Optimizations For Copying Garbage Collection

Erin Zsolcsak
Kathryn McKinley
Maria Jump
University of Texas at Austin

Steve Blackburn
Australian National University

Outline

- Dynamic Storage and Garbage Collection
- Beltway
- Optimizations for Beltway
- Appel Generational Copying GC
- Optimizations for Appel
- Results
- Future Work in Beltway
- Conclusion

Dynamic Storage Allocation

- Allocation of memory from the heap at run-time
- Programmer must allocate and explicitly deallocate this memory in languages such as C and C++

Problems with Explicit Memory Deallocation

Dangling Reference

- Due to premature reclamation
- Object is deallocated but still bound to an identifier

`pi` is deleted before all references to it have been cleared

Example in C++

```
void wrong() {
    int *pi = new int;
    int * q = pi;
        ……
    delete pi;
    int x = 2 + *q;
}
```

Problems with Explicit Memory Deallocation

Memory leak

- Object to which no identifier is bound
- Memory can no longer be returned to the heap

`pi` goes out of scope and `*pi` is never deleted

Example in C++

```
void wrong() {
    int *pi = new int;
          ……
    int x = 2 + *pi;
}
```

Solving the Problem

Garbage Collection

- automatic reclamation of *dead* objects in memory
- releases programmer from the duty of deallocating the object

Garbage – dead objects

Dead – objects that are no longer reachable

Lisp, Haskell, Prolog, Java, C# use garbage collection

Beltway

Key Ideas

- Most objects die young
- Give objects time to die
- Avoid collecting old objects
- Incrementality improves responsiveness
- Copying GC can improve locality

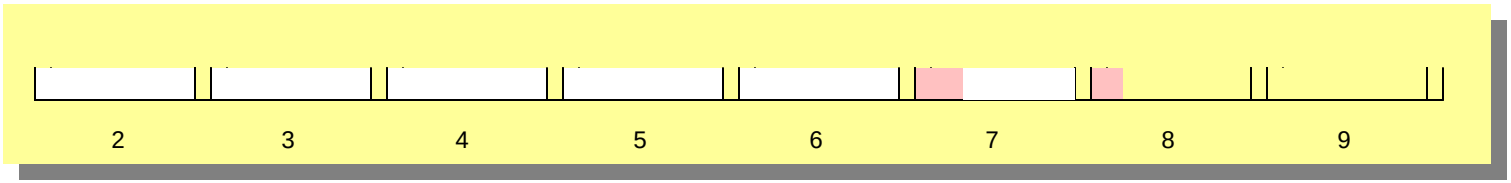
Organizational Terms

- Increment – independently collectible region of memory
- Belt – grouping of increments

Belts are collected independently

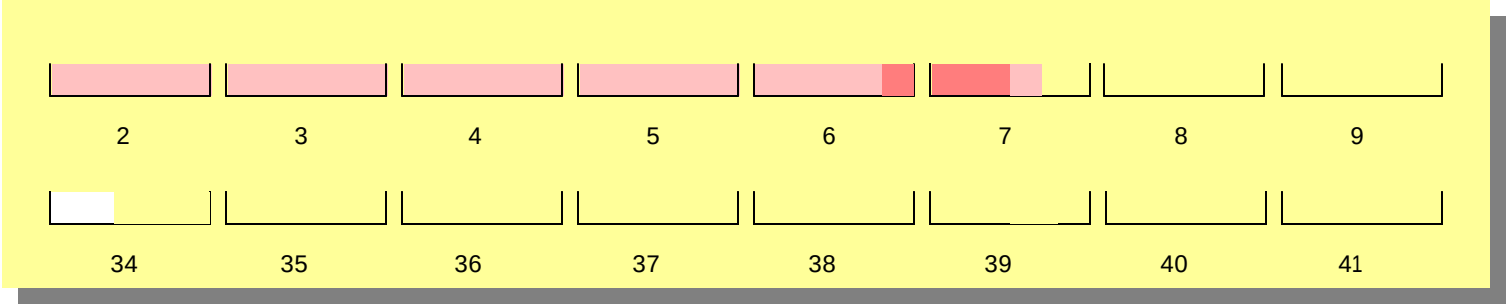
Increments can also be collected independently

A Simple Example



- Notice:
  - Collection occurs in FIFO order
  - Mix copies and newly allocated objects in memory

A More Interesting Example



- Notice:
  - Generational & older first principles
- Called “Beltway X.X”
  - X is the increment size (e.g. “Beltway 20.20” or “Beltway 14.14”)
- “Beltway 100.100” = Appel-style generational

So What's the Problem?

Pointers and Write Barriers

How are pointers handled?

- Intra-increment pointers are ignored

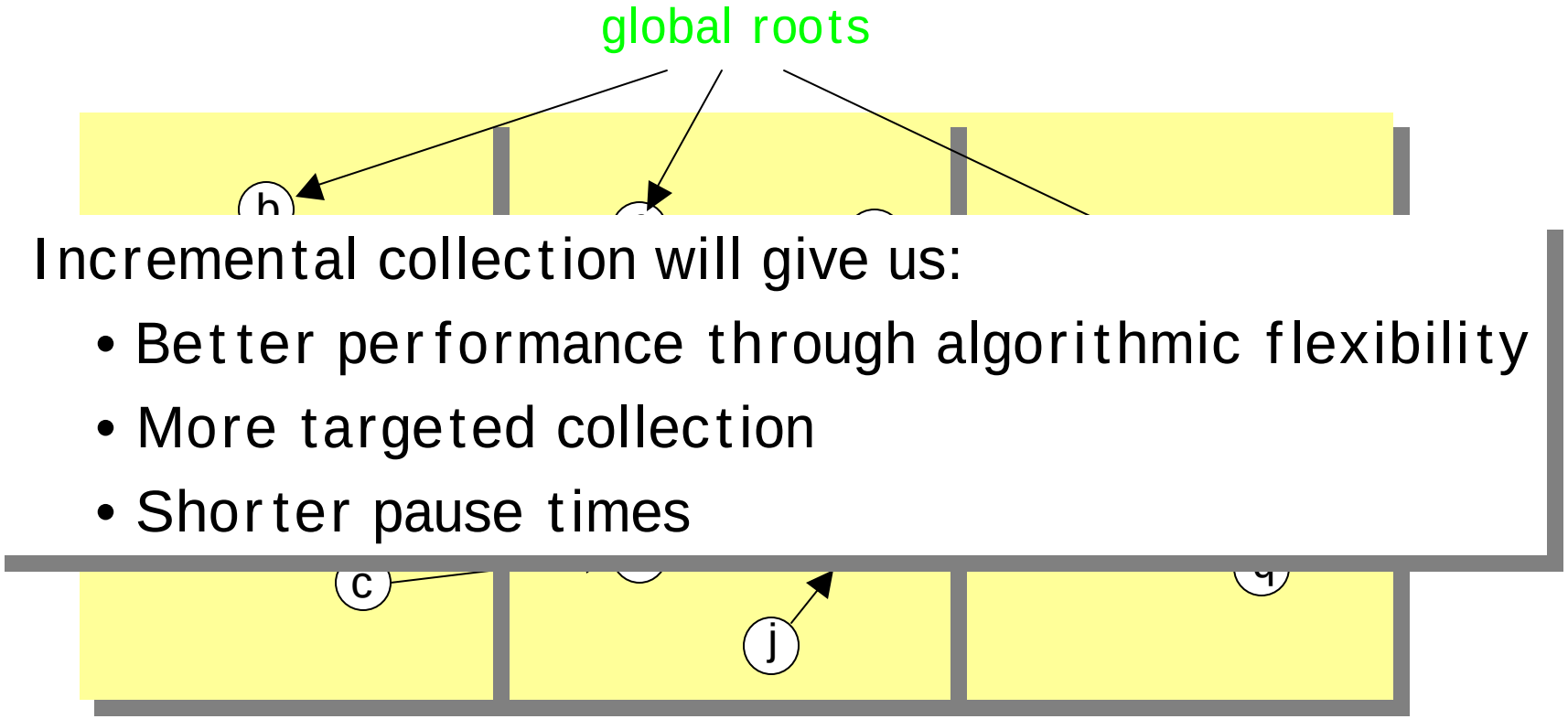
> *Intra-increment pointers* – pointers whose source and target are in the same increment

- Inter-increment pointers must be tested

> If there is a possibility that the *target* will be collected before the *source*, the pointer must be remembered

Only old-young pointers need to be remembered

What are Write Barriers?



Incremental collection will give us:

- Better performance through algorithmic flexibility
- More targeted collection
- Shorter pause times

need to remember: f -> e d-> h k-> n

c -> g

n -> k

Overhead of the Write Barrier

Testing a pointer in Beltway requires:

Fast path test
- Does the pointer cross an increment?

Slow path test
- Does the pointer need to be remembered?

Fast path is cheap
mask, compare, conditional

Slow path is expensive
3 loads and a comparison

Motivation for Optimization

- Slow path was evaluated a lot in the nursery
  - Because of FIFO nursery (look-up to decide whether to remember or not)
  - Causing significant performance hit
- Quick solution: remove FIFO behavior (special case)
- Not getting benefits of Older-First (OF) behavior in the nursery
- Importance of OF in the nursery
  - Avoids collecting youngest objects who have not had time to die yet

Why Was Slow Path Evaluated a lot?

- Hypothesis: Pointers to Type Information Blocks (TIBs)

What is a TIB?

- Created when a class is loaded
- Of type `object[]` allocated into the nursery (like all other objects)
- Stores information about the class

Each objects' header points to a TIB

The TIB Write Barrier

- Updates to the pointer are not seen by the normal write barrier
- Have explicit TIB write barrier which remembers pointers to TIBs if necessary

At runtime, TIB write barriers account for most write barrier activity

Never necessary to remember pointers to TIBs

Optimizations for Beltway

Immortal belt for TIBs

- when a TIB is allocated, it is put into a immortal space where it can live for the duration of the program
- eliminates overhead from copying the TIB over and over

Remove TIB write barrier

- pointers to TIBs are no longer tested by slow path

A Simple Case First

Appel Generational Copying GC

- Objects are grouped into “generations” according to their *age*

Age – determined by amount of heap allocated since object was created

Appel Facts to Remember

- Nursery always collected before older generation (never FIFO)
- Nursery is collected most often

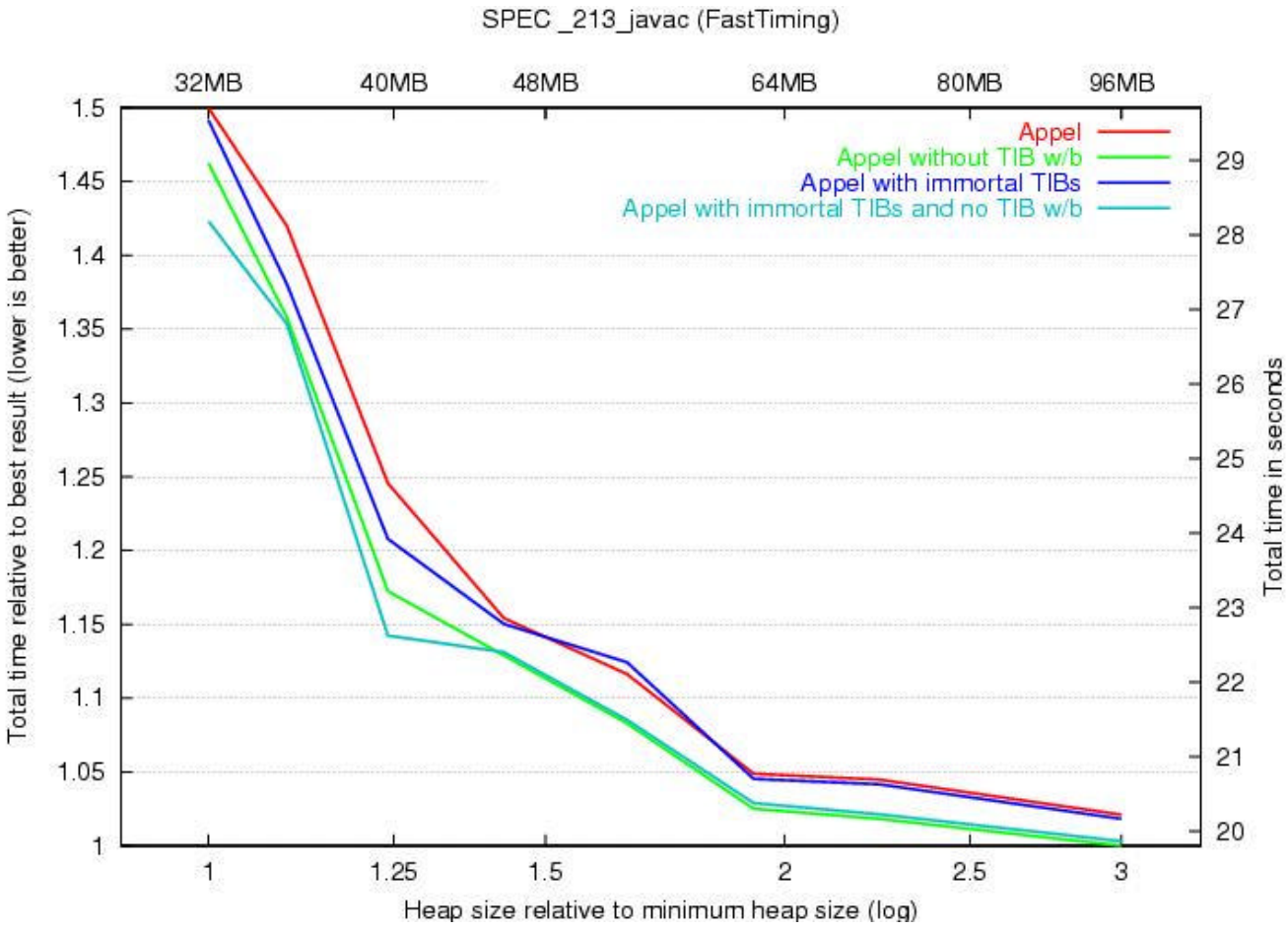
Weak generational hypothesis – most objects die young

Statistics show 80-98% of objects die before one further megabyte of heap storage has been allocated

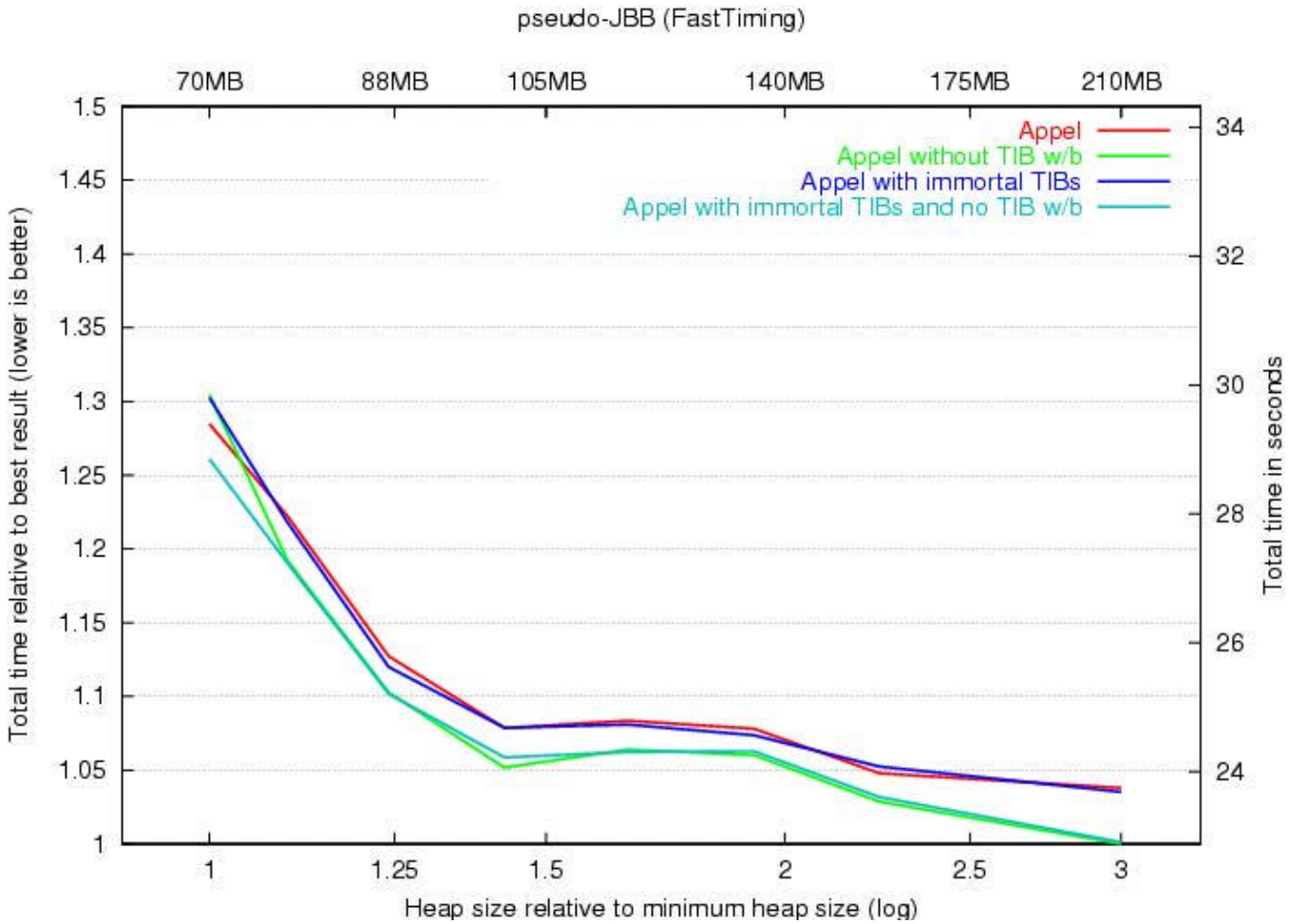
Optimizations for Appel

- Immortal space for TIBs
- Remove TIB write barrier
  - Pointers to TIBs are no longer tested by the *fast* path

Results for Appel



Results for Appel



Results for Beltway

none yet but…

Future Work in Beltway

- Pretenuring large objects

> *Pretenuring* – allocation of long-living objects into an older belt or into an immortal space

- Large object space

> *Large object space* – eliminates the copying of large objects

Conclusion

- In Appel, removing fast path test for TIBs reduces run-time