference bit. This is not a problem

[2] https://www.intel.de/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

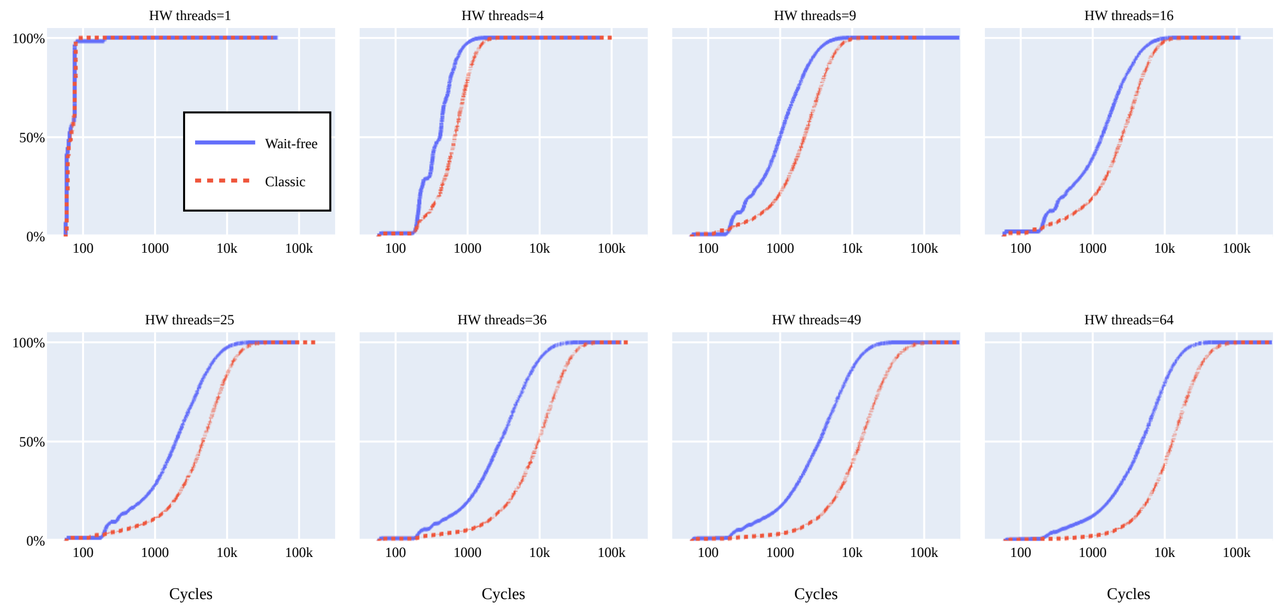

Figure 6. Percentage of operations terminating within a specified cycle count for a single upgrade operation and the subsequent strong release.

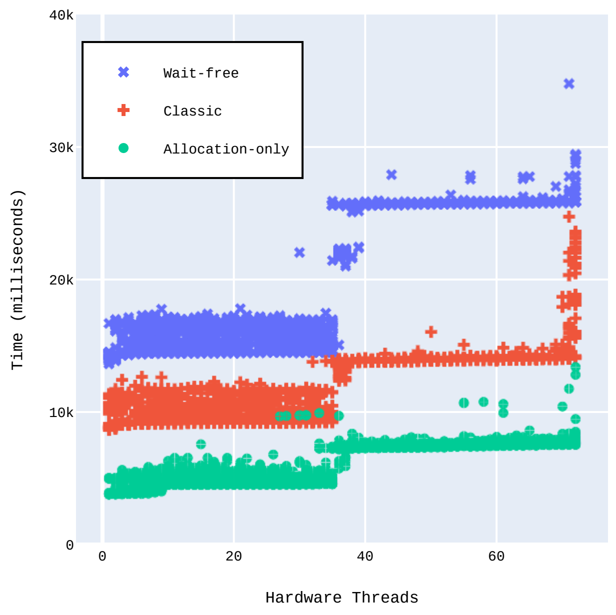

Figure 7. Throughput of 1 million strong references releases per thread.

as the strong reference count cannot reach zero until the weak-reference bit is set. This is because a thread that is attempting to set the weak-reference bit must own a strong reference count.

We rerun the worst case scenario for the wait-free algorithm and present the results in Figure 10. This illustrates that the optimised algorithm does not suffer any overhead on this case. This is expected because the optimised algorithm does very little additional work in this case. We present two hardware thread counts, one not using hyper-threading and one using hyper-threading.

We also consider the worst case scenario for the optimised wait-free algorithm. This is where a single thread creates a reference counted object, creates a single weak-reference, and then drops both the strong and weak references it owns. We present the results in Figure 11. Here we see that the optimised algorithm is the worst of the three algorithms with an overhead of approximately 30%. This benchmark is design to defeat the optimised path and cause the maximum overhead. As a worst case this is less common than the previous worst case.

We reran the previous best cases and the results between the optimised wait-free algorithm and the wait-free algorithm are indistinguishable.

We prove (1) that the optimised algorithm is wait-free, as we need no loops and it only uses other wait-free primitives; and, (2) that the optimised algorithm correctly implements its specification, by writing a proof which Starling checks.

We do not prove that the object is destroyed before deallocation, and like with the proof of the wait-free algorithm Starling cannot check that references are not leaked, we prove that the algorithm is correct but not that the included C++ implementation implements it correctly.

The optimised wait-free algorithm is proved similarly to the wait-free algorithm, by implicitly gaining a `WeakRef`-like view when releasing the last strong reference, paired

$$\begin{tikzcd}
 & \texttt{strong}(0) & \texttt{strong}(1) \arrow[l, bend left, "\texttt{decref}"] \arrow[r, bend left, "\texttt{incref}"] \arrow[d, "\texttt{weak}"] & \texttt{strong}(2) \arrow[l, bend left, "\texttt{decref}"] \arrow[r, bend left, "\texttt{incref}"] \arrow[d, "\texttt{weak}"] & \cdots \arrow[l, bend left, "\texttt{decref}"] \\
\texttt{closed} \arrow[loop above, "\texttt{incref}"] & \texttt{strong}^W(0) \arrow[l, "\texttt{close}"] \arrow[r, bend left, "\texttt{incref}"] & \texttt{strong}^W(1) \arrow[l, bend left, "\texttt{decref}"] \arrow[r, bend left, "\texttt{incref}"] & \texttt{strong}^W(2) \arrow[l, bend left, "\texttt{decref}"] \arrow[r, bend left, "\texttt{incref}"] & \cdots \arrow[l, bend left, "\texttt{decref}"]
\end{tikzcd}$$

Figure 8. State machine for the optimised wait-free algorithm

```cpp
void release_strong() {
  auto old = strong.fetch_sub(SINGLE_RC);
  if (old > (SINGLE_RC + WEAK))
    return;

  if ((old & WEAK) == 0) {
    body.~T();
    free(this);
    return;
  }

  old = WEAK;
  if (strong.compare_exchange_strong(old, CLOSED))
    body.~T();
  release_weak();
}

void acquire_weak() {
  size_t old = weak;
  if (old == 0) {
    if (weak.compare_exchange_strong(old, 2)) {
      strong += WEAK;
      return;
    }
  }
  weak++;
}
```

Figure 9. Optimised wait-free algorithm

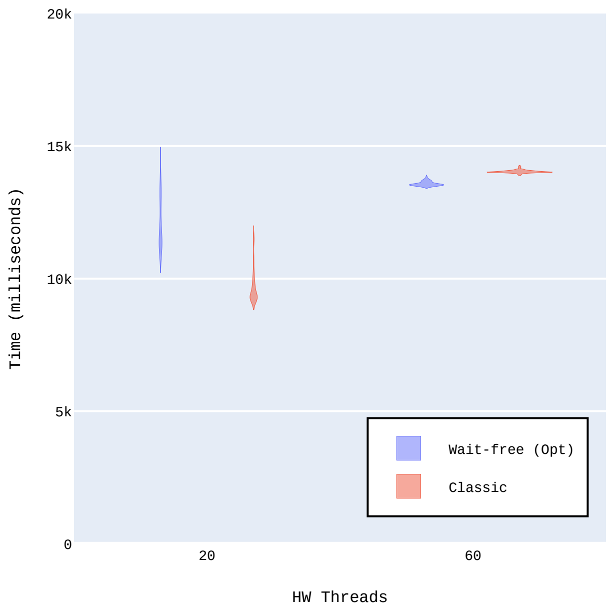

Figure 10. Optimised wait-free algorithm on previous worst-case

with losing a WeakRef when acquiring a strong reference on incref from strong^W(0).

For the proof we introduce variants of the WeakRef and StrongRef views: WeakRefShared to represent holding the implicit WeakRef gained on decref to strong^W(0); WeakRef-Closed for a weak reference to the closed object which is not yet destroyed; StrongRefWeakBitNotSet which is gained immediately after incrementing the weak reference count, but before setting the weak bit; and StrongRefWeakNotZero for holding a strong reference where the weak reference count is not zero. Figure 12 presents the proof outline, overlaid on the C++ code from Figure 9 for the release_strong and acquire_weak methods. Note that it only shows the proof for the specification of acquire_weak with the Strong-Ref precondition, for brevity. The full proof follows the outline, with constraints for the views based on those from Section 5, but where particular care is required around the weak bit. The proof cannot assume that the weak bit has been set when given a weak reference, as the setting of the bit does not happen atomically when incrementing the weak counter, and the bit becomes unset again when closing the strong counter. If there is a WeakRef and the weak bit is not set, then either there must be a StrongRef and therefore the count cannot be closed yet, or the strong reference count must already have been closed. This is captured by the

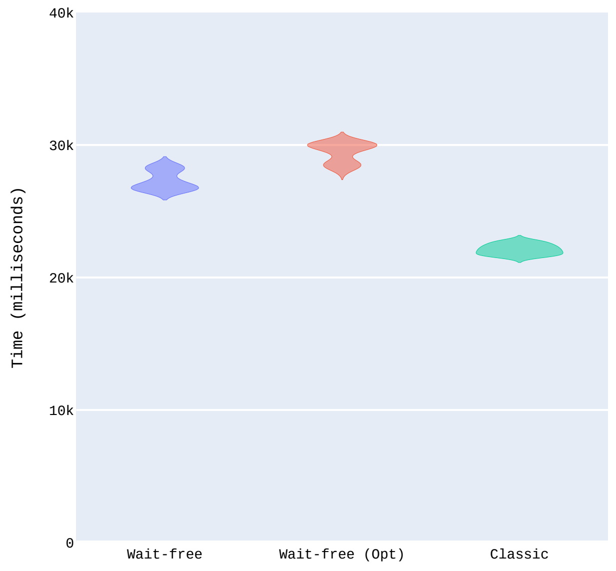

Figure 11. Optimised wait-free algorithm on new worst-case

following constraint:

```
constraint
  iter[w] WeakRef *
  iter[wc] WeakRefClosed *
  iter[ws] WeakRefShared *
  iter[s] StrongRef *
  iter[sw] StrongRefWeakBitNotSet *
  iter[snz] StrongRefWeakNotZero ⟶
    w > 0 ⟹ (
      (weak_bit ∧ s + sw + snz > 0
         ∧ weak > w + wc + ws)
      ∨ (weak_bit ∧ ws > 0)
      ∨ (¬weak_bit ∧ (closed ∨  sw > 0))
    )
```

This says that that having a WeakRef means either: the weak bit is set and there are strong references and so there must be the +1 reference initially set by acquire_weak; or there are shared weak references and the weak bit is set but not necessarily requiring there to be strong references (although it is permitted); or if the weak bit is not set then either the strong reference count is already closed or there is a strong reference being held in the state waiting to set the weak bit, after updating the weak reference count. The proof then contains a number of other constraints which exactly capture which states the counters can be in given the state of the weak bit.

The full Starling proof can be found on GitHub [10].

Overall, we believe the optimised version is a good candidate for a default implementation to replace the existing classic algorithm.

```
void release_strong() {
  // StrongRef
  auto old = strong.fetch_sub(SINGLE_RC);

  // if (old > (SINGLE_RC + WEAK))
  //   {emp}
  // else if ((old & WEAK) = 0)
  //   {Destruct}
  // else
  //   {WeakRefShared}

  if (old > (SINGLE_RC + WEAK))
    return;

  if ((old & WEAK) == 0) {
    // Destruct
    body.~T();
    return;
  }

  // WeakRefShared
  old = WEAK;
  if (strong.compare_exchange_strong(old, CLOSED))
    // WeakRefClosed
    body.~T();
  // WeakRef
  release_weak();
}

void acquire_weak() {
  // StrongRef
  size_t old = weak;
  // if (old = 0)
  //   then {StrongRef}
  //   else {StrongRefWeakNotZero}
  if (old == 0) {
    // StrongRef
    if (weak.compare_exchange_strong(old, 2)) {
      // StrongRefWeakBitNotSet
      strong += WEAK;
      // StrongRef * WeakRef
      return;
    }
    // StrongRefWeakNotZero
  }
  // StrongRefWeakNotZero
  weak++;
  // StrongRef * WeakRef
}
```

Figure 12. Overlaid Starling proof sketch of the optimised wait-free algorithm onto the original C++.

8 Related work

Weak references have been around for almost 40 years [2]. They have appeared in many languages, either integrated with reference counting (such as in C++, Python and Rust), or integrated into a tracing GC (such as in Java and C#). The implementation of weak-references in a tracing GC are very different to a reference counted system. The tracing GC is able to remove the incoming weak references rather than having the two stage deallocation process that is in a reference counted system with weak references. Our approach in this paper can be applied to reference counted systems, rather than tracing GC.

We are not aware of any other work that provides wait-free weak references in a reference counted system. The closest work to this paper is on using reference counting for memory management in lock-free and wait-free algorithms [7, 13, 14]. These works use the same representation as we develop here for the strong reference count with an additional state of closed being represented by the bottom bit. Their use is different as they are attempting to handle optimistic reads that later add a reference count. If the reference count is closed, then the optimistic read must be rolled back. Additional checks are required, such as hazard pointers [6], before memory can be reused. These works are solving a different problem to us, but with a similar solution.

9 Conclusion

In this paper, we present a new implementation of weak reference counting that is wait-free. We provide a formal proof of aspects of the correctness of the algorithm using the `Starling` tool. We evaluate the performance of the algorithm and show that it is faster than the classic algorithm in the best case, but has an overhead in the worst case.

We present a more complex algorithm that effectively combines the classic and the wait-free algorithm. It has a much better performance in the worst case, while maintaining the benefits of the wait-free algorithm.

The implementation here could be used as a drop-in replacement both in C++'s reference counted smart pointers (`shared_ptr` and `weak_ptr`), and in Rust's `Arc<T>` and `Weak<T>`. In C++, due to some aspects of the C++ API being implemented in the header files, it would require a revision of the library's ABI. In Rust, by simply changing the standard library implementation and recompiling the program, it could use this approach.

Acknowledgments

We thank Wes Filardo, Marios Kogias, the ISMM'23 anonymous reviewers for their feedback on the paper, and Mike Dodds and Matt Windsor for discussions about the proof in Starling. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC AdG ELVER).

References

[1] T. H. Axford. Reference Counting of Cyclic Graphs for Functional Programs. *The Computer Journal*, 33(5):466–470, 01 1990.

[2] D. R. Brownbridge. Cyclic reference counting for combinator machines. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*, pages 273–288, Berlin, Heidelberg, 1985. Springer-Verlag.

[3] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 287–300, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. Association for Computing Machinery.

[5] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. Snmalloc: A message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2019, pages 122–135, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Maged M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[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.

[8] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, Berlin, Heidelberg, 2001. Springer-Verlag.

[9] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[10] Matthew J. Parkinson, Sylvan Clebsch, and Ben Simner. Wait-free weak reference counting: Supplementary material. https://github.com/microsoft/verona-artifacts/tree/main/WFWeakRC, 2023.

[11] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Grasshopper. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–139, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[12] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[13] H. Sundell. Wait-free reference counting and memory management. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10 pp.–, 2005.

[14] John David Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, USA, 1995. UMI Order No. GAX95-44082.

[15] Matt Windsor, Mike Dodds, Ben Simner, and Matthew J. Parkinson. Starling: Lightweight concurrency verification with views. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 544–569, Cham, 2017. Springer International Publishing.