UKSM: Swift Memory Deduplication via Hierarchical and Adaptive Memory Region Distilling

Nai Xia†, Chen Tian†, Yan Luo‡, Hang Liu‡, Xiaoliang Wang†

† Nanjing University, China

‡ University of Massachusetts Lowell, USA

USENIX FAST'18

Background

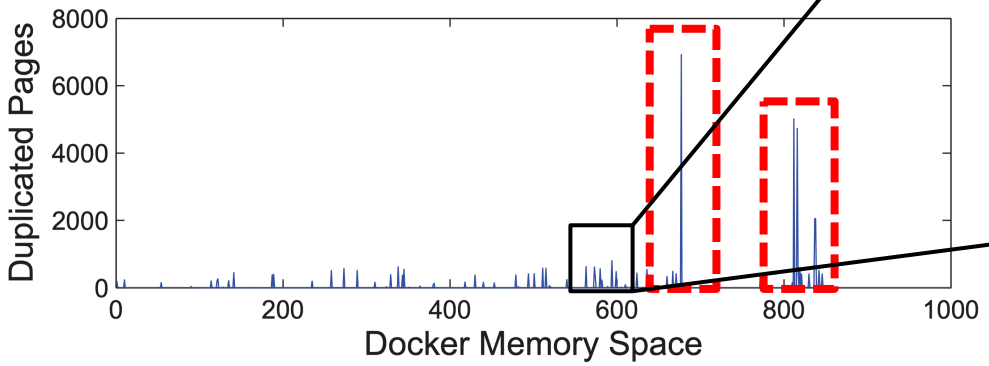
- **Content Based Page Sharing (CBPS)**
  - Full memory scan to find duplicated pages
  - Page hash comparison -> byte-to-byte comparison -> page merging
  - Copy-on-write
  - Kernel Same-page Sharing (KSM): unstable tree and stable tree
- **Four duplication patterns**
  - Sparse, COW-broken, short-lived
  - Statically-duplicated (ideal for deduplication)

Background & Problem

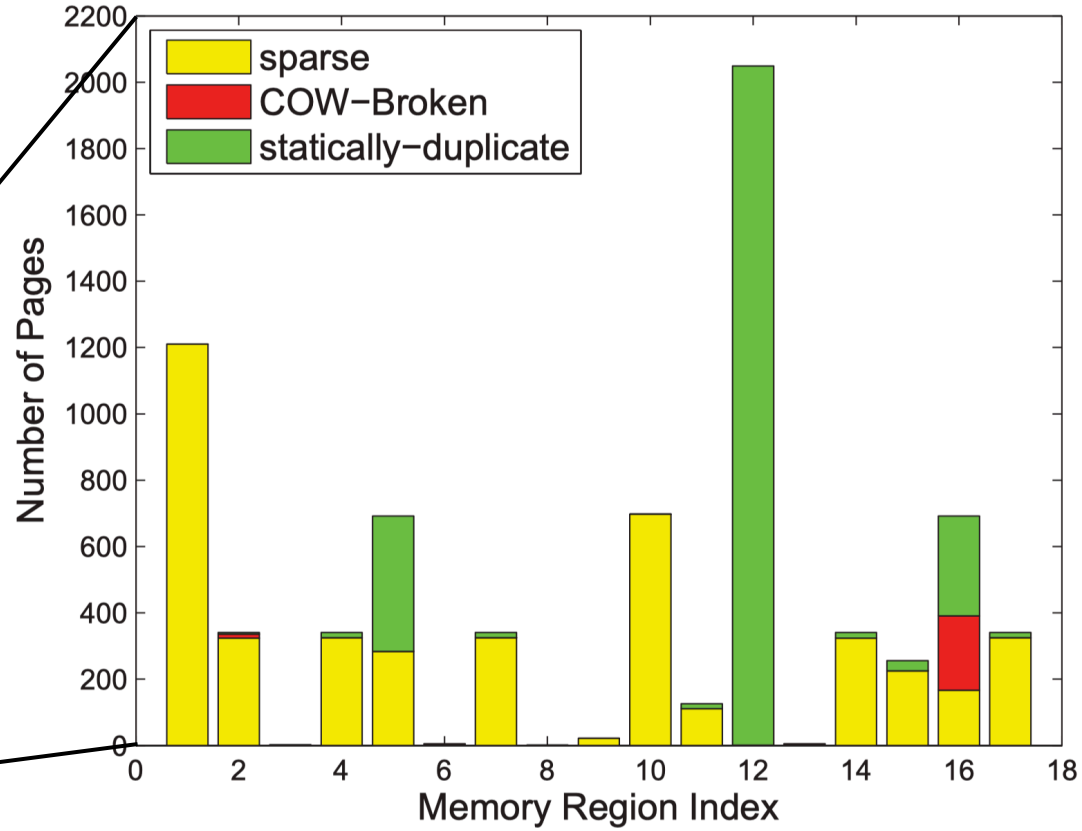
- CBPS: KSM with *ksmtuned*
  - Control page scanning speed just by memory usage
  - Coarse-grained system info, treating every page equally
- I/O hints BPS: Cross Layer I/O-based Hints (XLH)
  - Find sharing chance using I/O hints in host's virtual filesystem
  - Ignore dynamically created duplicated pages (in memory)
- Existing Storage Deduplication
  - No necessary need of dynamic characteristics and fast deduplication system

Motivation

- Pages in the same region present similar deduplication patterns
- Partial page hashing need to be adaptive: time-consuming in page hashing vs. byte-to-byte page comparison (collision)



(a) Number of duplicated pages



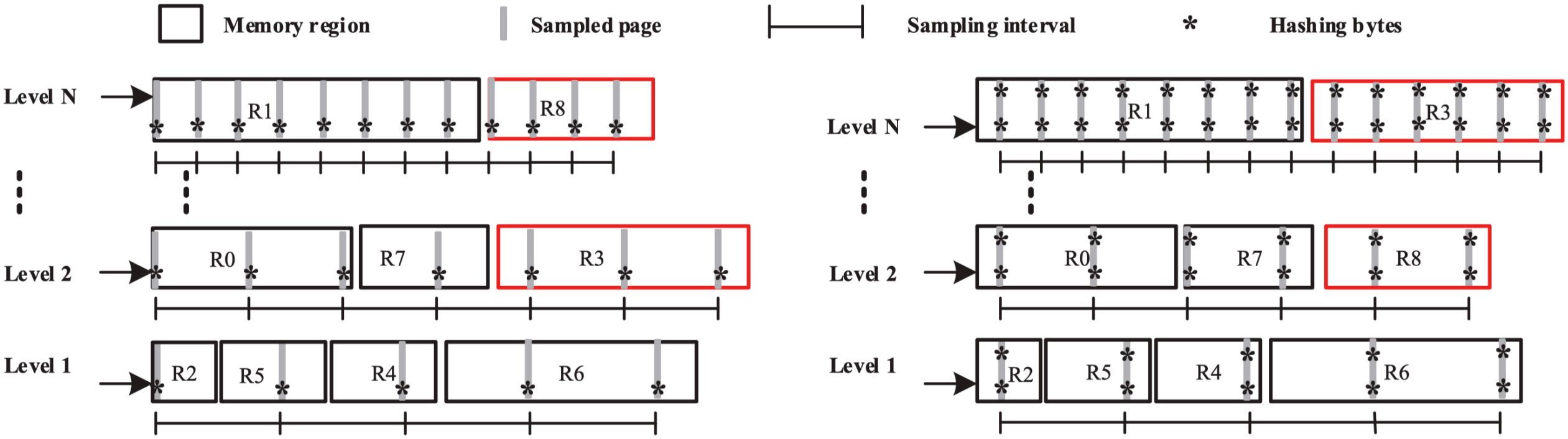
(b) A snapshot of Docker memory fragment

Contribution

Ultra KSM

- Hierarchical Region Distilling
  - Candidate region identification
  - Hierarchical sampling procedure
- Adaptive Partial Hashing
  - Hash strength adaptation
  - Progressive hash algorithm

Design Overview



(a) round 1 with a smaller hash strength **(b) round 10 with a larger hash strength**

Figure 2: Memory region hierarchical distilling in two sampling rounds with different hash strength.

Candidate Region Identification

- 3 threshold values
  - $if\,(dup\_ratio > V_{dup}\ \&\&\ COW\_ratio < V_{COW}\ \&\&\ lifetime > V_{life})$
  - $\quad promote(region);$
  - $else$
  - $\quad demote(region);$
- Duplication ratio: duplicated pages / pages sampled
- COW ratio: COW-broken page faults / pages sampled

Hierarchical Sampling Procedure

- Pick sample points by the length of *interval*
- Random offset of sample points
- Get page partial hash and look it up in two red-black tree
  - Read-only merged(stable) tree
  - Unmerged(Unstable) tree
- Sample from level 1 to N, calculate COW ratio and Duplication ratio

$$interval = \frac{L_{level}}{sample\_points} = \frac{L_{level}}{t_{level} * (\frac{p}{s})}$$

Hash Strength Adaptation

- $\max benefit = \max\ (profit - penalty)$
  - Time-consuming in page hashing vs. byte-to-byte page comparison
- "Probing" state
  - Simulate TCP slow start, *delta* = 1, Hash strength = strong hash strength / 2
  - Hash strength -= *delta* until benefit stop increasing (can also +=)
  - *delta* *= 2 each time until it reach to 32
- "Stable" state
  - Change to probing need to wait 1000 sampling rounds
  - or no byte-to-byte comparison by hash collision in last 2 sampling rounds
  - or benefit decrease 50%+ in last sampling round.

Progressive Hash Algorithm

- Random offset to sampling bytes in each sample page
- one-at-a-time hash
  - Ancestor of SuperFastHash
- Progressive hash and reverse function

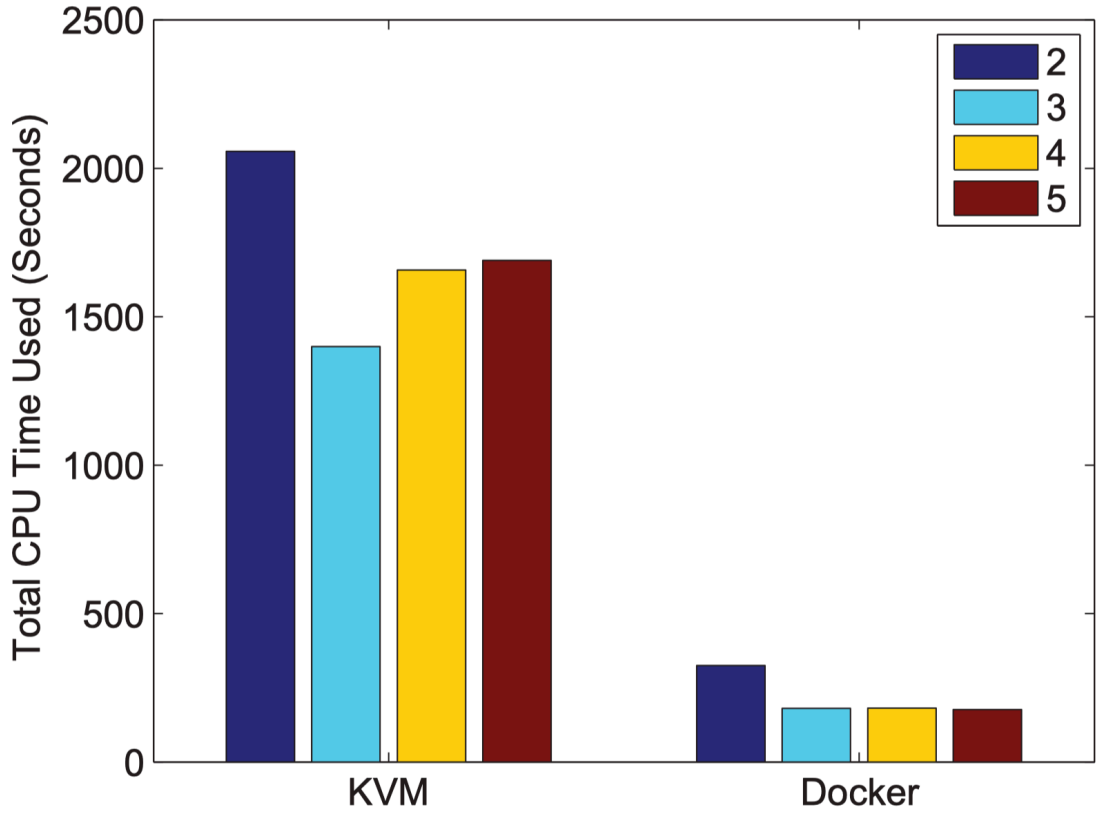
```
#define STREN_FULL (4096/sizeof(u32))
u32 shiftr, shiftl;
u32 random_offsets[STREN_FULL];
u32 random_sample_hash(u32 hash_init,
    void *page_addr, u32 strength) {
    u32 hash = hash_init;
    u32 i, pos, loop = strength;
    u32 *key = (u32*)page_addr;

    if (strength > STREN_FULL)
        loop = STREN_FULL;
    for (i = 0; i < loop; i++) {
        pos = random_offsets[i];
        hash += key[pos];
        hash += (hash << shiftl);
        hash ^= (hash >> shiftr);
    }
    return hash;
}
```

Figure 5: Progressive hash procedure.

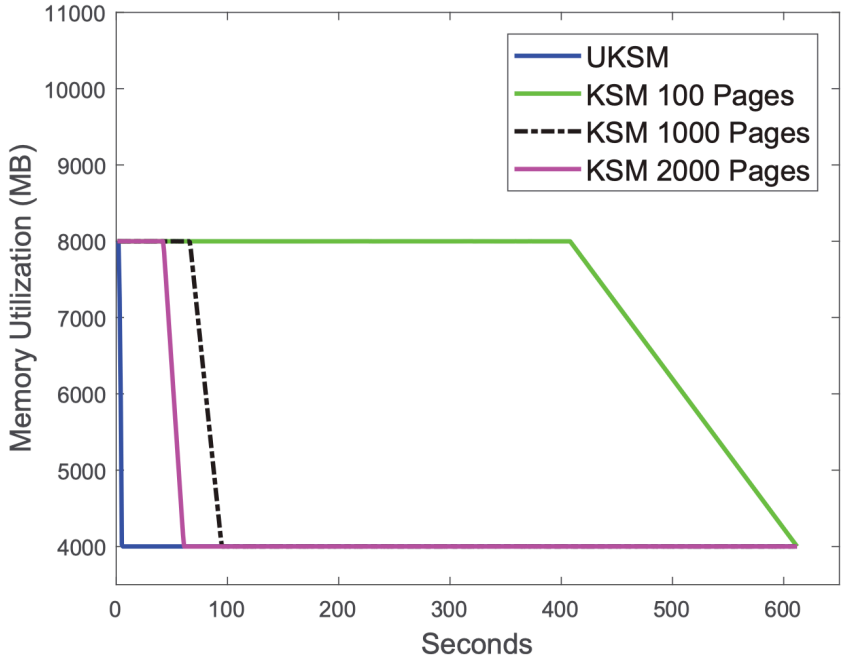
Evaluation

- Configuration
  - *Full* governor for default (responsive)
  - Sampling: 5 levels for KVM, 3 levels for Docker, 4 levels for default
- Benchmark
  - CentOS 7 (with Linux Kernel 4.4)
  - Intel(R) Core(TM) i7 CPU 920 with four 2.67GHz cores
  - 12 GB RAM
  - KSM usable (SuperFastHash)

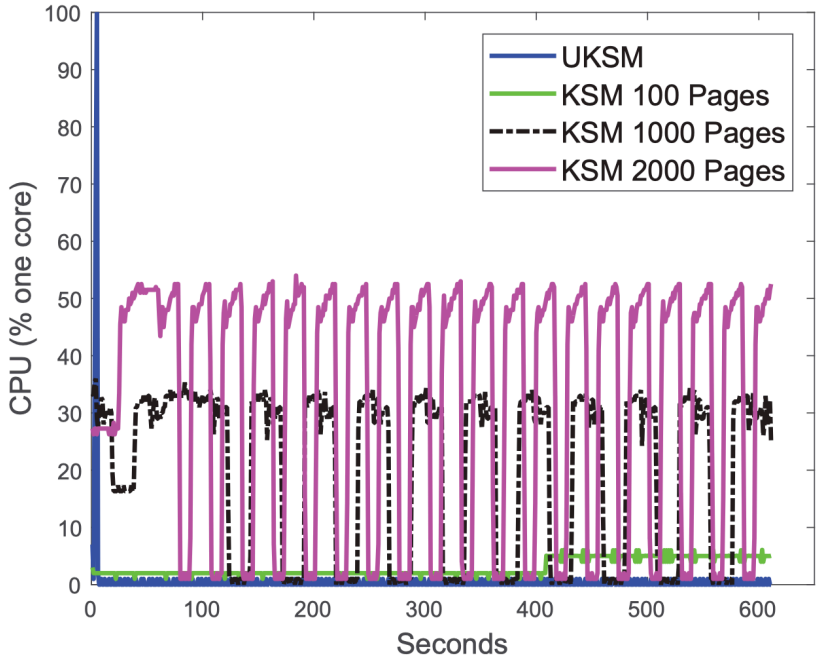


(c) Performance with different level numbers

Evaluation: Statically Mixed Workload



(a) Deduplication speed and memory saving

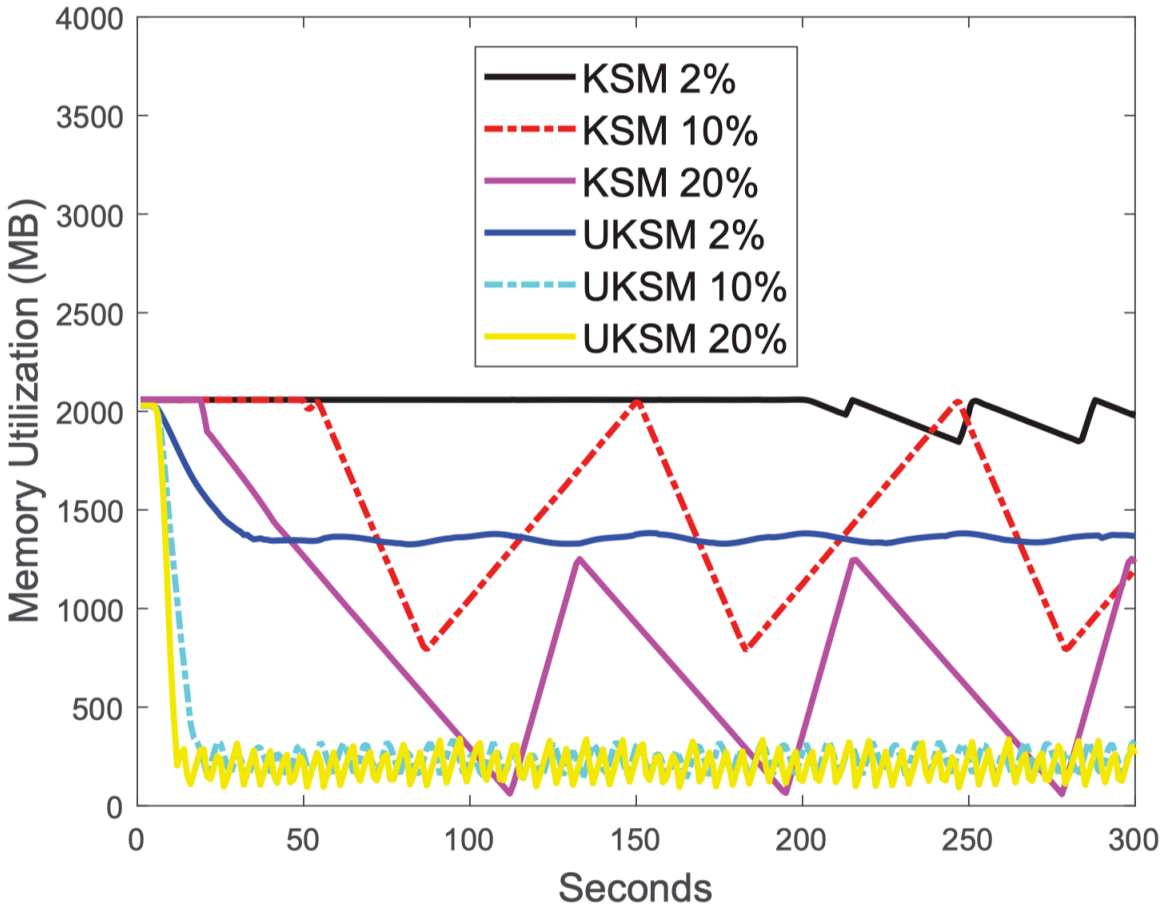


(b) CPU consumption

- Take only 5 seconds to merge all duplicated pages for UKSM
- Low background CPU consumption
- 8.3X/12.6X/11.5X duplication efficiency than KSM

Evaluation

- For COW-broken (each 10ms)
  - Stable performance
  - 3X~5X memory saving of KSM
- For short-lived
  - UKSM can merge pages when lifetime is less than 200ms (KSM can merge only when lifetime is higher than 2s)



(c) Memory Utilization for COW-broken

Evaluation

- Real world benchmark
  - KVM: 3X memory saving of KSM in 25 VMs
  - UKSM requires no swap usage in booting 40VMs with memory overcommit
  - Docker, desktop server, general deduplication system
- Hash Function Speed
  - In the worst case, can comparable to SuperFastHash
  - In ideal case (no same page), 5.9X higher than hash-based KSM and 7.4X higher than original KSM
  - In ideal case (all same page), 2.5X higher than hash-based KSM and 7X higher than original KSM

Conclusion

- **UKSM**: memory deduplication in different levels of scanning and sampling area with fine-grained dynamic characteristics via:
  - Hierarchical Region Distilling
  - Adaptive Partial Hashing

Comments