jwmalloc

A Verified Memory Allocator for Mobile Devices

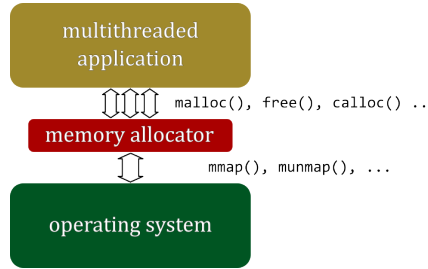
Jiawei Wang[1] Ming Fu[1] Ruixian Wang[1] Chao Xu[1]
Jonas Oberhauser[1,2] Haibo Chen[1,3]

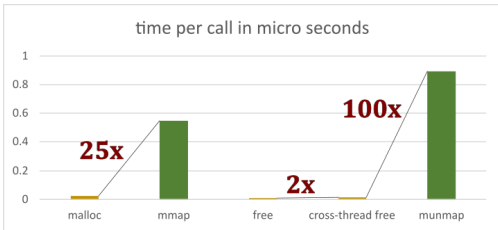
[1] *Huawei Central Software Institute* [2] *Huawei Hilbert Research Center*
[3] *Shanghai Jiao Tong University*

The Role of Memory Allocators

A Crucial Interface in the System Stack

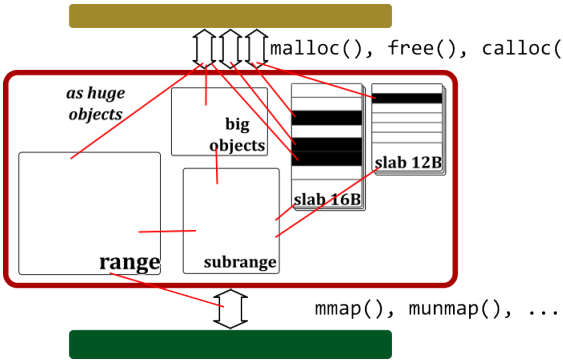
- **Upper Layer: Serving (typically) Multi-Threaded Apps**
  - Provides standard runtime interfaces like `malloc()` and `free()`.
- **Lower Layer: Interacting with the OS**
  - Manages raw memory by utilizing expensive system calls such as `mmap()` and `munmap()`.





Non-scientific measurement on my machine. Allocators hide the latency of OS memory management. Freeing memory allocated by other threads (**cross-thread free**) also has synchronization overheads.

Anatomy of a typical Allocator

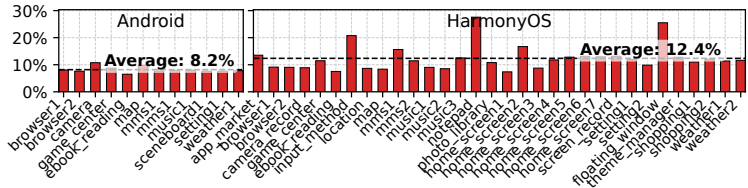


- Allocator gets **ranges** from OS
- splits into **slabs** (object arrays for small to midsized object) or bigger objects
- uses metadata like freelists to recycle objects
- returns ranges (or subranges) to OS when no longer needed

Opportunities in Mobile Workloads

Scale vs. Resource Scarcity

- **Warehouse-Scale Datacenters:**
  - Allocator activity consumes **~7% CPU cycles** [Google] with highly optimized jemalloc.
  - Even a 1% optimization yields significant server cost savings.
- **Mobile Devices – limited DRAM:**
  - DRAM is intensely shared among apps.
  - If memory isn't `munmap`'ed ASAP: **paging** + **background app kills** :(

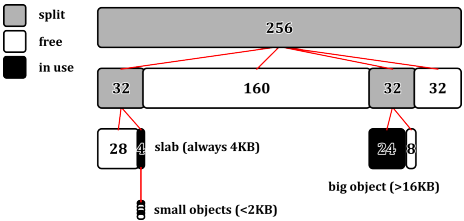


Allocator overhead comparison under real-world mobile workloads.

Many mobile apps frequently `malloc+free`, allocator has to `munmap` ASAP.
Allocator became a primary bottleneck (again)!

jwmalloc: A Mobile-First Allocator

Key differences: **closed sibling tree + generational reclamation**



Closed sibling tree managing a range of 256KB. Ranges are subdivided into 4KB slabs for small objects, larger slabs for mid-sized objects, or used directly as a big object. Huge objects (>4MB) fall pack to `mmap`.

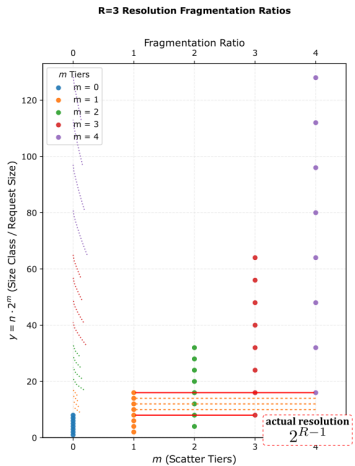
Evaluation Highlights

- **Micro-benchmarks** *(rptest, xmalloc, mstress)*
  - **74%** avg. throughput improvement over jemalloc
  - **82%** reduction in allocator-side instructions
  - **1.5 μs** P99.99 latency vs. 5.9 μs for best competitor
- **Macro-benchmarks** *(Mate 70 Pro, HarmonyOS 5.1)*
  - **3.84$\times$ to 4.79$\times$** lower user-space allocation instructions
  - **10% to 13%** lower overall device instruction overhead
  - **5% to 11%** CPU-cluster power reduction

New Concept: Allocator Resolution

R-Resolution: allocator can manage every size of shape $n \times 2^m$ where $1 \leq n \leq 2^R$.

wasted memory factor (**fragmentation ratio**): $\leq 1/2^{R-1}$



$x = \underbrace{011}_{n} 0000000000000000....$ (R, m)

$x = \underbrace{110}_{n'} 0000000000000000....$ (R, m')

$y = 1000\,0000000000000000....$ (R+1)

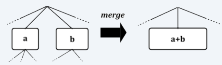
special case: $n = 2^R$

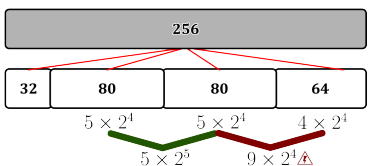
- Worst case $R = 1$: support only powers of two (1,2,4,8,...), up to 1x wasted memory (17KB round up to 32KB)
- Typical cases: buddy (R=1), extended buddy variants ($R = 2$), modern allocators ($R = 3$).
- Closed Sibling Tree: first data structure where R can be configured arbitrarily

Design of Closed Sibling Tree (CST)

In a closed sibling tree (of resolution R),

- each node has a R-resolution size,
- merging any adjacent siblings results again in a closed sibling tree





- Keep number K of node-sizes as small as possible

$$C_1, \ldots, C_K$$

- Efficient linear search in bitmap:
  - `freenodes[i]` if any free nodes of i-th size class C_i
  - To get node of size $X \geq C_i$, find $j = i, i{+}1, \ldots, K$

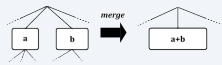
$$\texttt{freenodes}[j] = 1$$

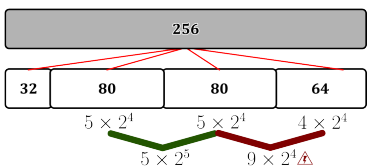
- Large $K \implies$ better switch to binary tree etc.

Design of Closed Sibling Tree (CST)

In a closed sibling tree (of resolution R),

- each node has a R-resolution size,
- merging any adjacent siblings results again in a closed sibling tree





- Keep number K of node-sizes as small as possible

$$C_1, \ldots, C_K$$

- Efficient linear search in bitmap:
  - `freenodes[i]` if any free nodes of i-th size class C_i
  - To get node of size $X \geq C_i$, find $j = i, i{+}1, \ldots, K$

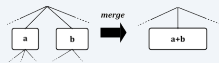
$$\texttt{freenodes}[j] = 1$$

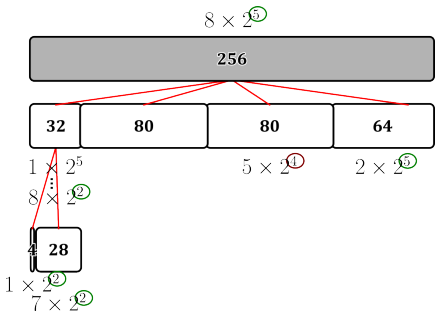
- Large $K \implies$ better switch to binary tree etc.

Design of Closed Sibling Tree (CST)

In a closed sibling tree (of resolution R),

- each node has a R-resolution size,
- merging any adjacent siblings results again in a closed sibling tree





- Key invariant of our algorithm: Parent + Children can be represented by same m !

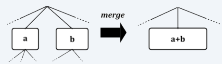
$$p = n_p \times 2^m, \quad c_i = n_i \times 2^m, \quad n_p,\, n_i \in [1 : 2^R]$$

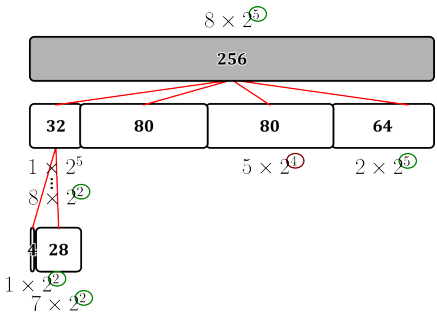
2^m is like a *unit* of those children

Design of Closed Sibling Tree (CST)

In a closed sibling tree (of resolution R),

- each node has a R-resolution size,
- merging any adjacent siblings results again in a closed sibling tree



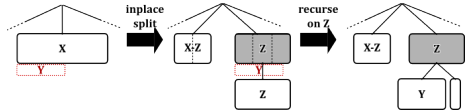


- Key invariant of our algorithm: Parent + Children can be represented by same m !

$$p = n_p \times 2^m, \quad c_i = n_i \times 2^m, \quad n_p,\ n_i \in [1 : 2^R]$$

2^m is like a *unit* of those children

- GETRANGE(Y): find node $X \geq Y$, split to next allowed value $Z \geq Y$, recurse on Z



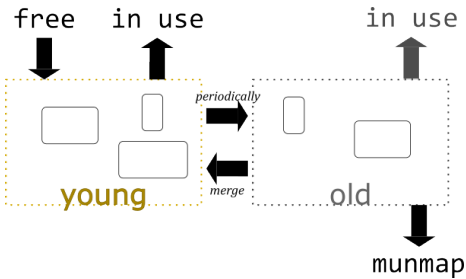
note: algorithm in paper is wrong, violates the invariant. Fix: compute m_i (unit) from *parent* of X

Generational Reclamation

Generational Hypothesis from **Garbage Collection**: **objects** that have lived for a while will continue to live for a long time

- Allocators prefer to merge adjacent freed ranges to do a single batch `munmap`
- Long-lived objects pin adjacent range, prevent mergin
- Question: `munmap` **now** to release memory vs. **wait** for adjacent range to be freed?

Our new heuristic:



- Freed nodes in CST are not unmapped immediately
- Put in "young pool". Search nodes in young first.
- Switch young & old periodically, `unmap` everything that stayed in old

Lock-Based Lockfree Programming

Lockfree: Every operation can always conclude, even if some other threads (e.g. lock holders) are not scheduled

CST nodes & other metadata are protected by locks, but:

- **if fail to acquire a lock on fast path**: **back off** and go to **slower/less optimal path**. e.g., free a node:
  1. acquire sibling lock and merge; if fail, just leave unmerged
  2. insert into concurrent bitmap; if fail:
  3. instead, insert into locked linked list; if fail:
  4. instead, insert into lock-free set

any half-done work will be cleaned up later by background thread...


```
25 acq_sibling(r) {
26   r' := r.sibling;
27   if (!knit_acq(r'))
28     return null;
29   if (nest_acq(r'))
30     return r';
31   knit_rel(r');
32   return null;
33 }
```

- only final `mmap` etc. may be blocking (kernel-side)

Verification — How to Increase Confidence?

- Hard to debug, super critical piece of software
- Full formal verification still too expensive
- How to get “biggest bang for the buck”?

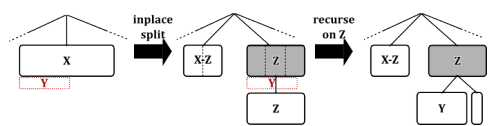
CST Algorithm	CST Impl.	Synchronization	Slab Allocator
Small, conceptual code. Relation to CST property not obvious	Complex, big code. CST represented as bitmap. Protected by lock.	Lock-free algorithms, complex back-off protocols spread over several modules	Fragile pointer arithmetic, alignment invariants
Design invariant, prove it implies CST property	Prove refinement relation to abstract CST, well-formedness	Modularize, write module level tests, explore all concurrent executions	Verify pointer arithmetic functions, alignment
ROCQ	frama C Software Analyzers	GenMC	frama C Software Analyzers

Mathematical Arguments: Theorem Prover

Prove: our invariant implies CST property, termination (500 LOC, roughly 2 days of work, 3 bugs)

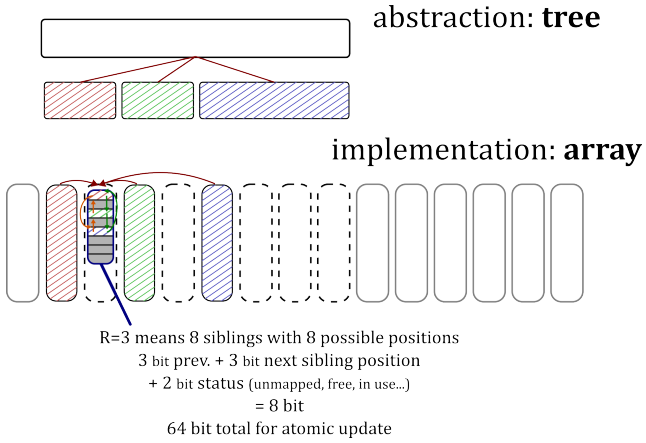
```
Definition valid_f x m := ∃ n, x = n * 2^m ∧ 1 ≤ n ≤ 2^R.

Lemma invariant_correct p c1 c2 m :
  valid_f p m →
  valid_f c1 m →
  valid_f c2 m →
  c1+c2 ≤ p →
    valid_f (c1+c2) m.

Lemma invariant_local p c1 c2 :
  (∃ m, valid_f p m ∧ valid_f c1 m) →
  (∃ m, valid_f p m ∧ valid_f c2 m) →
  c1+c2 ≤ p →
     (∃ m, valid_f p m ∧ valid_f c1 m  ∧ valid_f c2 m).
```

```
Lemma split_correct m x y :  R > 0 →
  valid_f x m →
  valid y →
  y < x →
  let z := ((y - 1) / 2^m + 1) * 2^m in
    valid_f z m
  ∧ (x ≠ z → valid_f (x - z) m)
  ∧ y ≤ z
  ∧ (y < z → m > 0 ∧ valid_f z (m-1)).
```



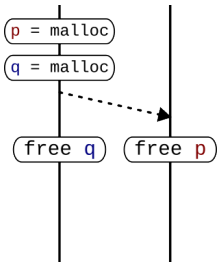
Tricky Code: Sequential deductive verification with Frama-C

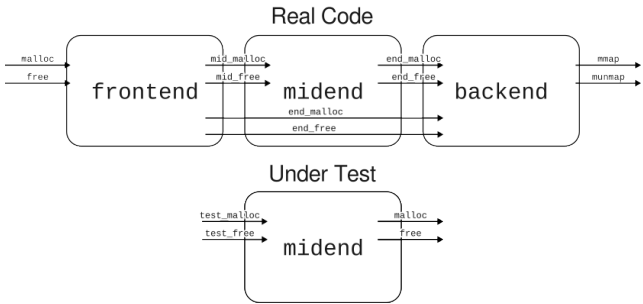
Prove: pointer arithmetic, pointer alignment, tree abstraction (Several KLoc, 10 person months, 3 bugs. Conclusion: merge verification code or it dies. Tooling need more love!)



Concurrency: Test-based model checking with GenMC

Prove: Specific racy scenarios **never** fail (Several KLoc, 6 person months, 1 bug.)





- Model Checker exhaustively explores all possible orders of race conditions in the test
- Challenge: find test cases that stress the code but not the Model Checker

And then...

- Deploy jwmalloc to real APP $\Longrightarrow$ ...

And then...

- Deploy jwmalloc to real APP $\Longrightarrow$ ...crash!

And then...

- Deploy jwmalloc to real APP $\Longrightarrow$ ...crash!
- Hyrum's Law:

  "all observable behaviors of your system will be depended on by somebody."

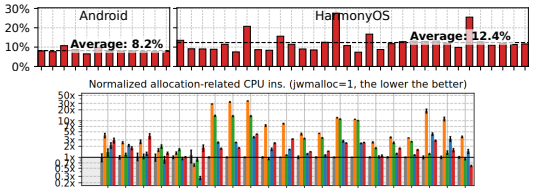
  — Hyrum Wright

Real APPs are buggy – rely on UB!

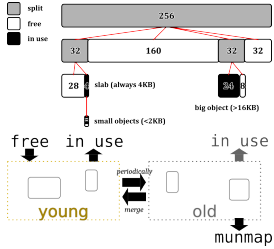
```
T *p = malloc(...);
...
foo(p[0]); // access uninitialized memory
           // -- happens to be 0 with jemalloc
...
```

jwmalloc stores metadata there...

Summary



mobile vs server opportunity



CST + generational reclamation

ROCQ frama C Software Analyzers GenMC

Formal Methods “sprinkled on top” is practical, finds bugs on top of n*1000h stress test

```
T *p = malloc(...);
...
foo(p[0]); // access uninitialized memory
           // -- happens to be 0 with jemalloc
...
```

Formal verification against official specs not enough to ensure smooth deployment!