

Concurrent Remembered Set Refinement in Generational Garbage Collection

David Detlefs

Ross Knippel

Sun

Microsystems

William D.
Clinger

Northeastern

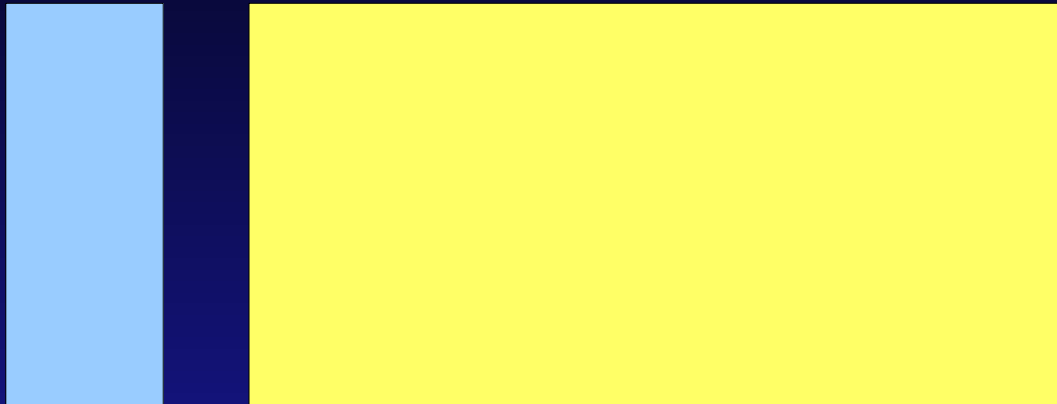
Matthias Jacob

Princeton

Introduction

- ☞ *Generational* collection can decrease GC latency, increase throughput...
- ☞ ...but requires a "tax" in the form of a *write barrier* to maintain a cross-generation *remembered set*.
- ☞ Write barriers usually present an *efficiency/precision* tradeoff...
- ☞ ...unless we can make some remembered set maintenance *concurrent*.

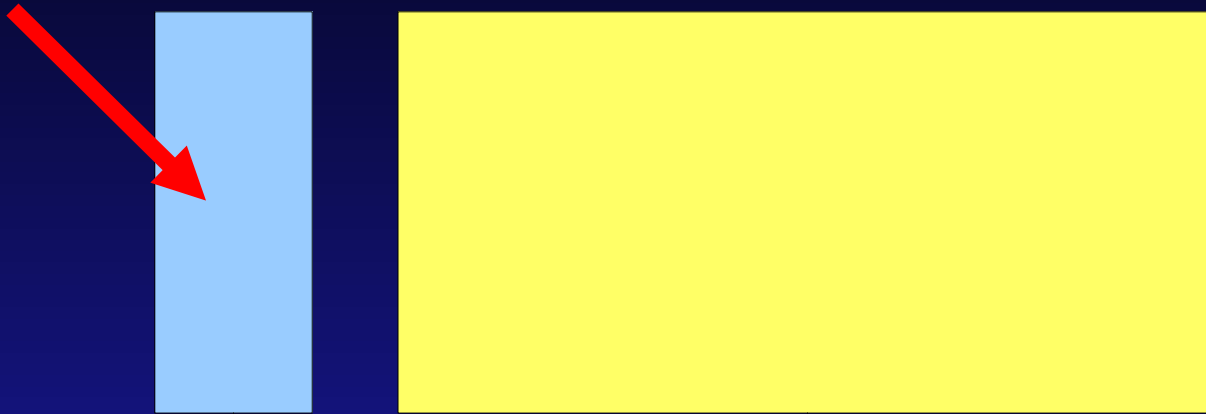
Generational Collection



- Reduces GC pauses (i.e., latency).
- (Generally) increases throughput.

Generational Collection

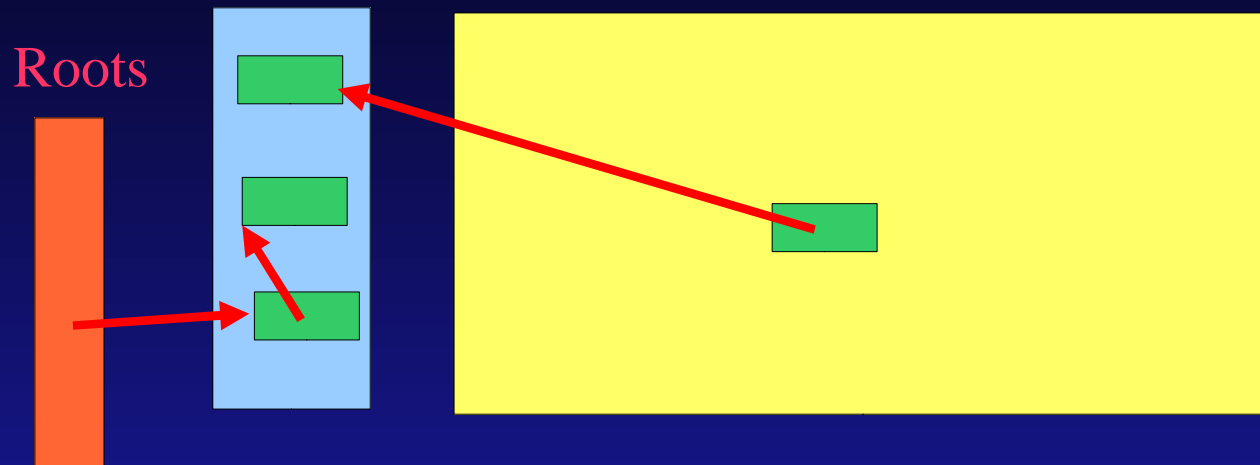
Allocation



- ➡ Reduces GC pauses (i.e., latency).
- ➡ (Generally) increases throughput.

Generational Collection

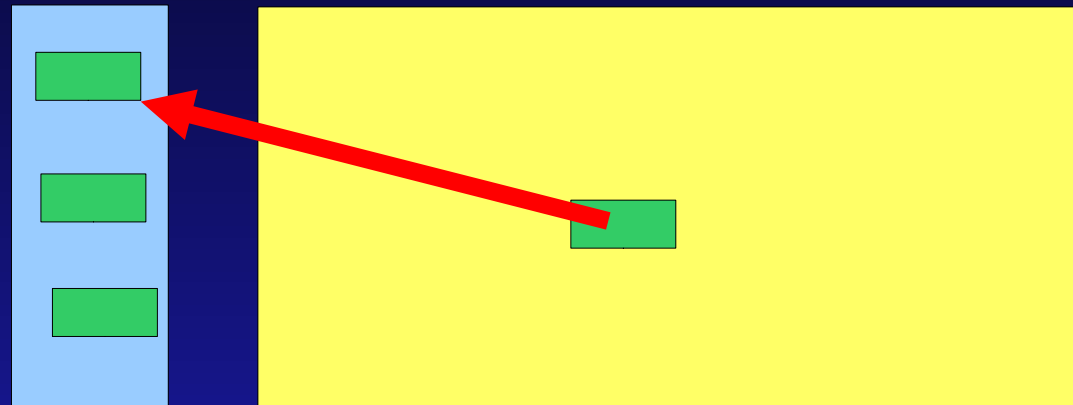
Young-gen collection



- ➡ Reduces GC pauses (i.e., latency).
- ➡ (Generally) increases throughput.

Remembered Set: Definition

- ☞ Data structure supporting iteration over "old-to-young" pointers.

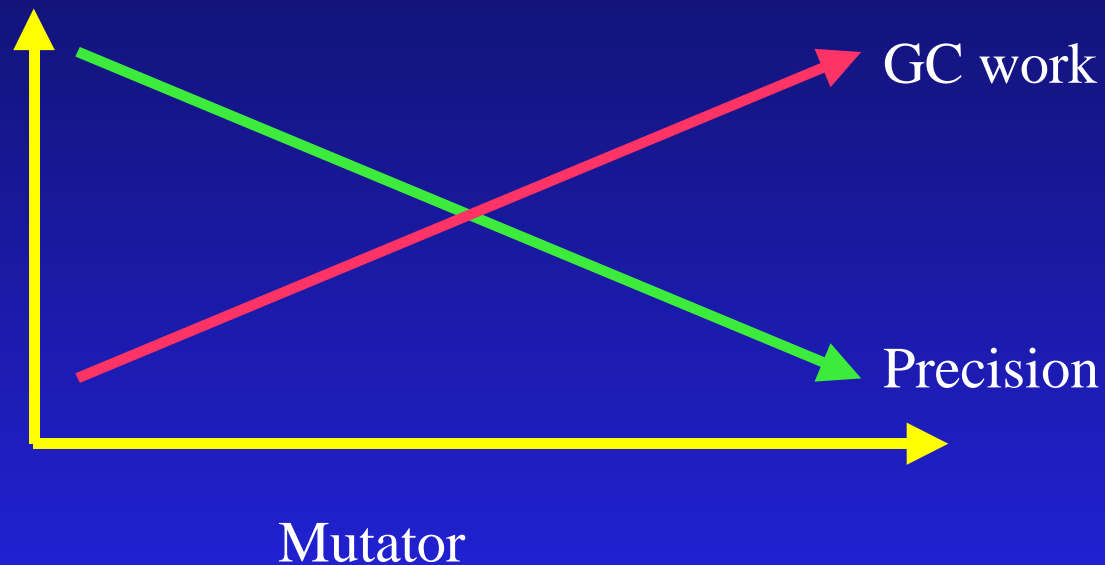


- ☞ Two parts:

- ☞ Tracking newly created cross-gen pointers (*write barrier*).
- ☞ Remembering ones left after previous collections (for configurations that allow this).

Remembered Sets: remarks

- ☞ This is the part of young-gen collection whose cost is *not* bounded by young-gen size.
- ☞ For newly-created pointers, *efficiency/precision* tradeoff:

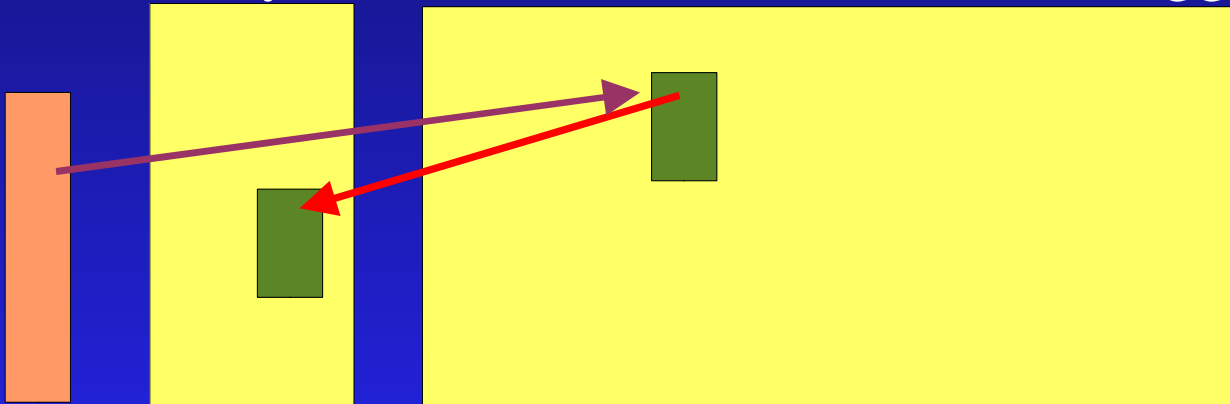


Common Remembered Sets: Sequential Store Buffers

e.g., [Hosking, Moss, Stefanovic 92]

☞ Sequential Store Buffer.

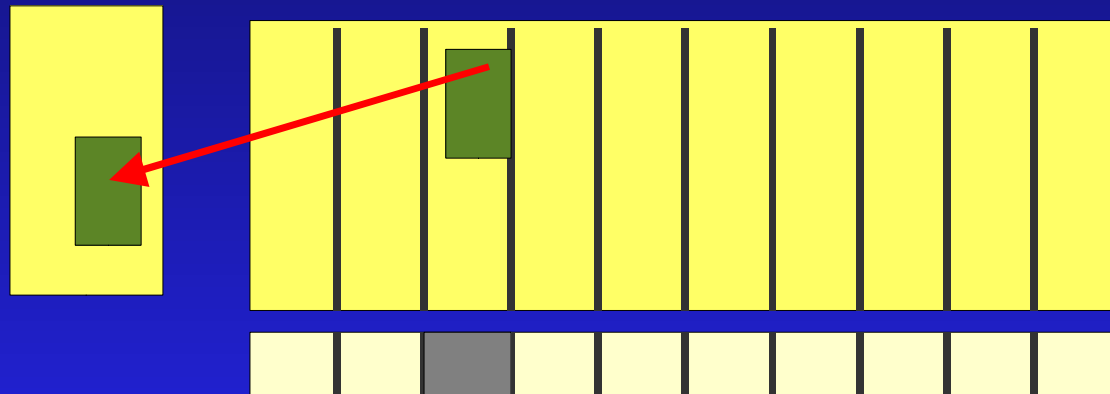
- ☞ (Thread-local) buffer storing the addresses of fields cross-generational pointers are stored into.
 - ☞ Objects vs. fields.
 - ☞ With and without *filtering* of intra-generation pointers.
 - ☞ With and without duplicate detection.
- ☞ Usually: fixed size buffers; overflow triggers GC.



Common Remembered Sets: Card Marking

e.g. [Holzle 93]

- Divide old-gen into equal-sized *cards*.
- *Card table* has entry per card, initially *clean*.
- Mutator pointer update writes *dirty* to corresponding card table entry.
- As few as 2 extra instructions per pointer write.

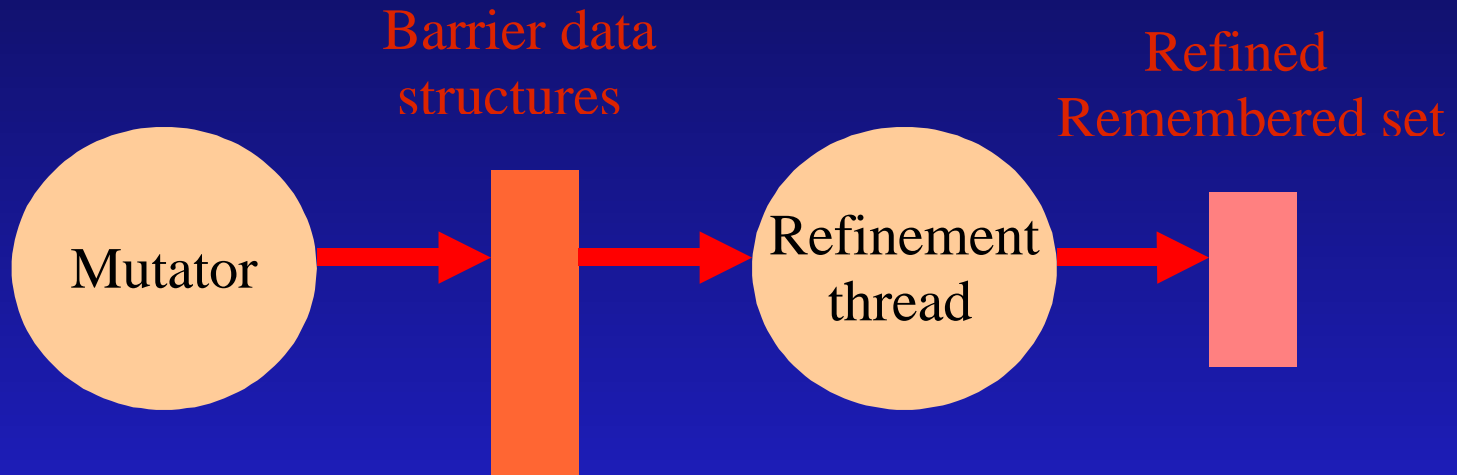


Remembered Set Tradeoff for these Barriers

- ☞ Card marking is efficient, but imprecise:
 - ☞ Spatial approximation
 - ☞ false positives.
- ☞ Store buffers may be precise, but precision impacts efficiency.
 - ☞ cross-gen filtering, duplicate checks
 - ☞ plus buffer overflow checks

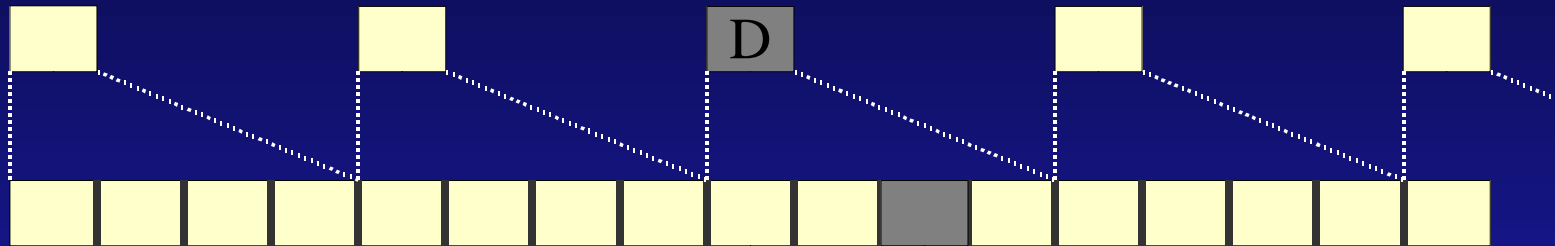
Concurrent Refinement

- ☞ Use an efficient, but imprecise, barrier.
- ☞ Do the checking that provides precision in a concurrent thread.



Remembered Set Designs: 2-level Card Table

e.g., [Sobalvarro 88]



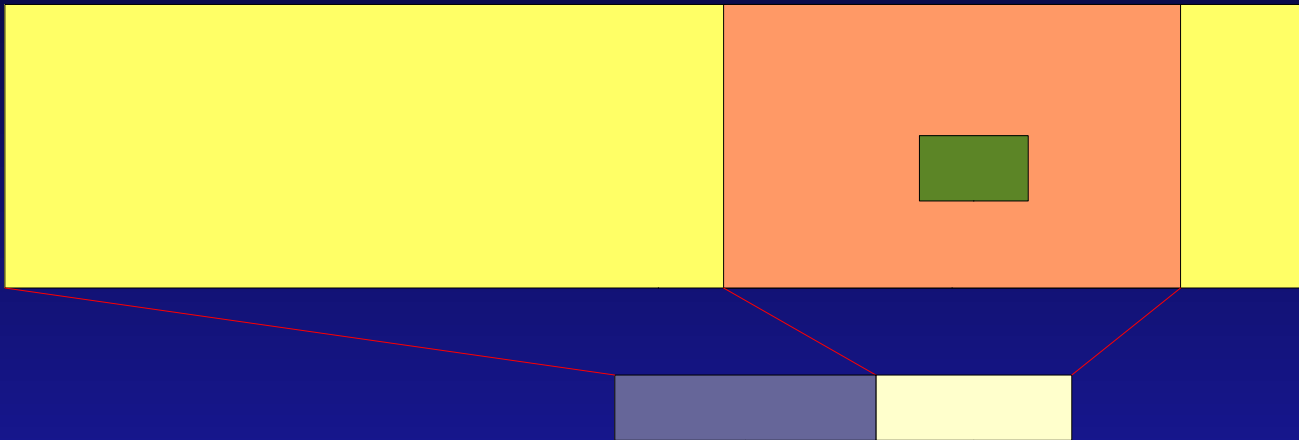
Remembered Set Designs: Hash Table

☞ Didn't do, but could have :-)

Barrier Designs: Card Table

☞ Directly update Card-table RS with *dirty*.

☞ "2 + ϵ " instructions



$x.f := y$

\implies

$reg_base := base_address$

$reg_f := reg_x + f_offset$

$reg_tmp := reg_f \gg log_card_size$

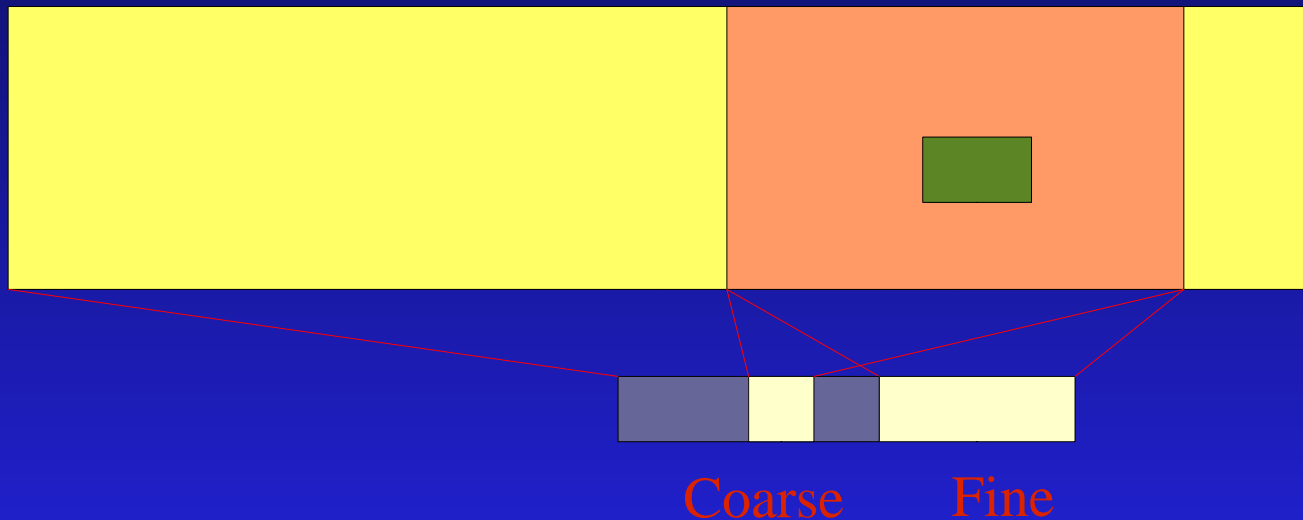
$M[reg_base + reg_tmp] := dirty$

Can be hoisted

or just use reg_x
(cf. Holzle)

2-Level Card Table Barrier

- ➡ Write dirty directly to both card table levels.
- ➡ Trick: one *base_address*.
- ➡ $5+\epsilon$ instructions (2 memory writes).



Concurrent Card-Table Refinement

- Mutator barrier updates pointer, then dirties corresponding card. (Order matters!)
- *Refinement thread* scans card table. For a dirty card i :
 - Set $cardTable[i] := refining$
 - Scan card i ; result is *clean*, *summary*, or *overflow*.
 - $CAS(cardTable[i], refining, result)...$
 - ...because concurrent mutator update may have invalidated scan. (Writes *dirty*.)

2-Level Card Table Refinement

☞ Do the same trick twice.

☞ Refinement thread: for each *dirty* coarse card:

☞ $coarse[card] := refining$

☞ refine each fine-grain card

☞ If all results are clean, $CAS(coarse[card], refining, clean)$

☞ Mutator (order matters!):

☞ Ptr write

☞ $fine[card] := dirty$

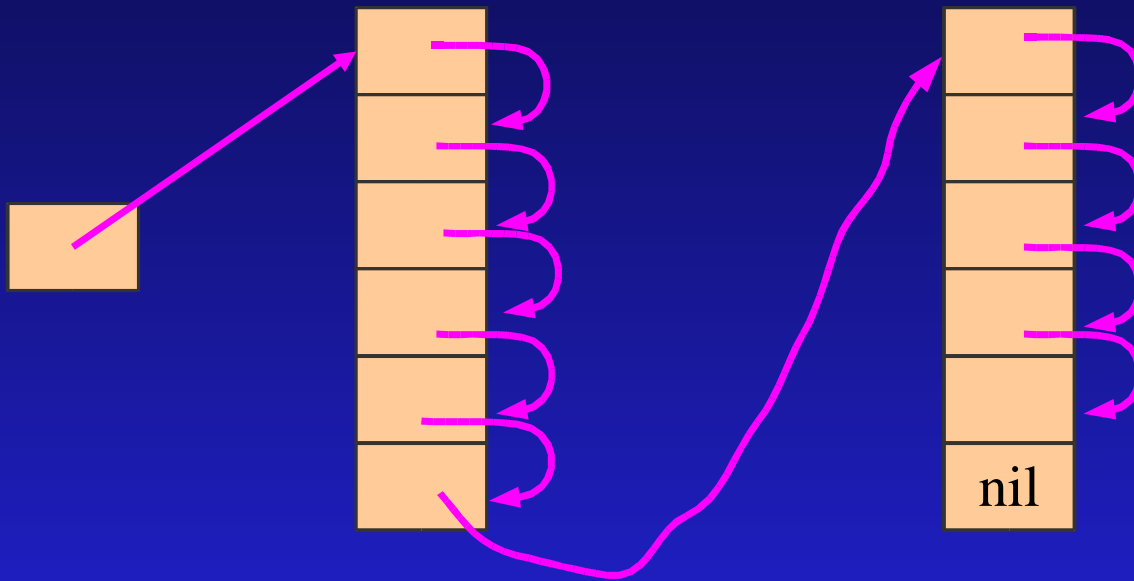
☞ $coarse[card] := dirty$

Log-based barriers (2 kinds)

- In JIT compiler, reserve a register to indicate the *next_log_entry* to write to.
- Barrier:
 - writes address of modified field into *next_log_entry*
 - updates *next_log_entry*
 - Write may cause *overflow*; extend log somehow.

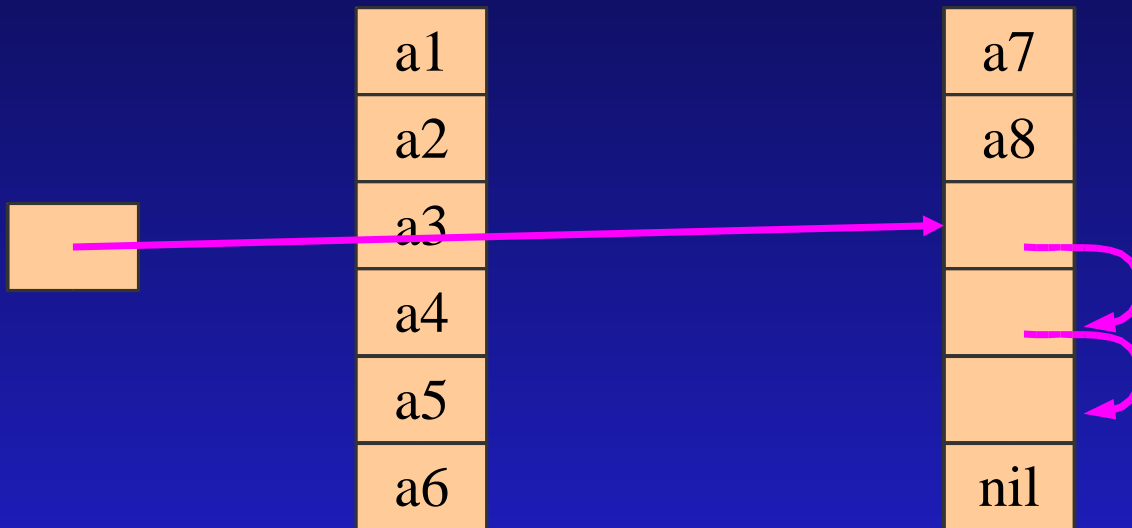
Self-Pointing Log Barrier

- ➡ Log entries initialized to point to next entry.



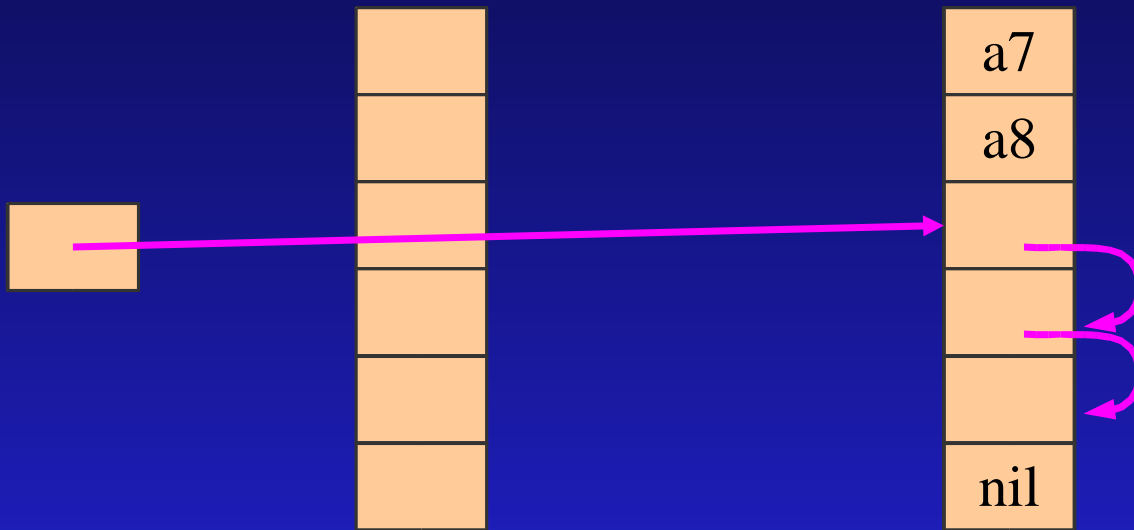
Self-Pointing Log Barrier

- ➡ Log entries initialized to point to next entry.



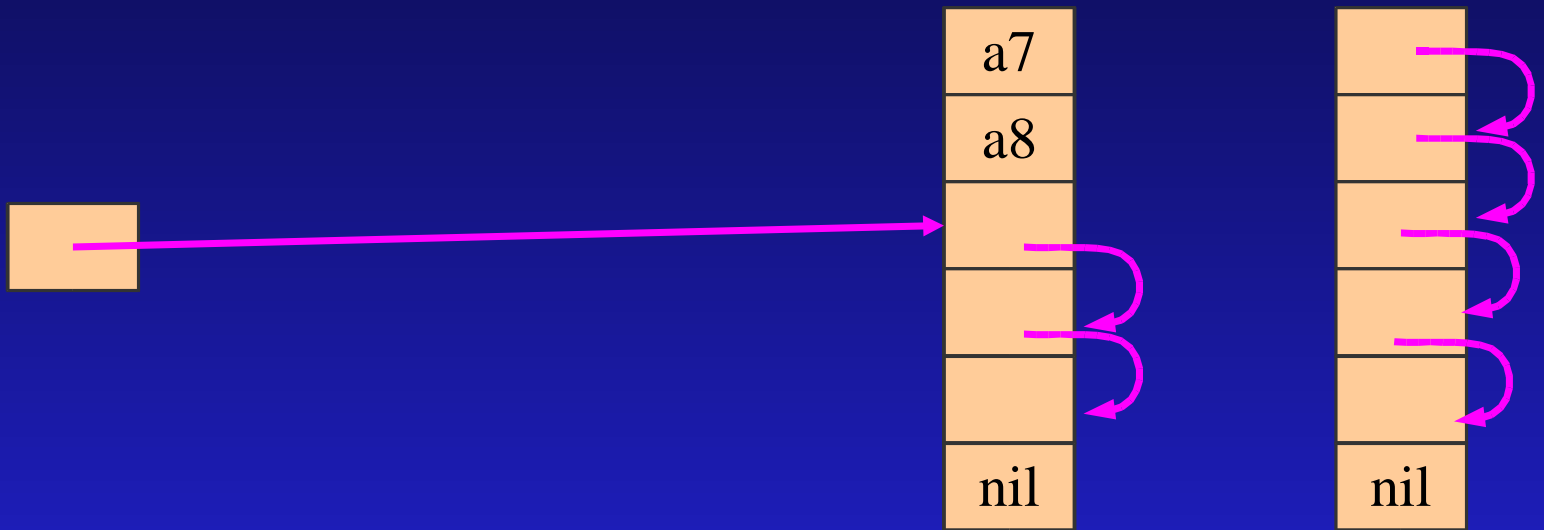
Self-Pointing Log Barrier

- ➡ Log entries initialized to point to next entry.



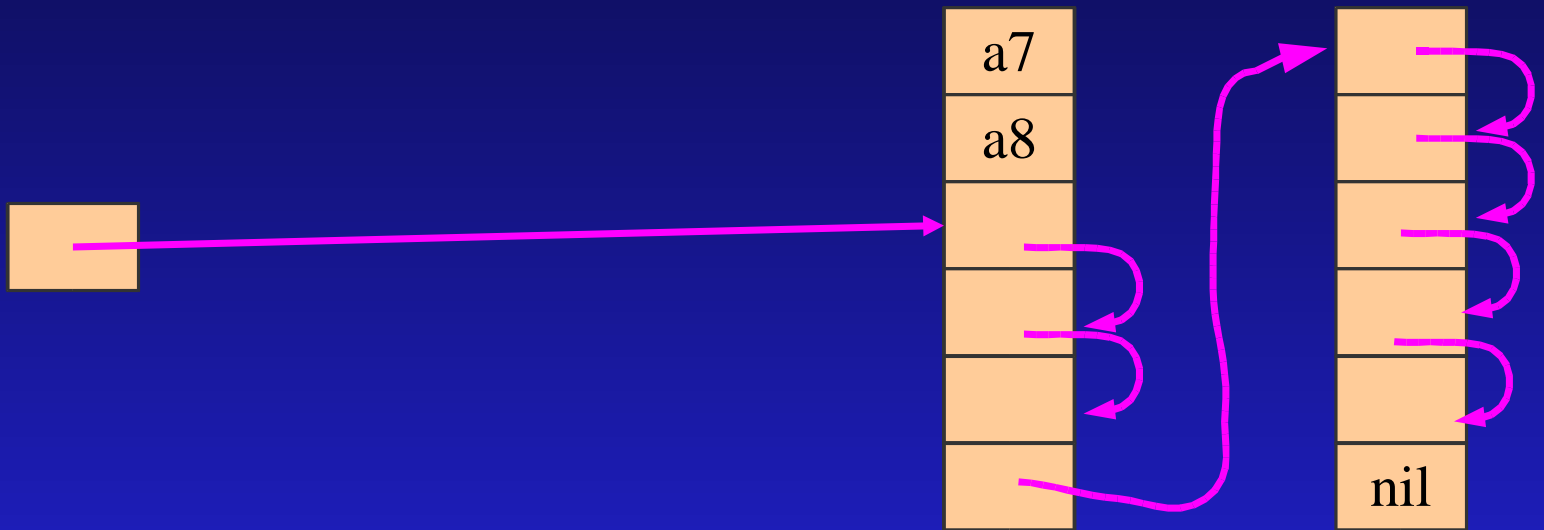
Self-Pointing Log Barrier

- ➡ Log entries initialized to point to next entry.



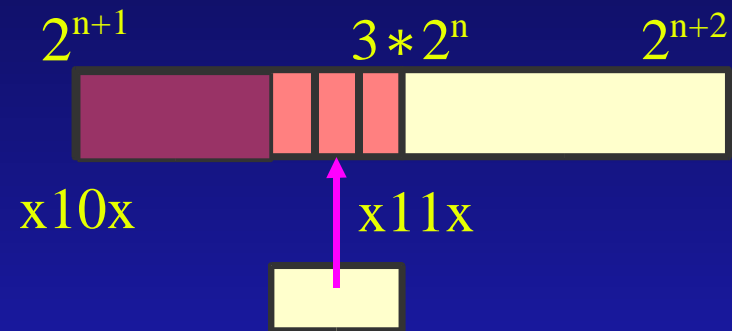
Self-Pointing Log Barrier

- Log entries initialized to point to next entry.
- Concurrent refinement can avoid overflow.



Misalignment UTRAP Barrier

☞ In Solaris™ Operating Environment, it happens that the misaligned access hardware trap can be handled 100x faster than SEGV ==> user-level signal handler.



$reg_f := reg_x + f_offset$

$reg_tmp := reg_next \gg (n-1)$

$M[reg_next-4] := reg_f$

$reg_tmp := reg_tmp \& '110'$

$reg_next := reg_next + reg_tmp$

! $reg_tmp == 4$ or 6

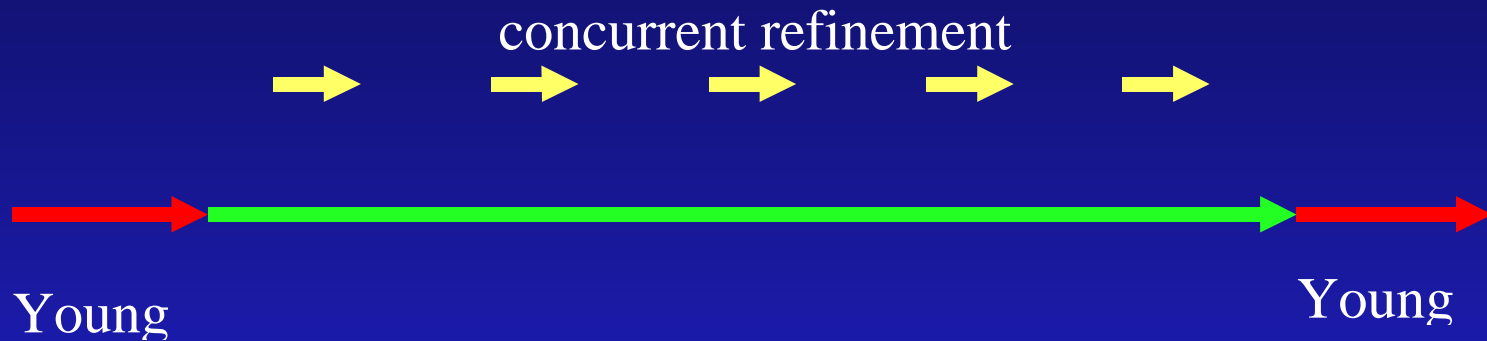
! overflow ==> misaligned

Measurements

- ☞ GCOld: synthetic benchmark.
 - ☞ Large heap, high mutation rate:
 - ☞ 4X speedup in young-gen from concurrent refinement.
 - ☞ All forms of concurrent refinement ~equivalent.
- ☞ Javac: 10% reduction in young-gen times...
 - ☞ ...but cost of non-CT barriers outweighs the gain.
 - ☞ Self-pointing: data cache
 - ☞ Misalignment, 2-level CT: instruction cache.

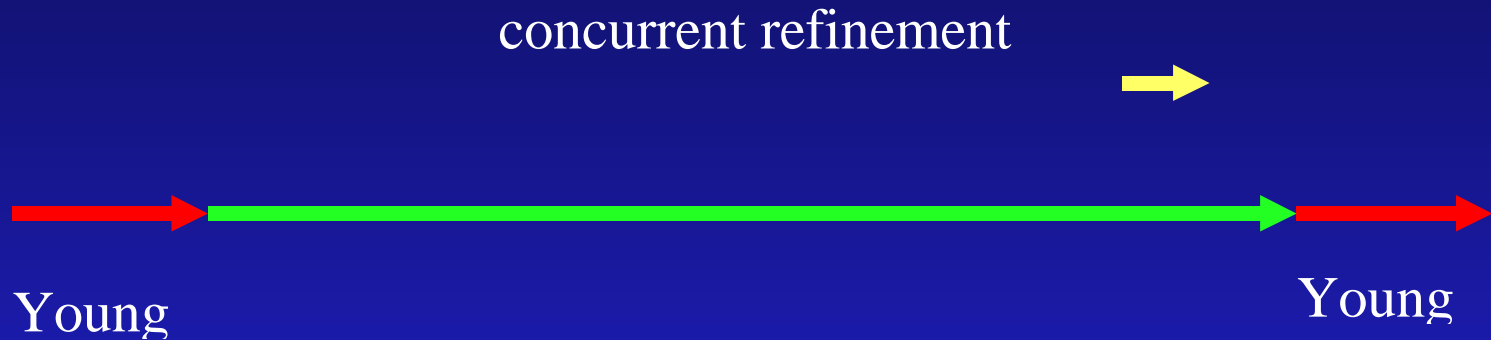
Productization

- ➡ Implemented CT + refinement.
- ➡ "Just-in-time" invocation.



Productization

- 👉 Implemented CT + refinement.
- 👉 "Just-in-time" invocation.



Productization: Results

- ☞ For our favorite Telco application...
- ☞ ...reduced young-GC pause times by 15%.
- ☞ They cared a *lot* about this.

Contributions

- Concurrent Refinement of 1- and 2- level card tables
 - Concurrency issues
 - One card-table base for 2-level table
- Log-based schemes with novel overflow-handling mechanisms:
 - Self-pointing barrier
 - Refinement avoids overflow
 - Misalignment utrap

Conclusions

- Very large heap \implies long young GC pauses:
 - remembered set costs proportional to
 - old-gen size
 - or to # of mutations.
- Two-level tables can obviously help
 - (at some mutator cost).
- Concurrent refinement is a way to use concurrency to improve this situation.
 - Chip multiprocessors (especially asymmetric) might make this more relevant.