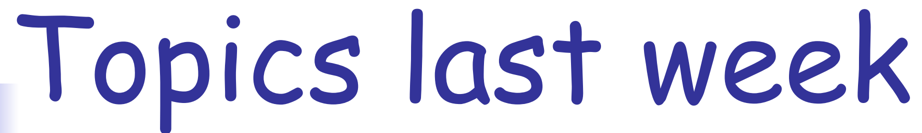
Algorithms for Dynamic Memory Management (236780)

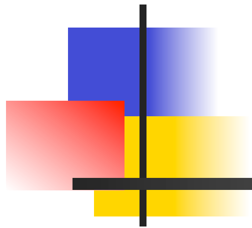
Lecture 6

Lecturer: Erez Petrank



- Concurrent garbage colleciton:
- Dijkstra,
- DLG

Concurrent Garbage Collection

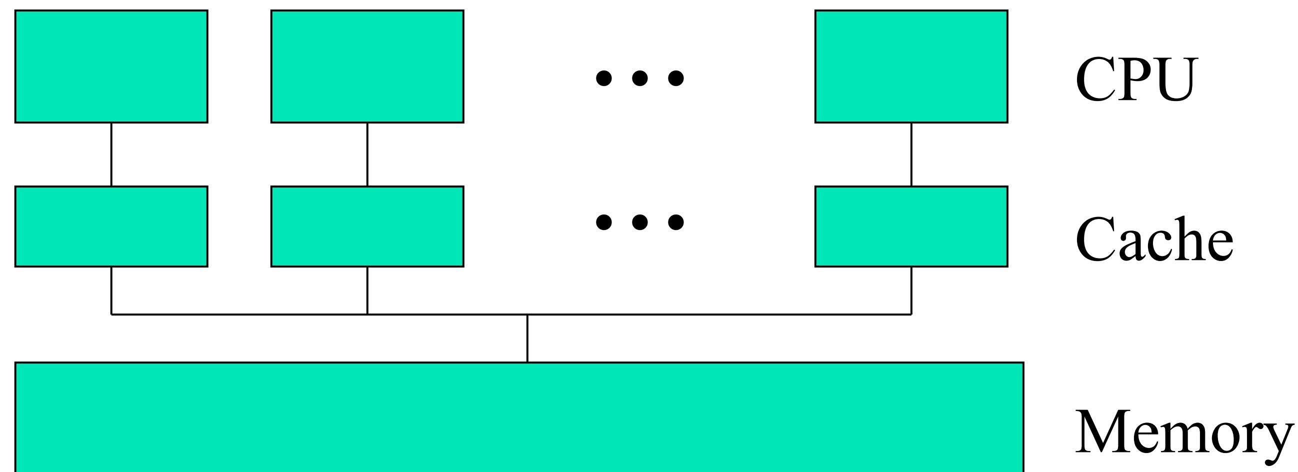


Plan (tentative)

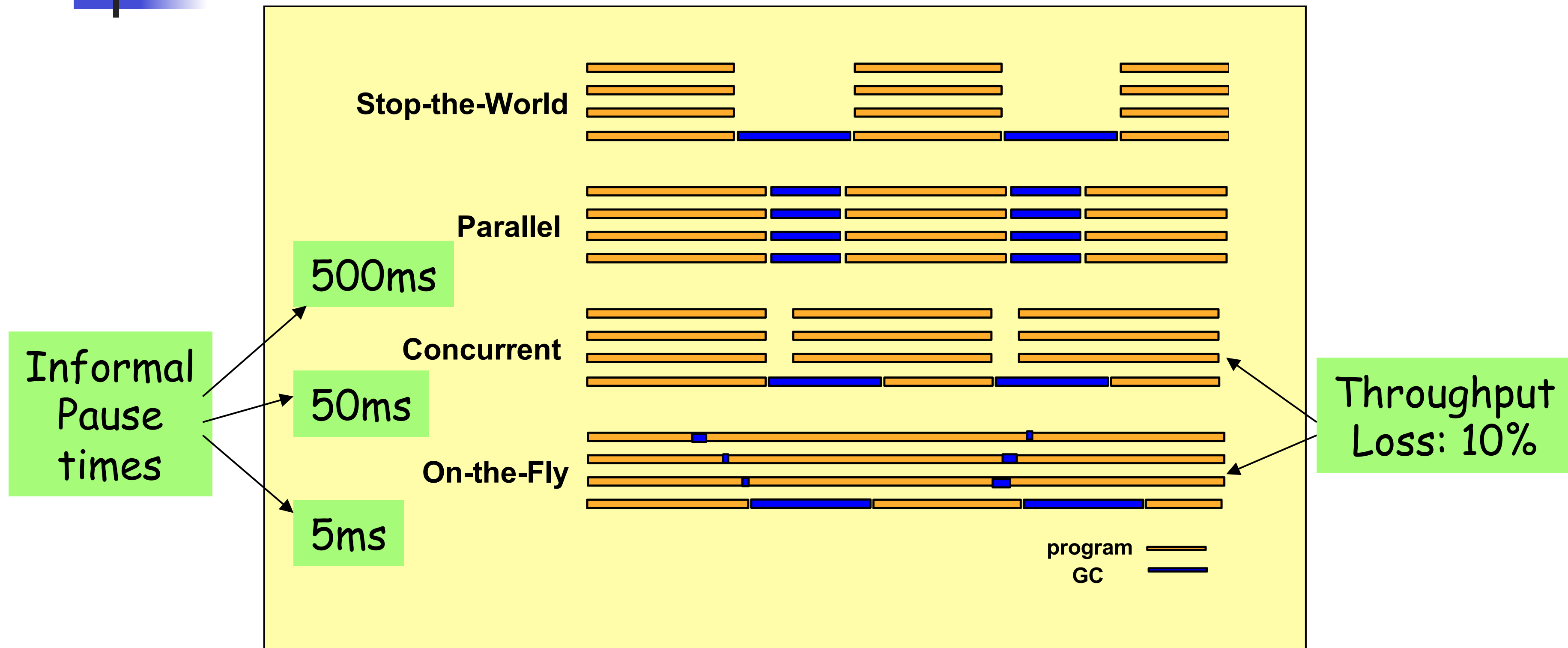
- On the fly collectors:
  - [Dijkstra-Lamport-Martin-Scholten-Steffens 1977]
  - [Doligez-Gonthier-Leroy 1993-94]
- Snapshot: the copy-on-write concurrent collector
  - [Demers-Weiser-Hayes-Boehm-Bobrow-Shenker 1990] + [Furusou-Matsuoka-Yonezawa 1991]
- Mostly concurrent collection
  - [Boehm-Demers-Shenker 1991] + [Printezis-Detlefs 2000] + [Ossia-Ben Yitzhak-Goft-Kolodner-Leikehman-Owshanko 2002] + [Barabash-Ossia-Petrank 2003]
- Sliding Views:
  - [Azatchi-Levanoni-Paz-Petrank 2003] following [Levanoni-Petrank 2001].

Platfrom in Mind

Strong machines (srevers): Symmetric MultiProcessing (SMP)
<u>shared-memory</u>, <u>multiprocessor</u>



Terminology



The DLG Algorithm

[Doligez-Leroy 1993] A concurrent, generational garbage collector for a multithreaded implementation of ML

[Doligez-Gonthier 1994] Portable, Unobtrusive Garbage Collection for Multiprocessor Systems

[Domany-Kolodner-Petrank 2000] Generational On-the-fly Garbage Collector for Java.

[Domani-Kolodner-Lewis-Salant-Barabash-Lahan-Petrank-Yanover-Levanoni 2000] Implementing an On-the-fly Garbage Collector for Java.

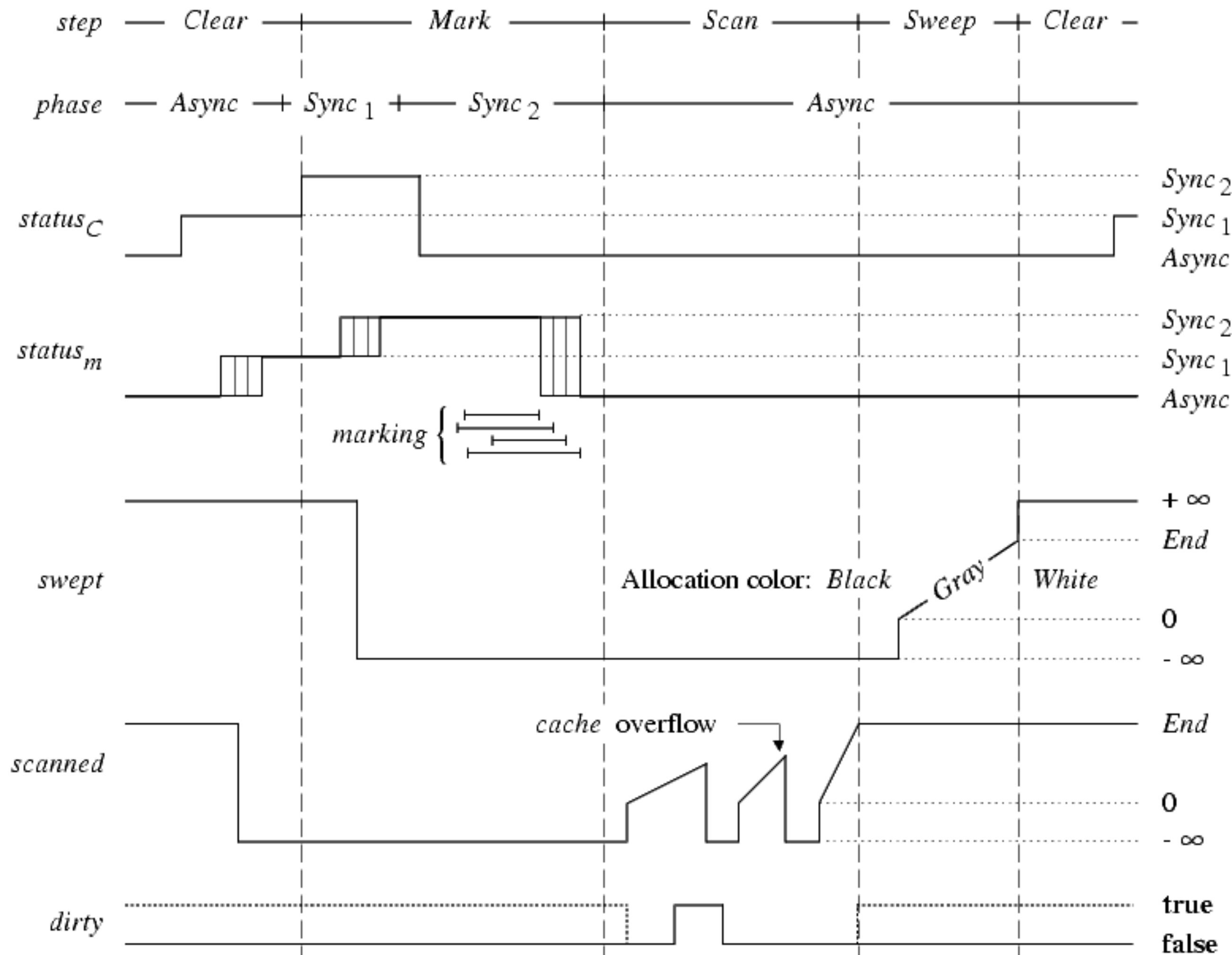
Topics

- Eliminating free-list traversal.
- Using handshakes to accommodate “shade and then change”.
- Eliminating write-barrier on local variables.
- A race between allocation and sweep.
- Avoid repeated heap traversals.

Handshakes

- To summarize, 3 handshakes are used:
  - Tell mutators to start the write-barrier
  - Tell mutators that root marking is approaching
  - Tell mutators to mark their roots (marking started)
- After responding to the first handshake and before responding to the third, the mutator's write barrier marks "old" + "target".
  During the rest of the marking phase, only "old" is marked.
- No real proofs provided…

Timing diagram for global variables



© Erez Petrank

The Color of New Objects

- If we allocate white – it may be reclaimed.
- During mark
  - black – it is true that they are reachable and their sons (none exist) have been traversed.
  - (gray is also OK, but no termination guarantee.)
- During sweep
  - Depending on object location.
  - white, if already swept
  - gray otherwise - to avoid reclamation

Race Allocation - Sweep

Allocation:

- If phase = marking then
  - Set object to black
- Else
  - If address(object) < sweep_pointer then
    - Set object to white
  - Else
    - Set object to black

Sweep:

- If object(sweep_pointer) is black set to white
- If object(sweep_pointer) is white reclaim.

Two problems:

- Using an “old” phase value
- Using an “old” sweep_pointer

The Solution

- It is always safe to set an object to gray
  - Allocation:

```
1.  if phase = marking then
2.     set the object to black;
3.     if phase = sweeping then
4.        set the object to gray;
5.  else
6.     if address(object) < sweep_pointer then
7.        set the object to white;
8.     else
9.        set the object to gray;
```

  - Sweep:
    - If object(sweep_pointer) is black: set to white
    - If object(sweep_pointer) is white: reclaim.

Avoid Repeated Heap Traversals

- Marking terminates when there are no gray objects in the heap.
- Saving heap traversals:
  - DLG went over the heap to find gray objects.
  - The practical Java implementations used a markstack
  - Parallel access to a markstack must be addressed in a modern implementation. We will elaborate in future lectures.

No Proofs, No Algorithm

- We will not fully present or partially prove the DLG collector.
- We only discussed major issues in the design.
- The proof that appears in the paper cannot be taught in (this) class.
- Producing a simpler proof = a project!

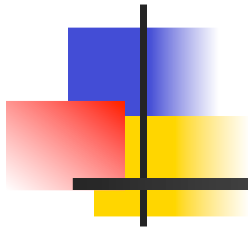
Properties

- On-the-fly.
- No write barrier on local variables.
- Adequate for multithreading.
- Handshakes & write-barriers overhead small.
- Made practical in a series of works.
- A modern version of the algorithm (adapted for Java) was/is used with the IBM production JVM on some IBM platforms.

Next

- Mostly concurrent: IBM's implementation
- Mostly Concurrent collector improvements.
  - An algorithmic approach
- Endo & Taura's idea on shortening pause times
- If time allows:
  - Snapshot copy-on-write
  - Parallel GC

Mostly Concurrent Garbage Collection



[Boehm-Demers-Shenker 1991]

[Printezis-Detlefs 2000]

[Endo-Taura 2002]

[Barabash, Ben-Yitzhak, Goft, Kolodner, Leikehman, Ossia, Owshanko, Petrank 2005]

Recall Motivation

- Garbage collection costs:
  - Throughput
  - Pause lengths
- The goal: reduce pause times with a small reduction in throughput.
- Idea: run (most of the) collection concurrently with mutators.

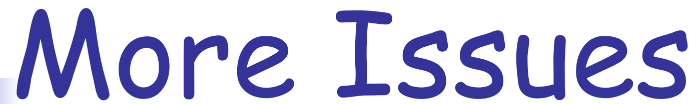
Recall Difficulty

- The heap changes during collection.
- The main problem is that marked (black) objects may point to unmarked (white) objects.
- Solution: trace again all marked objects that were modified by the program.
- Use a card table to record modified objects (actually, cards).

Mostly-concurrent GC

- Trace: run marking concurrently.
  - Write barrier: when a pointer is modified, its card is marked.
- Card cleaning: for each dirty card:
  - Clear dirty bit
  - Re-trace from all marked objects on the card.
- Stop mutators
- Repeat card cleaning while program halted.
- Resume mutators
- Sweep.

 © Erez Petrank



More Issues

- New objects are created black (marked).
- Liveness: Objects unreachable in the beginning of the collection are guaranteed to be reclaimed.
- Floating garbage: Objects that become unreachable during the collection might not be collected.
- Trade-offs: It is possible to do more phases of card cleaning.

© Erez Petrank

Conclusion

- This collector does most of the work concurrently with the program.
- It is stopped for a short while in the end to clear (and scan) all dirty cards.
- Properties:
  - Short pauses, simple write-barrier
  - Overhead: write barrier, repeated scanning of dirty cards.

Parallel, Incremental, and Mostly Concurrent GC

IBM Production JVM since ~2002

Published in PLDI'02, OOPSLA'03, TOPLAS'05

Talk prepared by Yoav Ossia,

modified by Erez

Motivation (Industry Style)

- Modern SMP servers introduce
  - Higher level of true parallelism
  - Multi-gigabyte heaps
  - Multi-threaded applications which must ensure fast response time
- New demands from Garbage Collection (GC)
  - Short pause time on large heaps
    - Not "real-time", but restricted below a given limit
  - Minimal throughput hit
  - Scalability on multi-processor hardware
  - Efficient algorithms for weak ordering hardware
    - We will not talk about this....

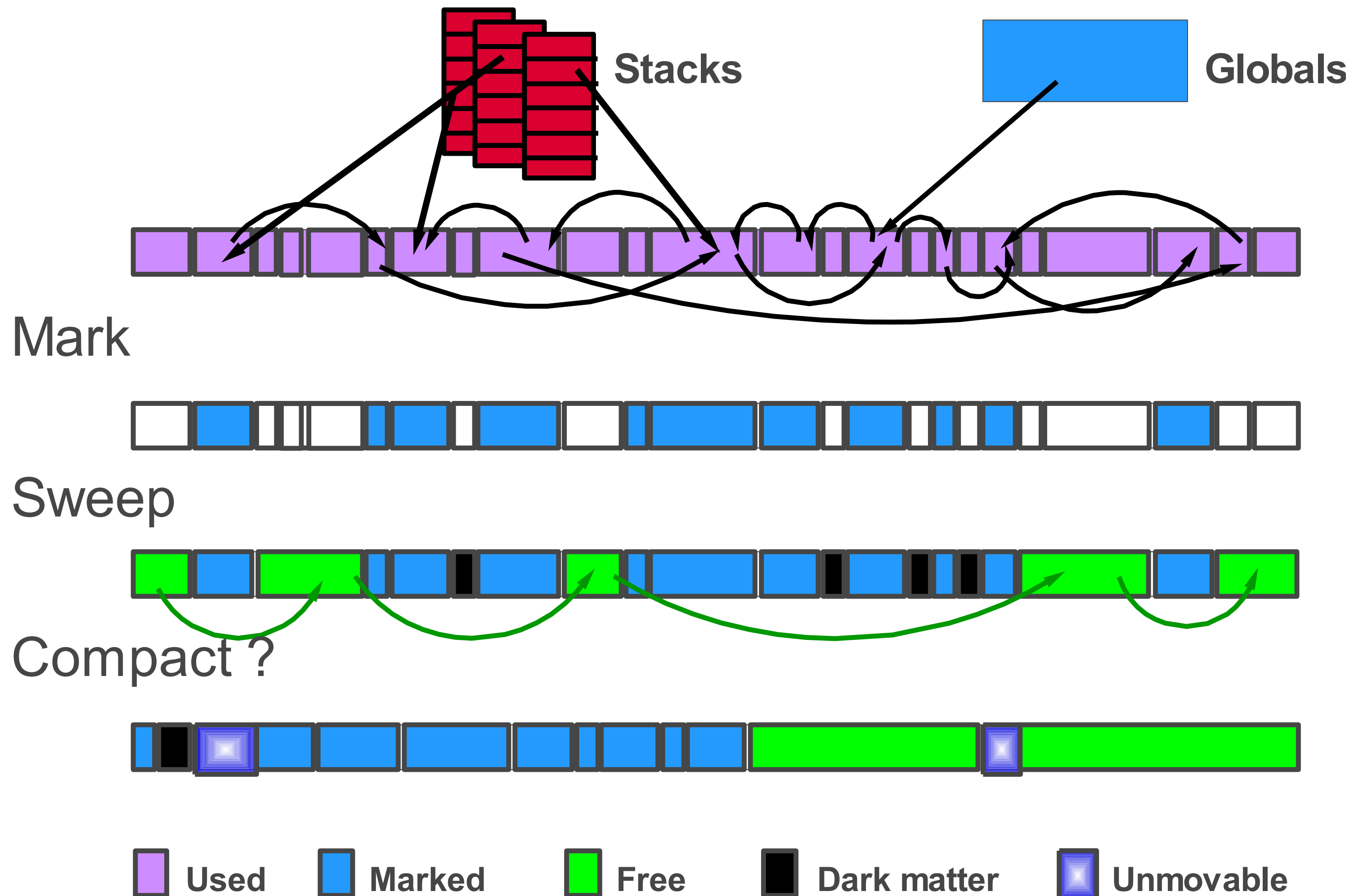
Goals

- Present the ideas and algorithms
  - As done in the IBM JVM
- Introduce the concerns of implementation and some engineering choices.

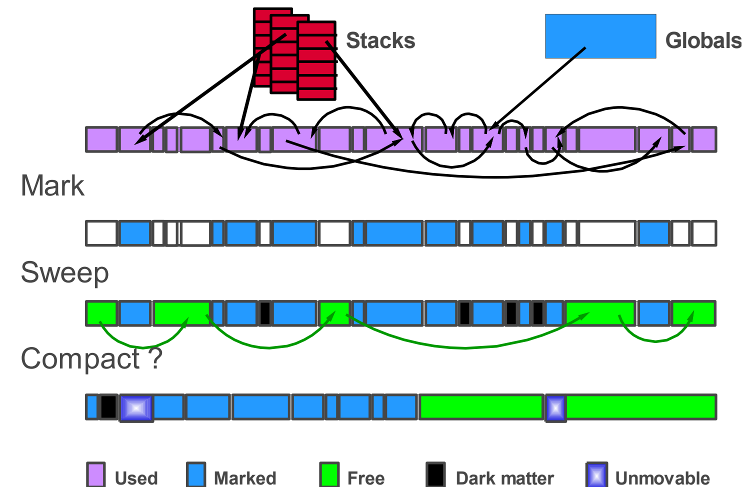
Mark-Sweep-Compact GC

- The collector is mark-sweep-compact.
- Mark - traces all reachable (live) objects in heap
- Sweep - Create a list of free chunks
- Compact
  - Move the live objects, to create bigger free chunks
  - Usually very long.

The General Picture



Subtle Issues



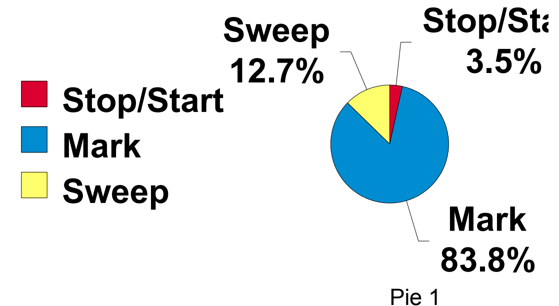
- Mark
  - Handle Risk of *Mark Stack Overflow* (linked list with Next as last field)
- Sweep
  - Make efficient via mark bit-vector and bitwise sweep.
- Compact
  - Consider (tradeoffs): quality of compaction, speed, parallelism.
  - On conservative (non-type-accurate) GC, not all objects can be moved
    - Can't tell if a slot on stack is a reference, or a numeric value

Bitwise Sweep

- A major overhead of sweep is finding & freeing each chunk of 0's in the mark-bit table.
- Solution: free only large chunks of objects.
  - Sweep uses only full zero words (fast search).
  - When found – free the entire chunk of zeros (not only the space presented by the zero word).
- Advantage: looking for a zero **word** is fast.
- Disadvantage: looses some space (but only small chunks).

The Concurrent Collection Overview

- Stop-the-world may get to seconds
- Cost of mark phase dominant



- Correctness kept by use of Write barrier
  - With each change of reference field, remember that the holding object was changed, and later trace it again.
  - In particular, use card marking:
  - Heap partitioned into cards, each card is marked (dirtied) when an object on it is modified.
  - Later, all objects on dirty cards are traced to see if an unmarked object is directly reachable from a marked object on a dirty card.

The Mostly Concurrent Collection

- Tracing done while mutator threads are active
- Mutator runs card marking: "dirties" cards that are modified.
- Card cleaning runs concurently.
- Finally, a short stop-the-world phase
  - Last clean and resulting tracing
  - Sweep (or ever better, lazy sweep)

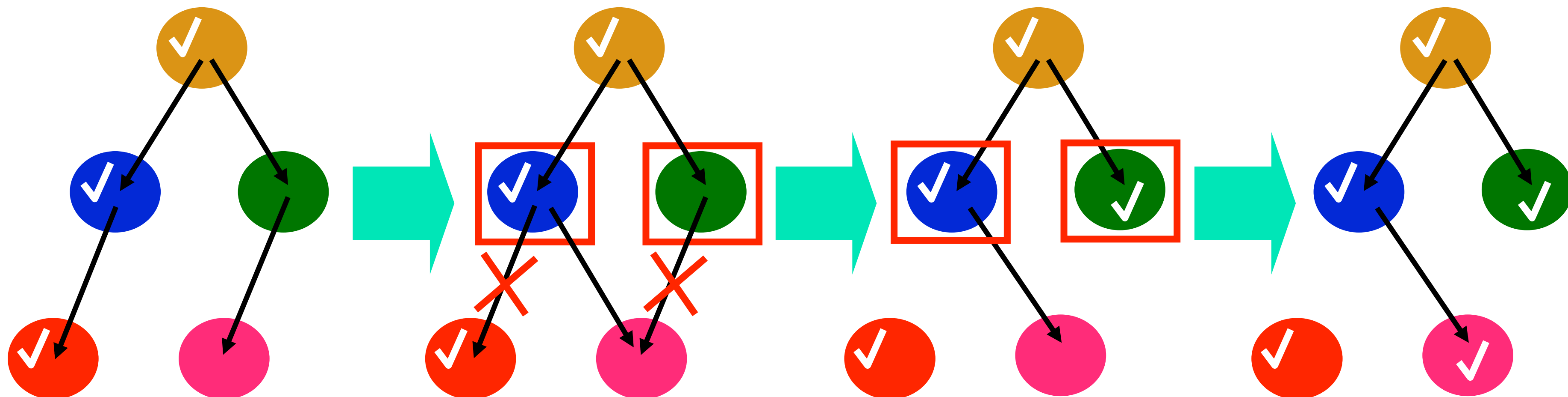
The Write Barrier and Object "cleaning" Solution

Tracer:
Marks and traces

Java Mutator:
Modifies Blue and Green objects
Write barrier on objects

Tracer:
Traces rest of graph

Tracer:
<u>Clean</u> blue object

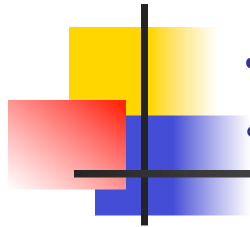


Write-barrier per object

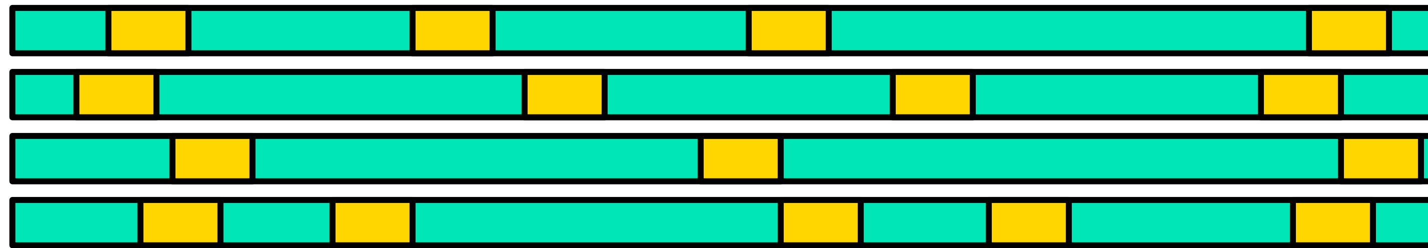
Write barrier per region in heap (AKA ***card***)

Who Runs the Collection ?

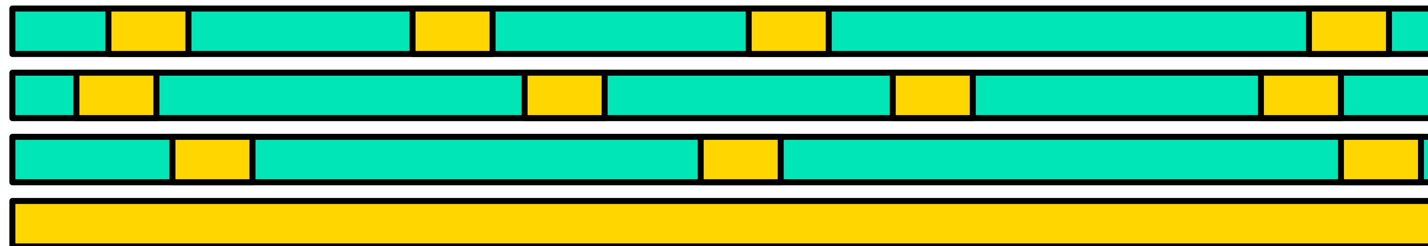
- Tracing done incrementally by the mutator threads, when allocating. (Recall Baker's copying collector.)
- Incremental and parallel: several threads may run their incremental share of the collection in parallel.
- Finally, add some concurrency to the picture.



Incremental and parallel



Incremental, Concurrent and Parallel:



- Concurrent phase
  - Reset the mark bits and the dirty cards table
  - Trace all reachable objects (incremental, parallel, concurrent).
  - Write Barrier records dirties cards in a **card table.**
  - A single card cleaning pass: in each dirty card, retrace all marked objects
- Final stop-the-world phase
  - Root scanning and final card cleaning pass
  - Tracing of all additional objects
  - Parallel sweep (To be replaced by concurrent sweep).

Write Barrier Implementation

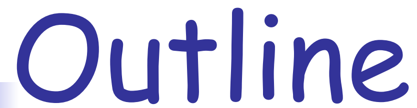
- Activated by the JVM, on each reference change done in Java.
- Cards are 512 bytes; card table has a bit for each card.
- Note that card cleaning may happen anytime
  - Including in the middle of write barrier execution!
- Therefore a race is possible...

Write Barrier Implementation

- A race between card cleaning and write barrier.
- Naïve write barrier:
- Obj1.a = Obj2
  1. Set Obj1.a to Obj2
  2. Dirty the entry of Obj1 in the card table
- If collector finishes between 1 and 2, it will not see the dirty card!
- Switching order does not help since card cleaning can happen between 1 and 2, erasing the dirty mark.

Avoiding Race

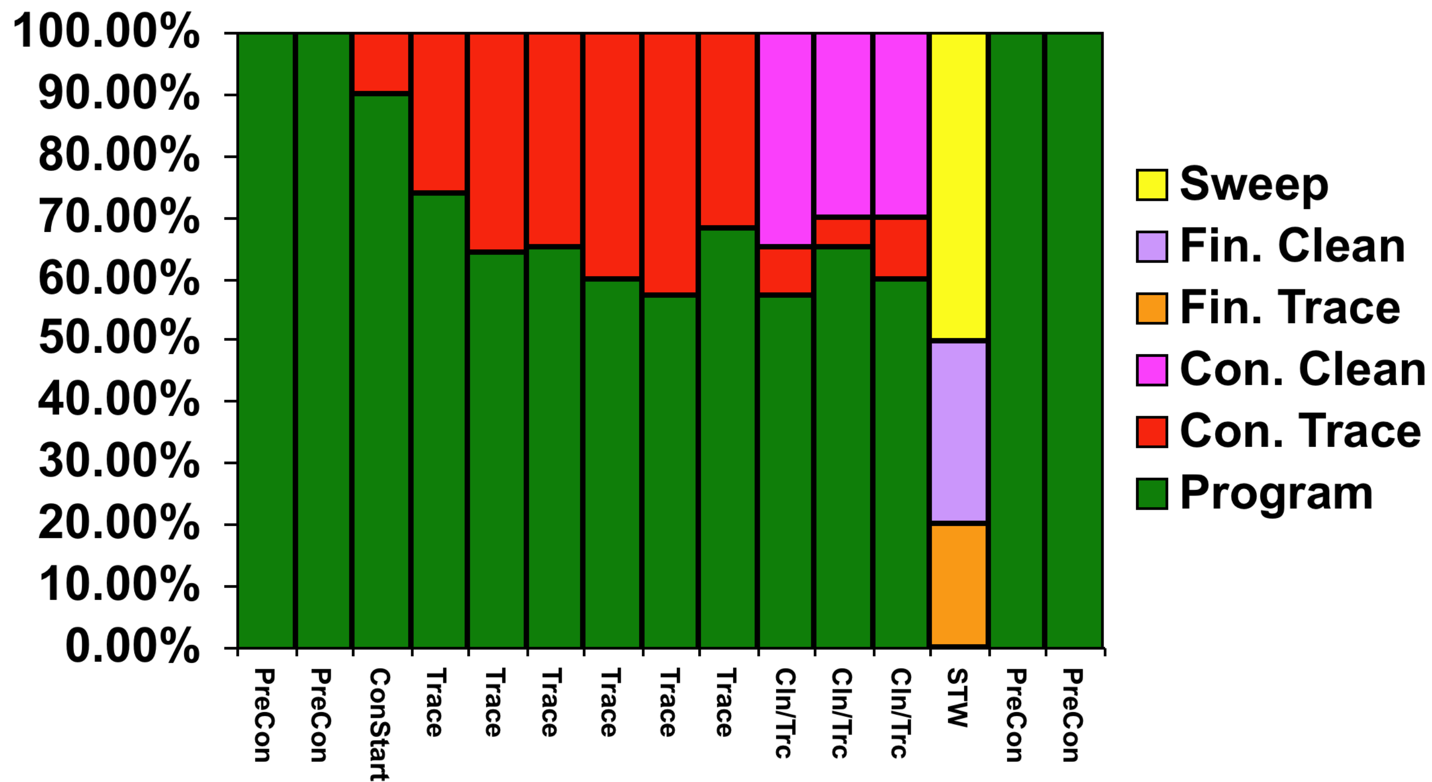
- Extra registration in a local variable.
- Obj1.a = Obj2
  1. Store Obj2 in a root (guarantied to be reachable)
  2. Set Obj1.a to Obj2
  3. Dirty the entry of Obj1 in the card table
  4. Remove Obj2 from root
- This also solves sensitivity to memory coherence, not discussed in this course.



Outline

- Introduction
- Principles of concurrent collector
- Dynamically controlling the concurrent work
- Parallel load balancing mechanism
- Coding issues
- Results (highlights)
- Algorithmic Improvement
  - Results

CPU distribution In Mostly Concurrent GC



The Problem of Punctual termination

- Stop-the-world collection starts when heap is full
- Mostly concurrent aims at completing the marking **exactly** when heap becomes full
  - If heap get filled earlier, program gets stalled.
  - If marking terminates before heap is filled, then we get more garbage collection cycles and pay an efficiency cost.
- Solution: adaptive incremental work.

Advantage of Incremental Work

- May influence the collection speed by choosing how much collection to do with each allocation.
- For normal concurrent collectors there is little control: the ratio of collector threads to program threads.
- The ratio of collector work to program allocation is denoted *tracing rate*.
  - The tracing rate is: work to do per allocated byte.
  - The tracing rate should be: work-to-be-done/free-space.
  - Tracing guarantied to terminate on time.

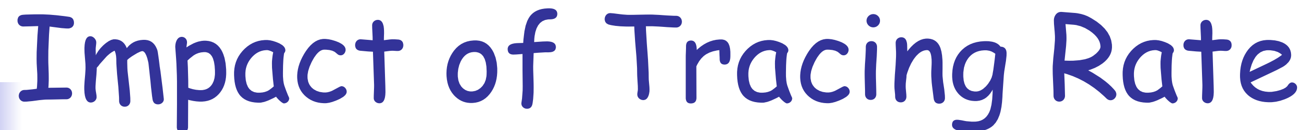
Dynamically Controlling Tracing Rate

- Estimate remaining work.
- If high: increase tracing rate, otherwise decrease it.
- Control assumes no background help, but if work progresses fast, the tracing rate is automatically decreased.
- The user specifies the initial tracing rate and the collector attempts to work with this rate.
  - Note that the tracing rate influences the run

Metering Formulas

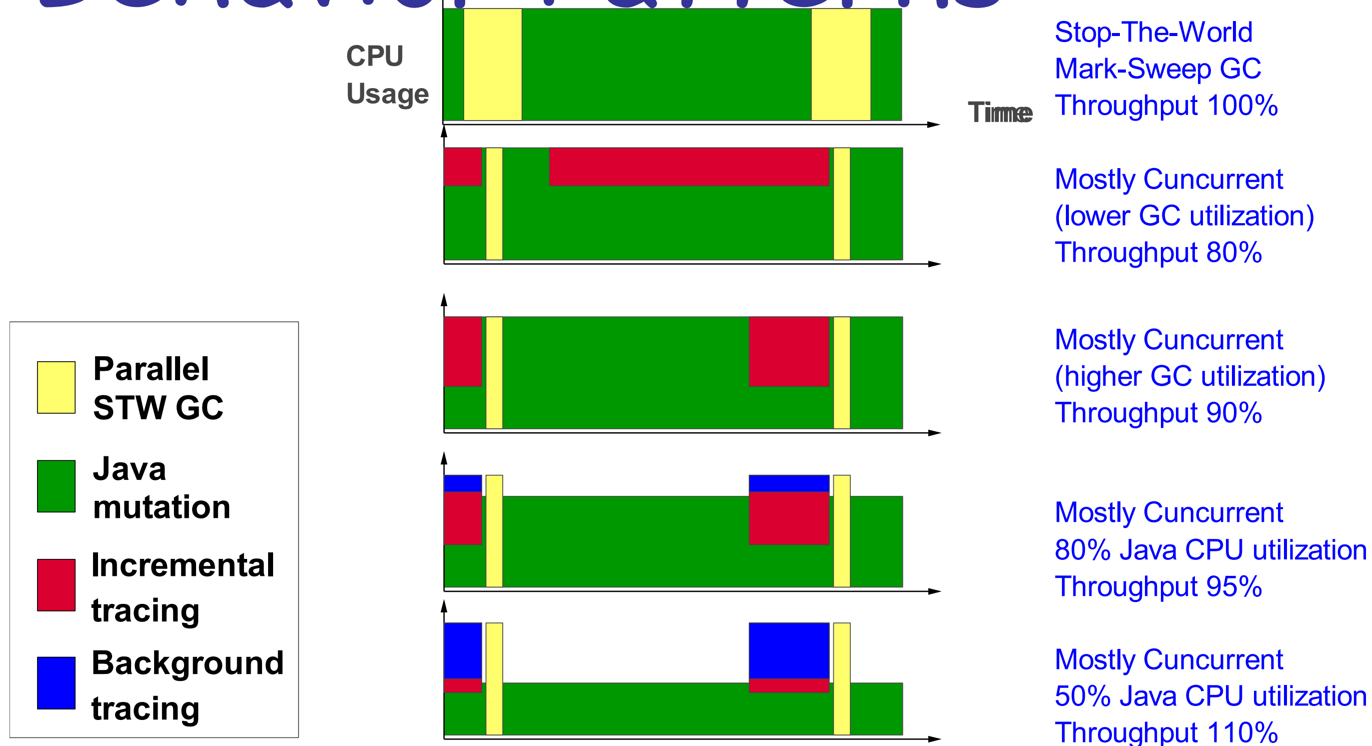
- Starting concurrent GC
  - User-specified Tracing Rate (***TR***)
  - Live Objects estimate (***Lest***), Dirty objects estimate (***Mest***)
  - Start concurrent when free space gets below
    (Lest+Mest) / TR
- Why does the user specify allocation rate?
  - Because this influences the run of the program.

Impact of Tracing Rate



- On a low tracing rate (trace little with each allocation)
  - GC takes a long time
  - Many cards get dirty
  - Repeated tracing takes more time
- Program gets more CPU percentage
- GC is less efficient, so program gets less CPU overall.

Behavior Patterns



JVM Utilization and Performance

with Mark-Sweep GC and Mostly Concurrent GC

Dynamic Metering & Control

- Estimate: remaining work on live objects, remaining work on dirty cards.
- Receive tracing rate (TR) from input (or default).
- Keep track of free space.
- Start a collection when: live + dirty > free × TR.
- During the work, maintain estimates and set TR = (live + dirty) / free.
- The actual formulas also account for background threads' expected work more wisely.

Outline

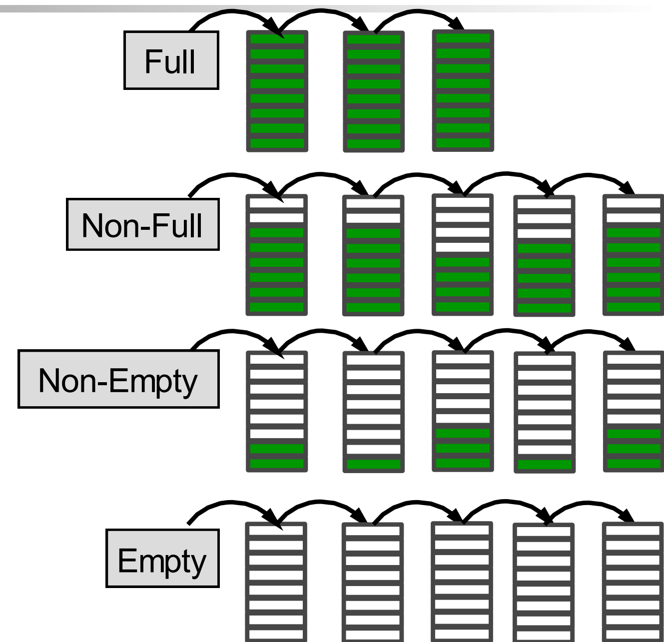
- Introduction
- Principles of concurrent collector
- Dividing the concurrent work
- Parallel load balancing mechanism
- Coding issues
- Results (highlights)
- Algorithmic Improvement
  - Results

Load Balancing Goal

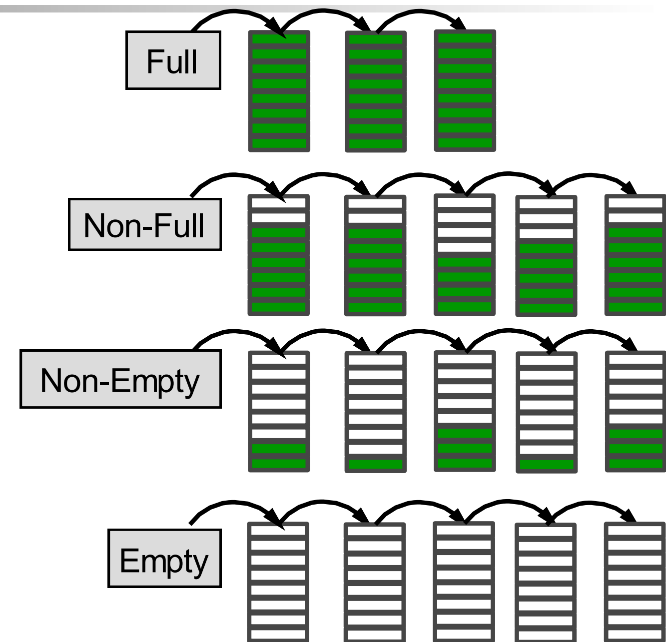
- Avoid starvation
- Work with unknown number of collection threads
- Efficient synchronization
- Simple termination detection.

Work Packets

- Separate mark-stack into smaller work packets.
- Obtain a packet using fine-grained synchronization (compare-and-swap)
- Group packets according to occupancy.
- Maintain a counter for each pool of packets.



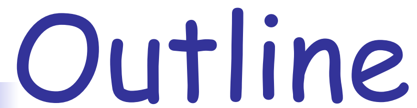
Work Packets



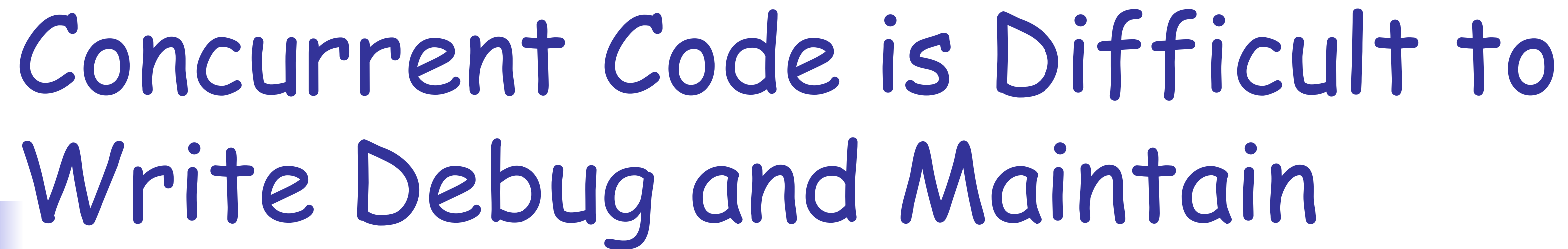
- Each thread keeps two work-packets: one for input and one for output.
- Pop an object from the input work-packet.
- Mark its children and push them all to the output work-packet.
- An empty input packet is returned to the empty-packets pool.
- Try to obtain as full as possible new input packet.
- A full output packet is returned to the “full” pool.
- Get as empty as possible new output packet.

Advantages of WorkPackets

- Fair competition when input is scarce
  - All threads get same chance for tracing input
- Simple detection of tracing state
  - Overflow - All packets are full
    - Scalability is possible, simply allocate more WPs
  - Starvation - Only empty WPs available, but not all WPs in the "Empty" list
  - Termination - all WPs in the "Empty" list
- Positive results measured
  - Low cost of synchronization
  - Fair distribution of work among threads



- Introduction
- Principles of concurrent collector
- Dynamically controlling the concurrent work
- Parallel load balancing mechanism
- Coding issues
- Results (highlights)
- Algorithmic Improvement
  - Results



Concurrent Code is Difficult to Write Debug and Maintain

- Races between concurrent tracing and program
- Races between concurrent tracers
- Scheduling is a major factor
- Debug version cannot reproduce Release bugs
- Problems surface only occasionally
- Behavior is machine dependent

About 40% verification code

- Sanity checks
  - Asserts, consistency checks
- Logging of collection activity, state, and history
  - Shadow heap, for tracing history
  - Shadow card table, for card state and treatment
  - Code to use the above for printing detailed information.

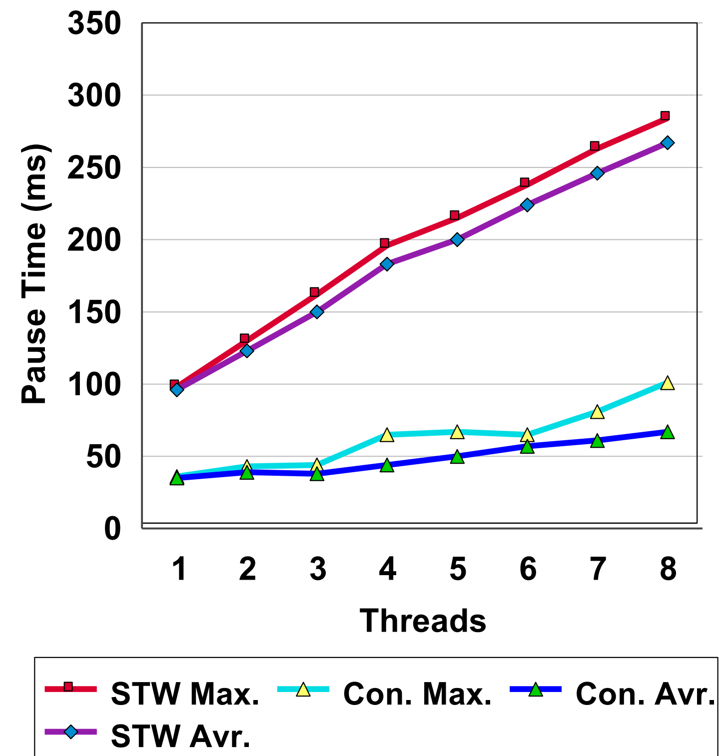
Outline

- Introduction
- Principles of concurrent collector

- Dividing the concurrent work
- Parallel load balancing mechanism
- Coding issues
- Results (highlights)
- Algorithmic Improvement
  - Results

Comparison with STW GC

- Compared to STW MSC
  - Using IBM's production level JVM
  - 4-way machines
  - NT, AIX and IA64
- Mostly testing SPECjbb
  - Server-side Java
  - Throughput driven
  - 60% live objects
- Pause time cut by 75%
- Mark time by 86%.
  - Sweep become dominant
- Throughput hit of 10%



Comparison with STW GC (cont.)

- Also testing pBOB
  - IBM internal benchmark
  - Fit for 2.5 GB heap, with
- Low CPU utilization
  - Many threads
  - Typical to modern server Apps.