

The Space Cost of Lazy Reference Counting

Hans-J. Boehm



Some figures are taken from the corresponding paper in the Proceedings of the 2004 POPL Symposium, and are reproduced by permission of ACM.

Automatic Memory Management

- **Tracing garbage collection**
 - Periodically trace through all live objects.
 - Reclaim untraced objects.
 - Common in language runtimes.
- **Reference Counting**
 - Associate count of incoming references with objects.
 - Reclaim object when count reaches zero.
 - Common in OS file systems, some C++ libraries.
 - Used occasionally in language runtimes.

Tradeoffs

- Classic reference count disadvantages:
 - Leaks cyclic garbage.
 - Expensive pointer assignments, (threads!)
- Classic reference count advantages:
 - Faster reuse: better cache behavior.
 - ~~Synchronous/deterministic finalization.~~
 - Possibly better memory utilization.
 - Supports copy avoidance.
 - **Avoids GC pauses?**

Classic Reference Counting

- When creating reference to p :

```
incr(p) { count(p)++; }
```

- When deleting a reference to p :

```
decr(p) {  
    if (--count(p) == 0) {  
        invoke decr on embedded references;  
        free(p);  
    }  
}
```

- Pointer assignment requires `incr(new)` followed by `decr(old)`.
- Count update/test must be atomic.
- Incr/decr usually inserted automatically.

Pauses avoided

```
big = make_huge_linked_tree();  
do forever {  
    temp = new foo();  
}
```

- Each implied incr()/decr() takes constant time.
- No significant pauses during execution.
- Simple tracing GC would trace `big` repeatedly, introducing pauses.

...but only sometimes

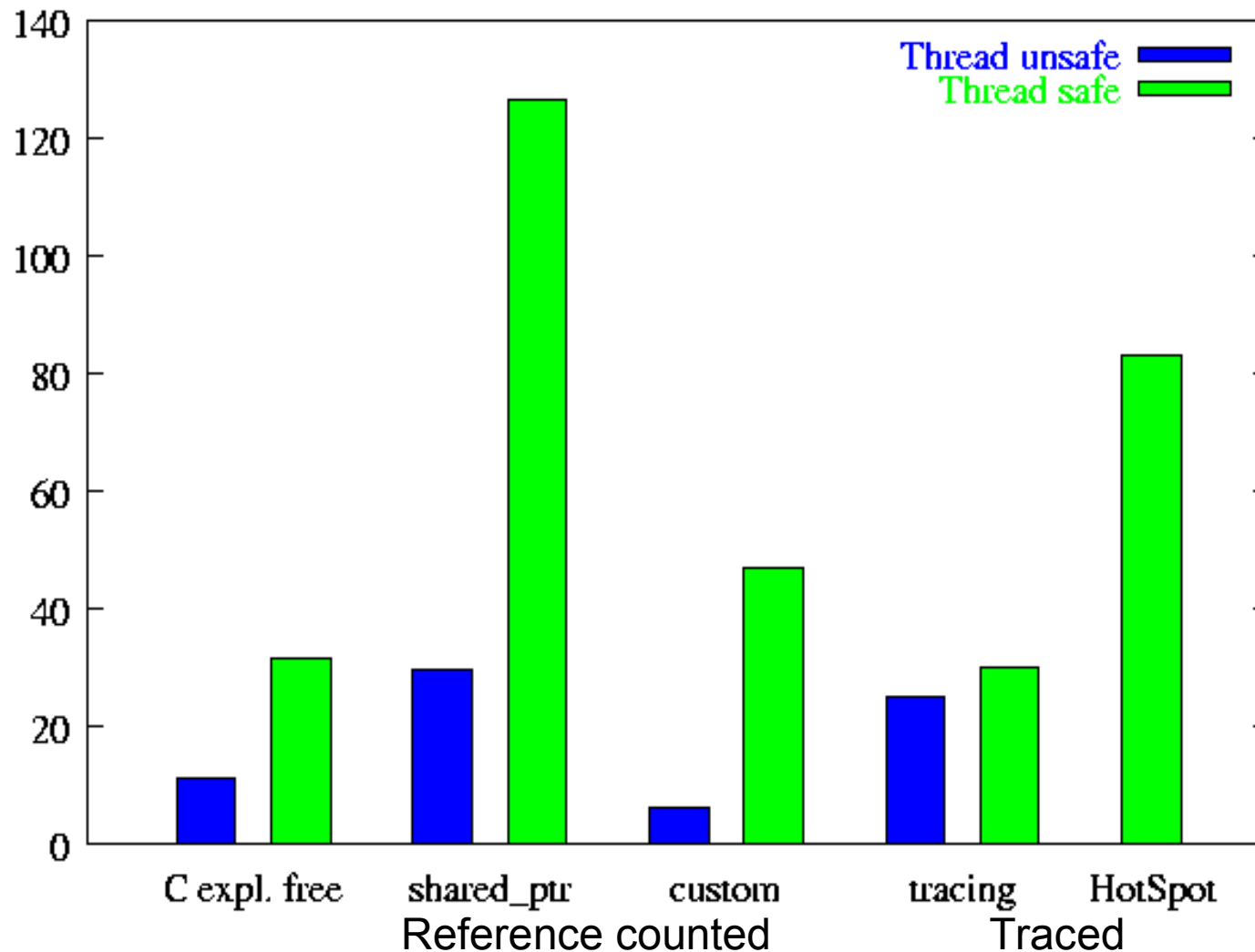
```
do forever {  
    tmp = make_huge_linked_tree();  
    ...  
    big = tmp;  
    ...  
}
```

- “big = tmp” assignment invokes `decr()` on old tree.
- Count becomes zero.
- `Decr()` recurses, deallocating and touching entire tree.
- Effectively a significant pause during assignment.

Tracing vs. ref. counting pauses

- Simple tracing GC stops entire process.
- Classic reference counting stops thread
 - ...which may hold critical lock.
- If deallocation time were predictable, we could easily deallocate manually.
 - Pauses are effectively unpredictable.
- Manual deallocation also adds “pauses”.
 - They are usually predictable.
- Recursive `decr()` calls need atomicity.
 - Usually more sensitive to threading.

Worst case “pause” times for GCBench (msecs, P4 2.0GHz, gcc, Linux)



Lazy deletion

- Decr(p):

```
if (--count(p) == 0)
    add p to to-be-freed;
```

- Before each allocation:

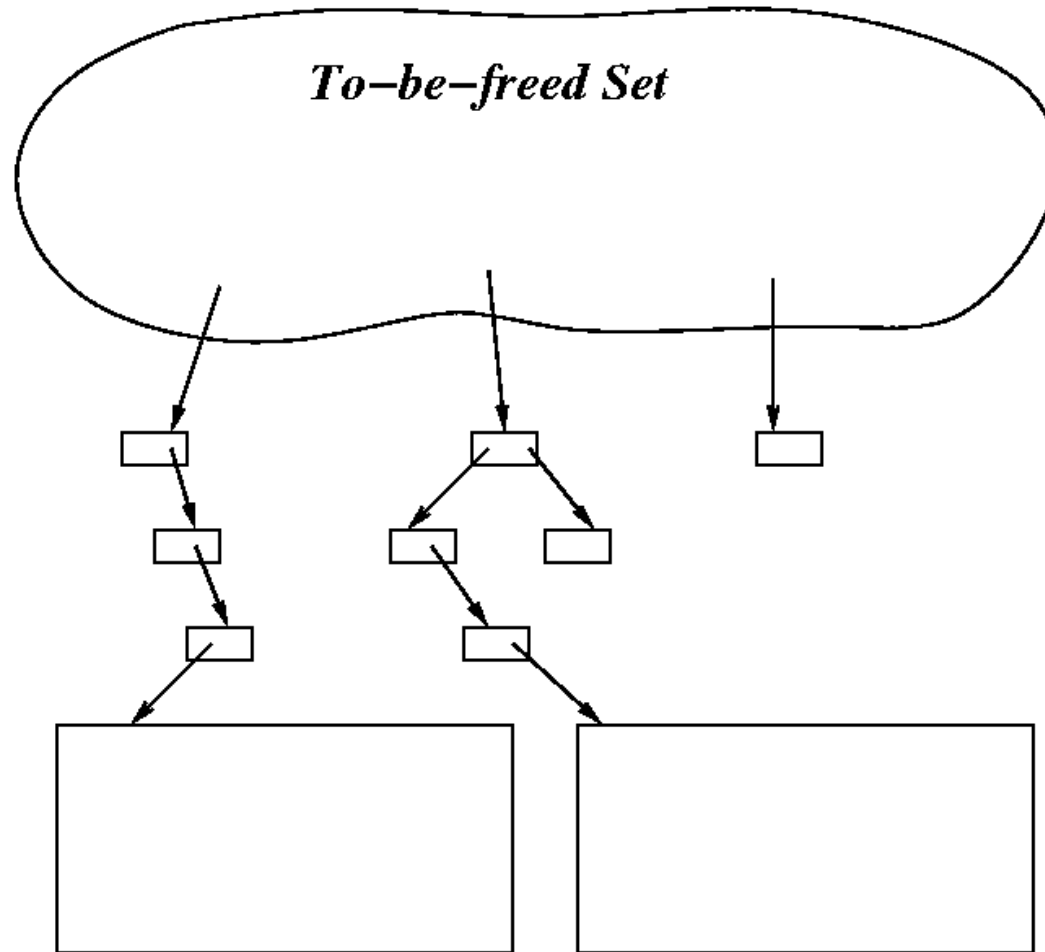
```
q = element of to-be-freed (if any);
invoke decr on embedded references;
free(q);
```

- Each allocation does one deallocation.

Lazy deletion (contd.)

- Dates back to 1963 paper (Weizenbaum).
- Works well for fixed size objects.
- But for multiple object sizes:
 - No sufficiently large object may be free.
 - Sufficiently large objects may be in to-be-freed set.
 - We may require additional heap size to satisfy allocations.

Hidden space



How much space overhead?

- Ignore fragmentation cost.
- Objects between s_{min} and s_{max} in size.
- We measure space overhead as:
$$\frac{\text{max size of allocated objects (incl. to-be-freed)}}{\text{max size of live objects}}$$
- Fragmentation adds at most a factor of $O(\log(s_{max}/s_{min}))$ to total heap size.
(Robson, 1971).

Space overhead (contd.)

- Lazy reference counting cannot increase the number of allocated objects above maximum number of live objects.
- Hence

$$\frac{\text{max allocated size}}{\text{max live size}} \leq \frac{S_{max}}{S_{min}}$$

Main result

- The preceding bound is asymptotically optimal.
- This holds for a large class of variants of the preceding algorithm.
- Smooth tradeoff:
allocation deallocates m items →
bound reduced by factor of m

Observations

- It may take a heap of size $\Omega(N^2)$ to accommodate N live bytes.
- If an n byte allocation deallocates at least n bytes, the max number of allocated bytes can't exceed the the max number of live bytes $\rightarrow$ only fragmentation overhead.
 - May require $s_{\max}/s_{\min}$ deallocations for a single allocation.

More precise statement

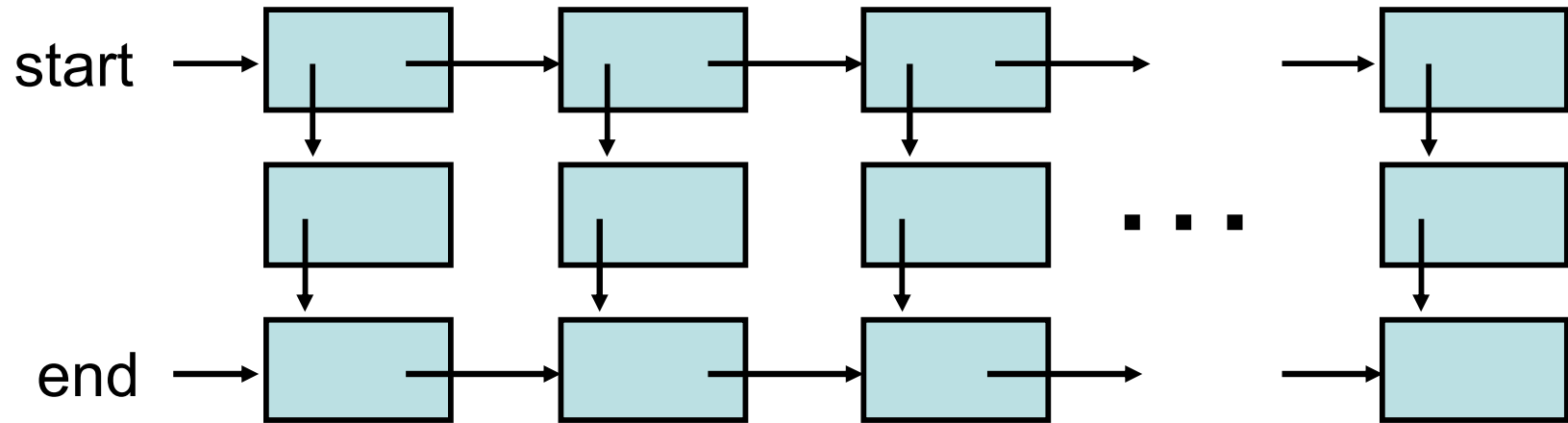
- Assuming that:
 - s_{min} and s_{max} satisfy some assumptions.
 - Affect only constants.
 - We have a “lookahead-free reference count implementation”.
 - Every sequence of 1 allocation, 2 $incr()$, and 2 $decr()$ calls deallocate $\leq m$ objects.
- There exists a “program” with no more than N referenced (live) bytes, such that:
 - The total number of allocated bytes is at least

$$\frac{N s_{max}}{2 m s_{min}}$$

Proof illustration

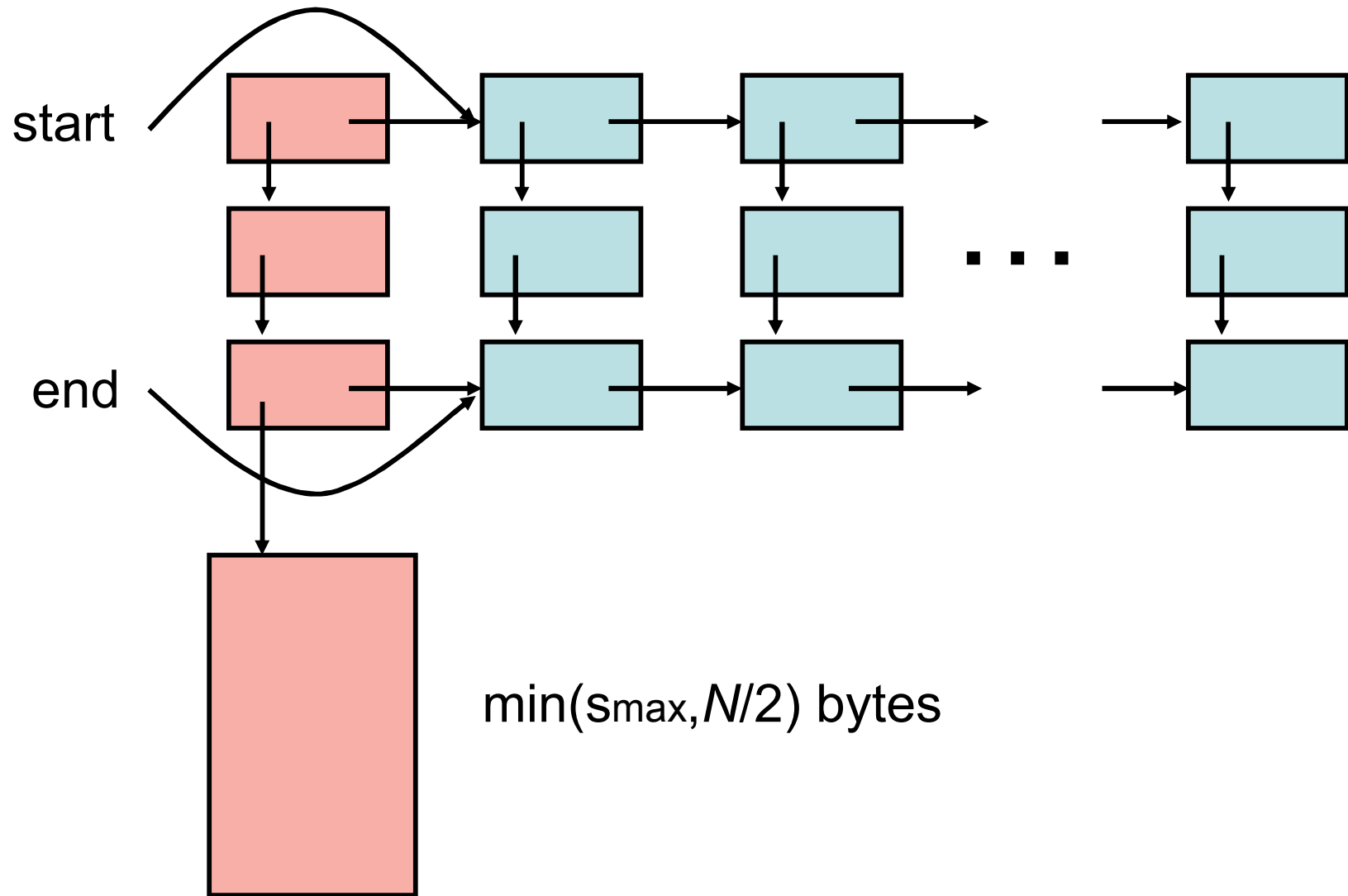
- We construct a program.
- Real proof adapts to deallocation order of reference counting algorithm.
- Here we assume instead:
 - $m = 3$
 - each allocation deallocates 3 *to-be-freed* items.
 - *to-be-freed* set is managed in LIFO order.

Step 1 – Allocate $N/2$ bytes in small objects

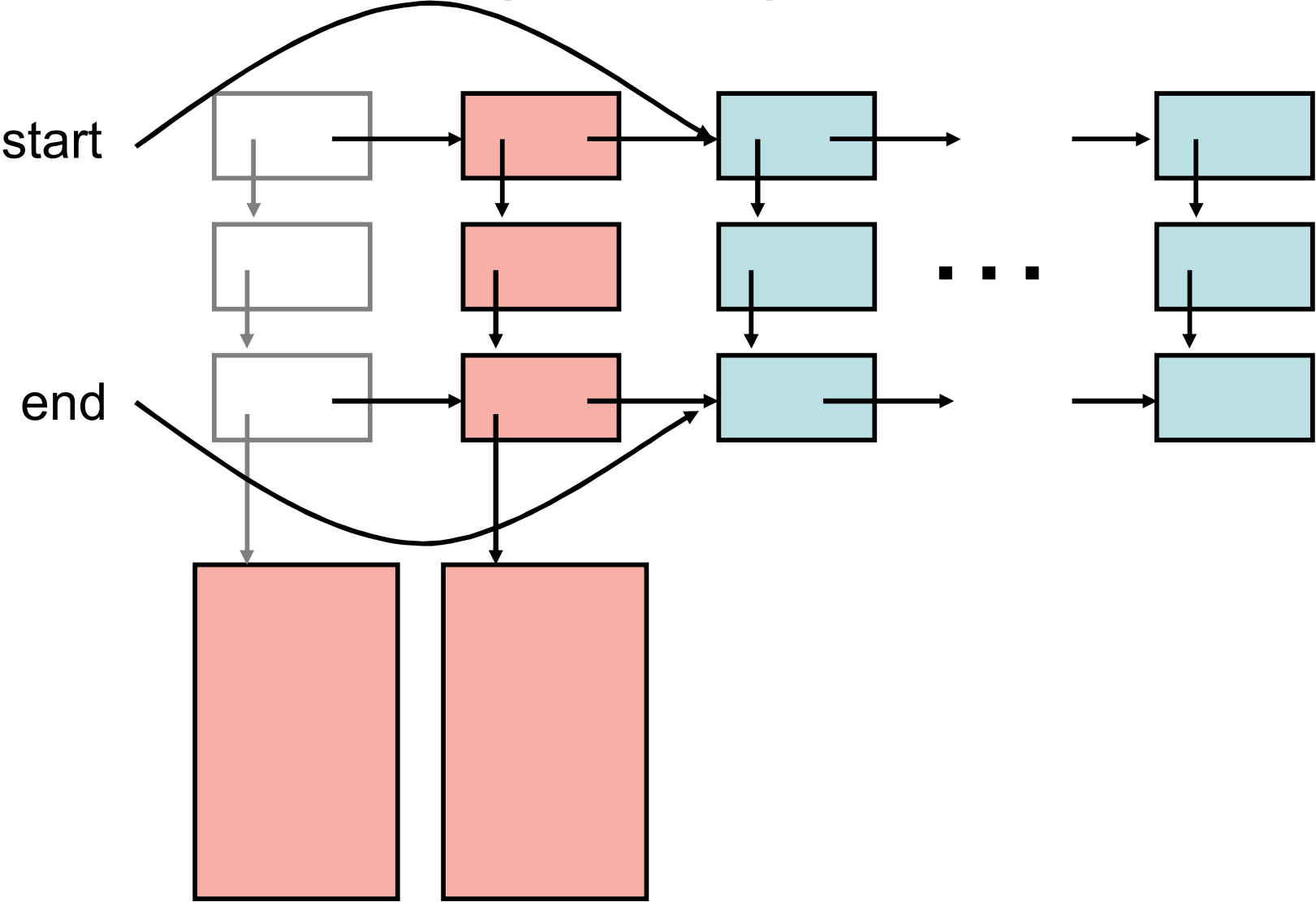


$$\frac{N}{6 \text{ smin}} \text{ columns}$$

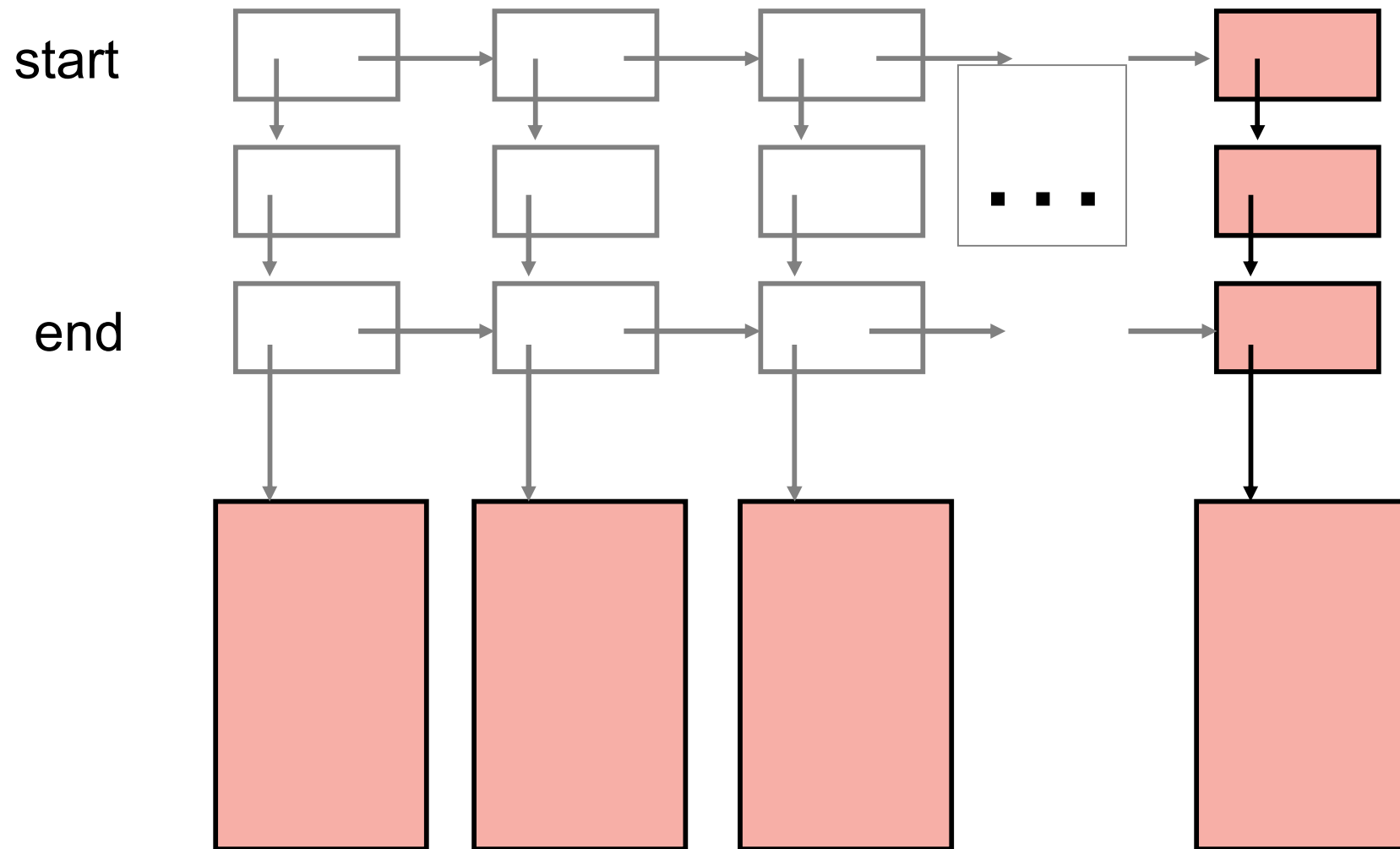
Step 2 – Allocate large object & advance



Step 3 – Repeat



Final State



Conclusions

- In a reference counted system, either:
 - There may be pauses,
 - Allocation takes time proportional to object size (as with tracing), or
 - It incurs a probably unacceptable (though finite) worst-case space overhead.
- The fixed size case is not an anomaly:
 - There is a smooth tradeoff with $s_{\max}/s_{\min}$.