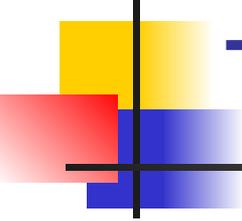


Algorithms for Dynamic Memory Management (236780)

Lecture 6

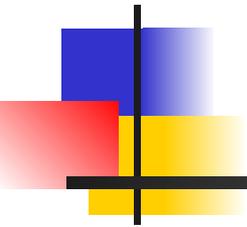
Lecturer: Erez Petrank

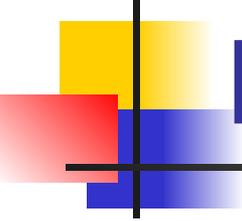


Topics last week

- Concurrent garbage collection:
- Dijkstra,
- DLG

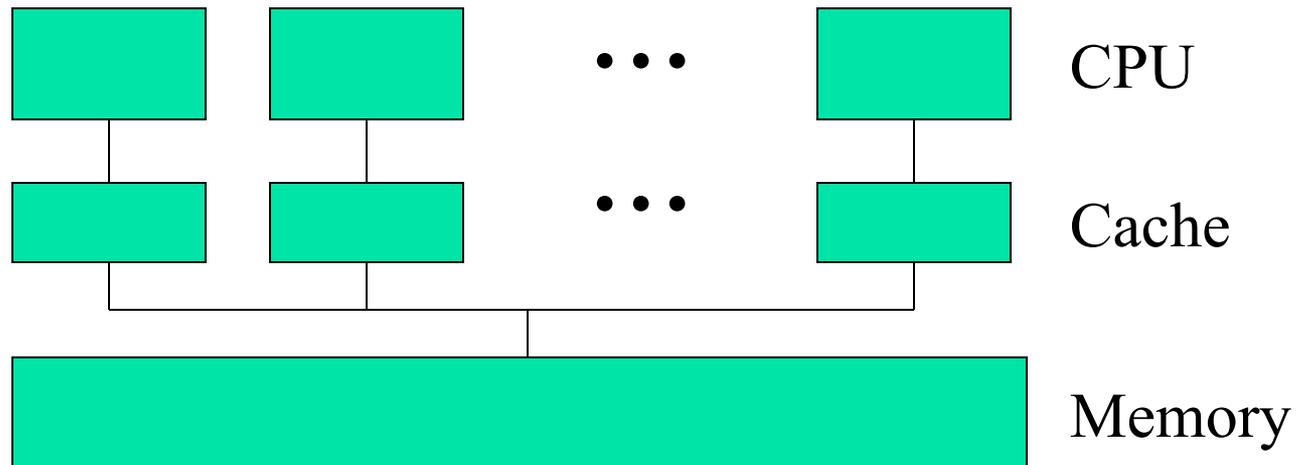
Concurrent Garbage Collection





Platform in Mind

Strong machines (servers): Symmetric MultiProcessing (SMP)
shared-memory, multiprocessor



Terminology

Informal
Pause
times

500ms

50ms

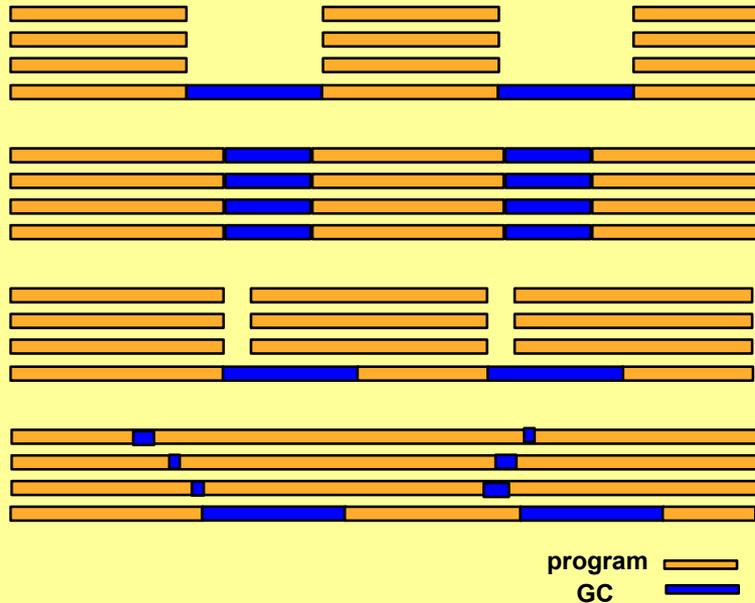
5ms

Stop-the-World

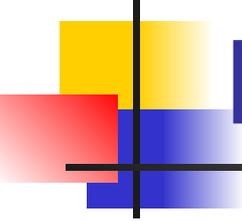
Parallel

Concurrent

On-the-Fly



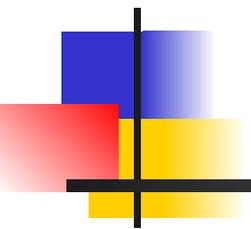
Throughput
Loss: 10%



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

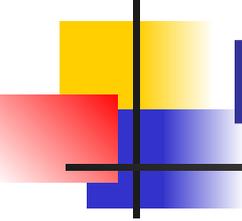


[Boehm-Demers-Shenker 1991]

[Printezis-Detlefs 2000]

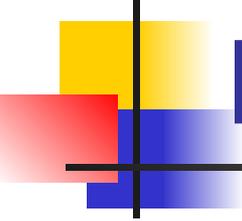
[Endo-Taura 2002]

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



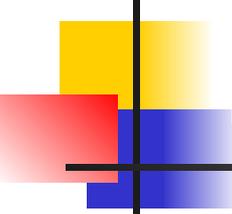
Recall Motivation

- 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.



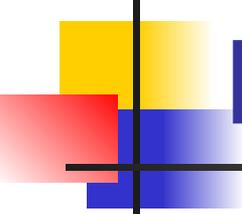
Recall Difficulty

- 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).



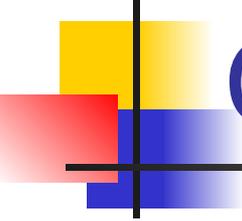
Mostly-concurrent GC

- **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.
 - (Roots are scanned, assuming they are all on dirty cards.)
- **Stop mutators**
- Repeat **card cleaning** while program halted.
- **Resume mutators**
- **Sweep**.



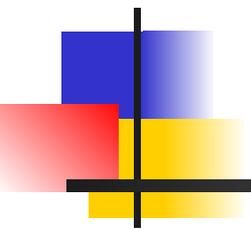
More Issues

- **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.



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

- A specific memory manager
- IBM Production JVM since ~2002
- Published in PLDI'02, OOPSLA'03, TOPLAS'05

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....

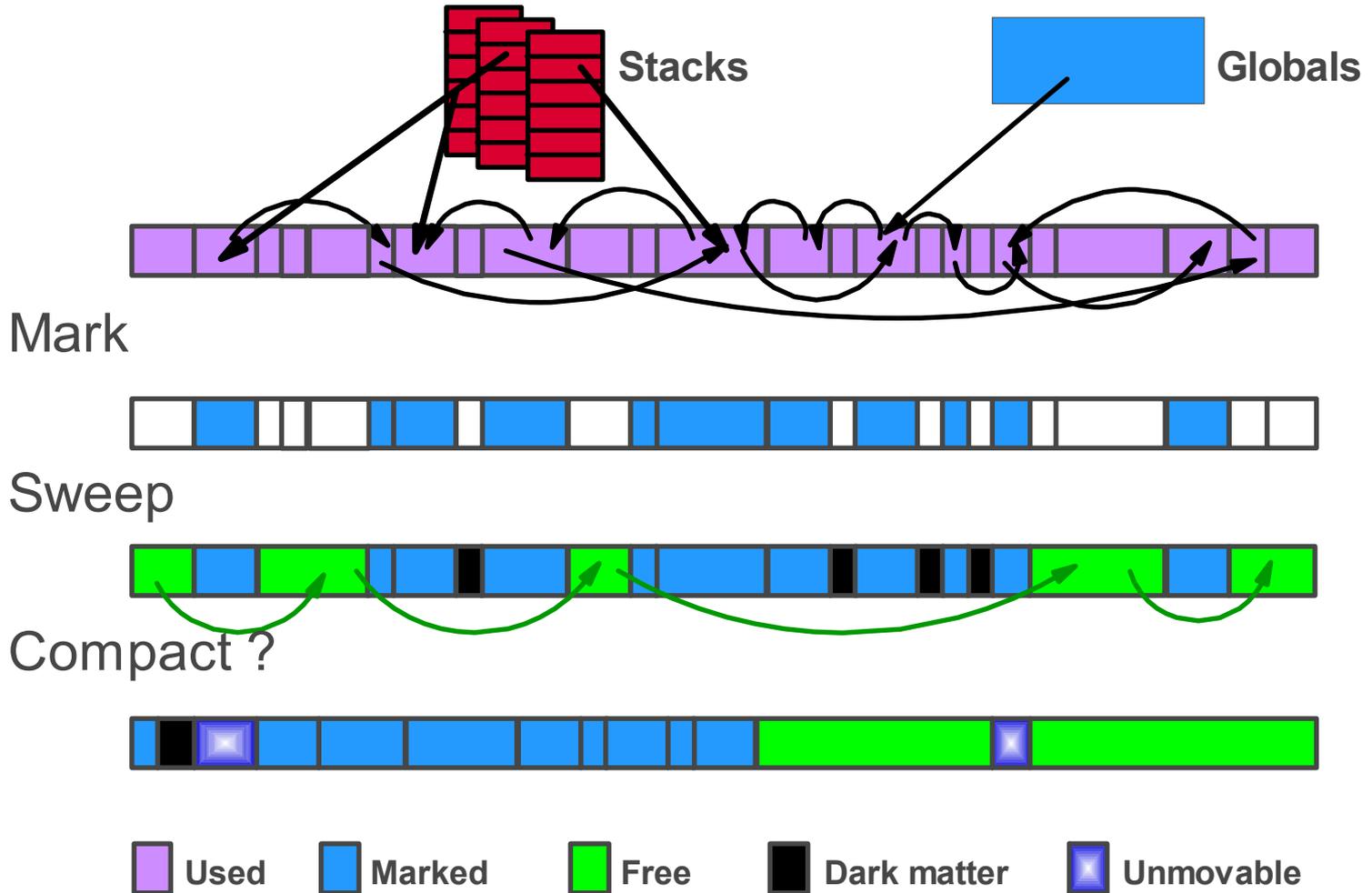
Mark-Sweep-Compact GC

- 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.

Bitwise Sweep

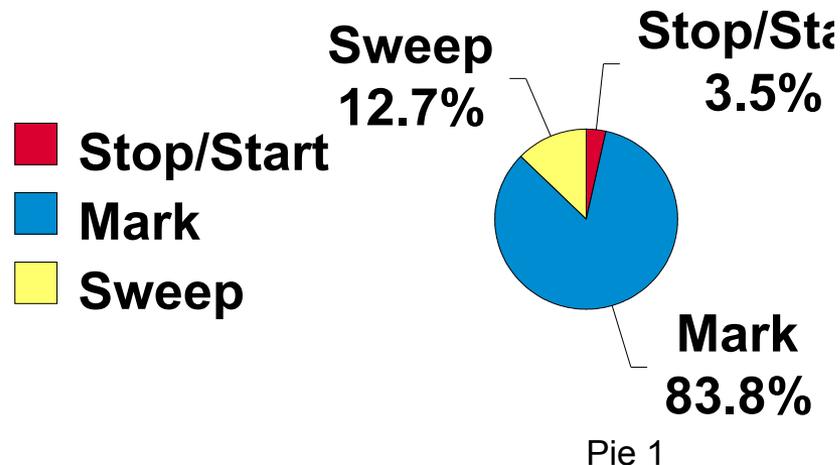
- A major overhead of sweep is finding & freeing each chunk of 0's in the mark-bit table.
- **Solution:** free only large chunks of objects.
 - Sweep uses only full zero words (fast search).
 - When found – free the entire chunk of zeros (not only the space presented by the zero word).
- Advantage: looking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

The General Picture



The Concurrent Collection Costs

- Stop-the-world may get to seconds (at 2004)
- Cost of mark phase dominant



The Mostly Concurrent Collection

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

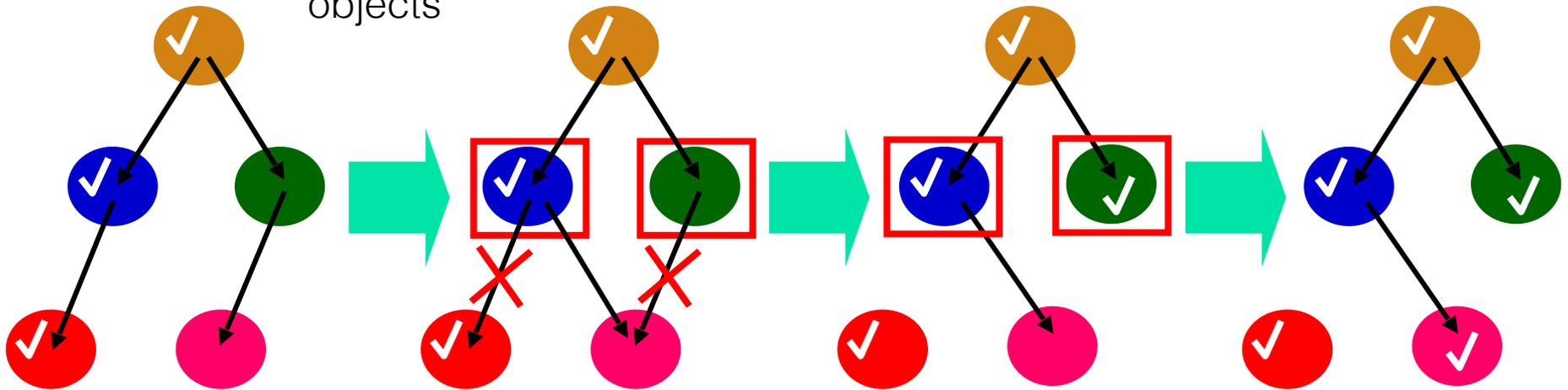
The Write Barrier and Object “cleaning” Solution

Tracer:
Marks and
traces

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

Tracer:
Traces rest of
graph

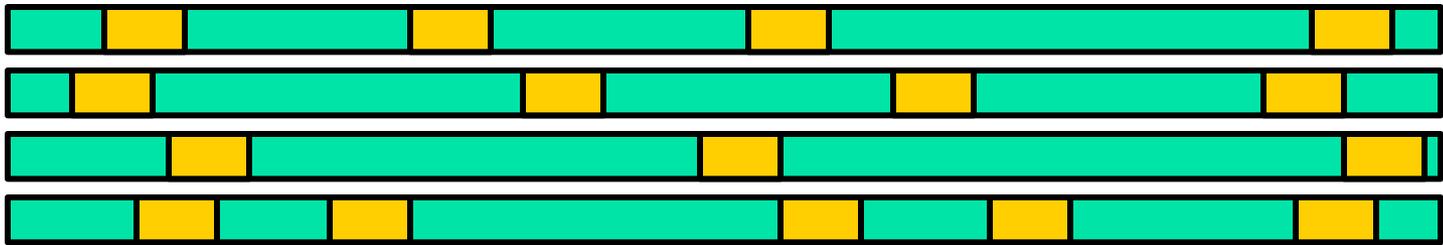
Tracer:
Clean blue object



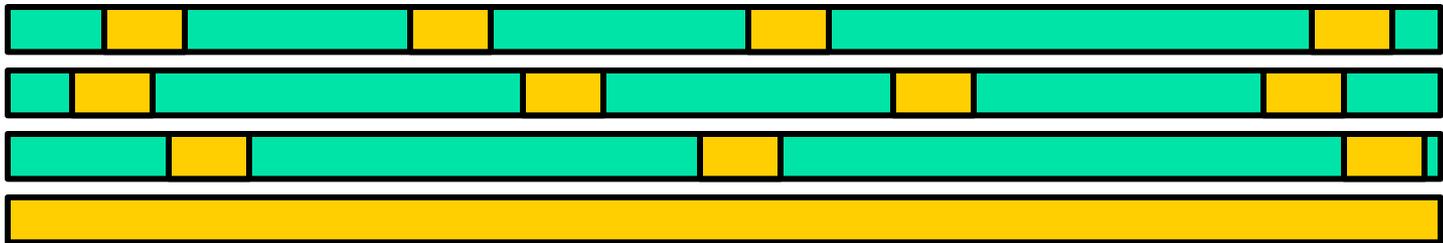
Who Runs the Collection ?

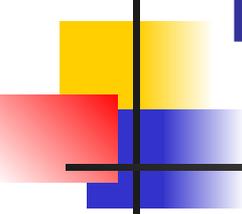
- Tracing done incrementally by the mutator threads, when allocating. (Recall Baker's copying collector.)
- Incremental and parallel: several threads may run their incremental share of the collection in parallel.
- Finally, add some concurrency to the picture.

Incremental and parallel



Incremental, Concurrent and Parallel:





Phases of The Collector

- **Concurrent phase**
 - Reset the mark bits and the dirty cards table
 - Trace all reachable objects (incremental, parallel, concurrent).
 - Write Barrier records dirty 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

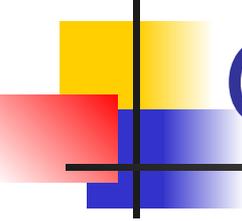
- Activated by the JVM, on each reference change done in Java.
- Cards are 512 bytes; card table has a bit for each card.
- Note that card cleaning may happen anytime
 - Including in the middle of write barrier execution!
- Therefore a race is possible...

Write Barrier Race

- 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 the Race

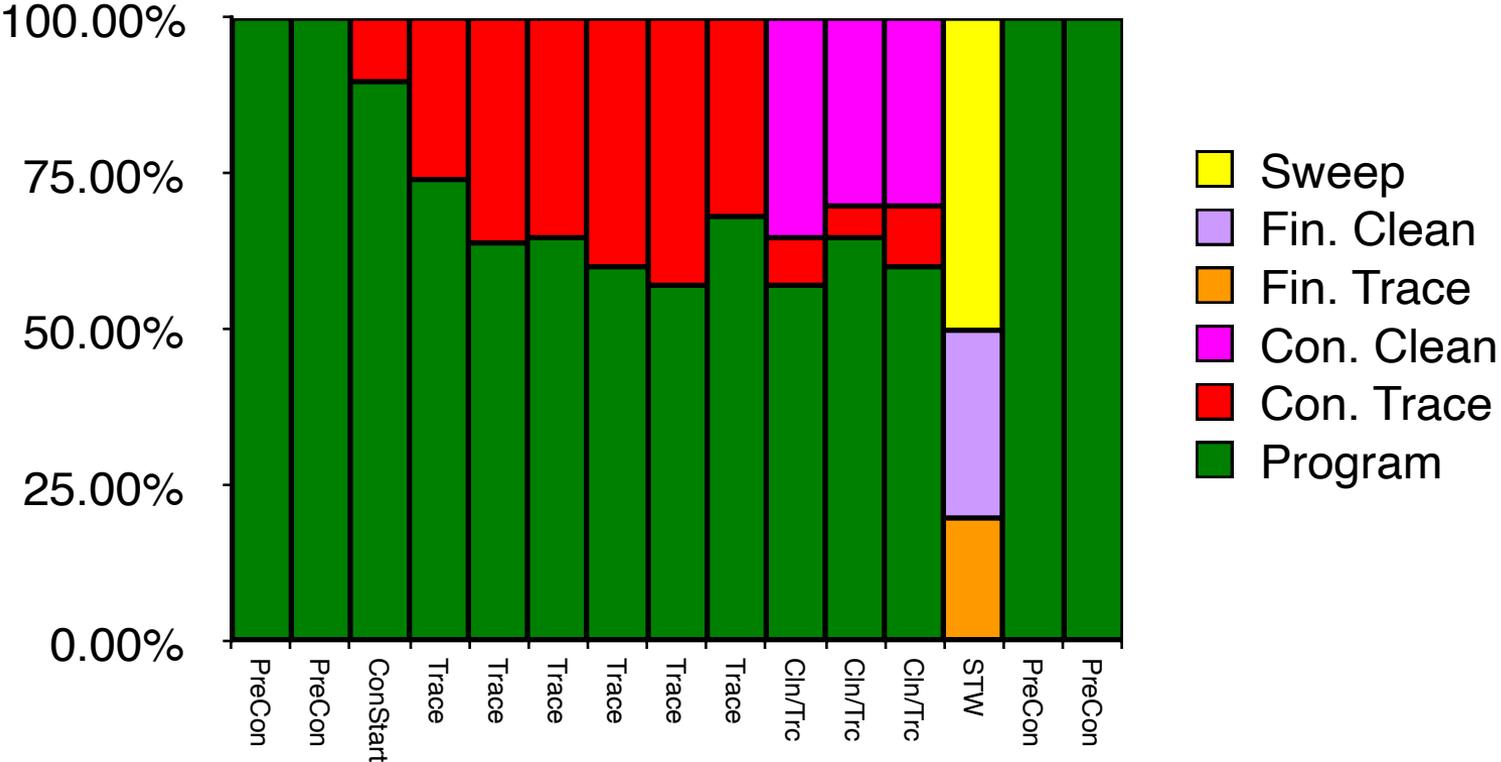
- Extra registration in a local variable.
- `Obj1.a = Obj2`
 1. Store `Obj2` in a root (guaranteed to be reachable)
 2. Set `Obj1.a` to `Obj2`
 3. Dirty the entry of `Obj1` in the card table
 4. Remove `Obj2` from root
- This also solves sensitivity to memory coherence, not discussed in this course.

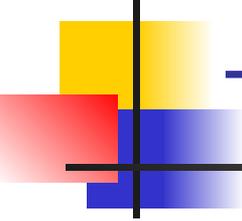


Outline

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

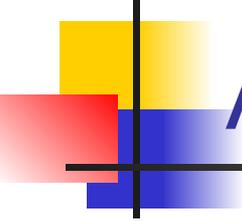
CPU distribution In Mostly Concurrent GC





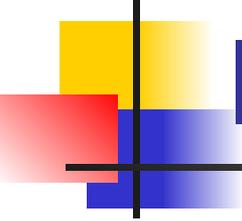
The Problem of Punctual termination

- Stop-the-world collection starts when heap is full
- Mostly concurrent aims at completing the marking **exactly** when heap becomes full
 - Completing GC late - program gets stalled.
 - Completing GC early - we get more garbage collection cycles and pay an efficiency cost.
- Solution: adaptive incremental work.



Advantage of Incremental Work

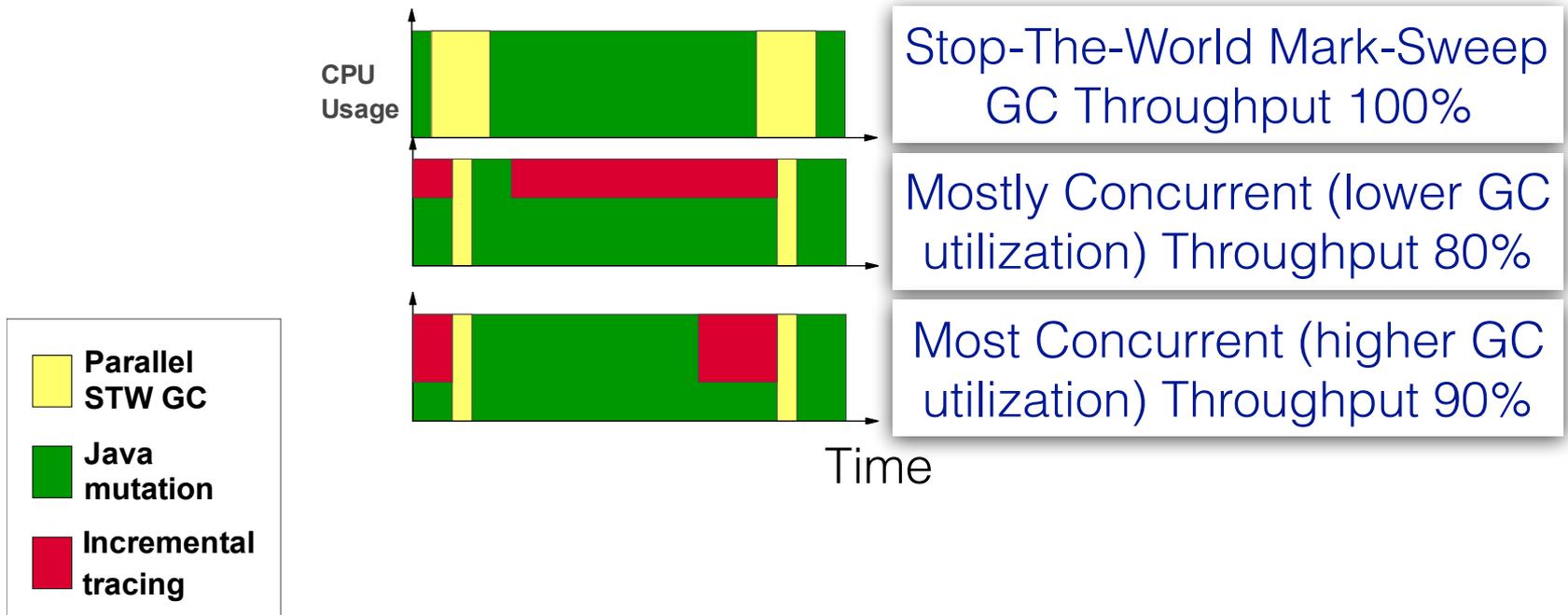
- We can influence collection speed by choosing how much collection to do with each allocation.
- The ratio of collector work to program allocation is denoted **tracing rate**.
 - tracing rate = work to do per allocated byte.
 - Goal: tracing rate = work-to-be-done/free-space.
 - Implies that tracing terminates on time.
- Estimate remaining work and free space, tune tracing rate.



Impact of Tracing Rate

- On a low tracing rate
 - GC takes a long time
 - Many cards get dirty
 - Repeated tracing takes more time
- GC is less efficient, program gets less CPU overall.
- But: program gets more CPU percentage.
- Therefore, we let the user specify an initial tracing rate.
- Start a collection when: estimated work on live objects > free×TR.

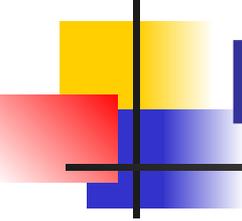
Behavior Patterns



JVM Utilization and Performance with Mark-Sweep GC and Mostly Concurrent GC

Outline

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



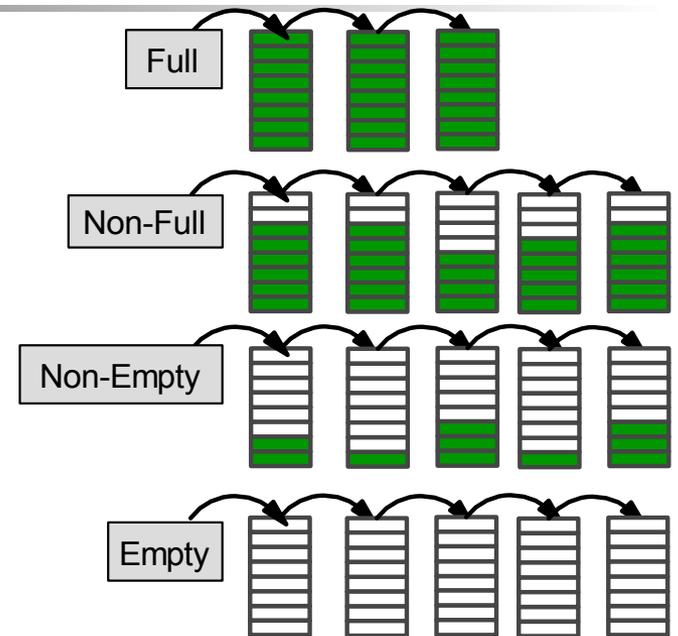
Load Balancing Goal

- Avoid idle time
- Efficient synchronization
- Simple termination detection.

- Small tasks —> good load balancing
- Large tasks —> low synchronization costs.
- Tasks = list of objects (pointers) to traverse.

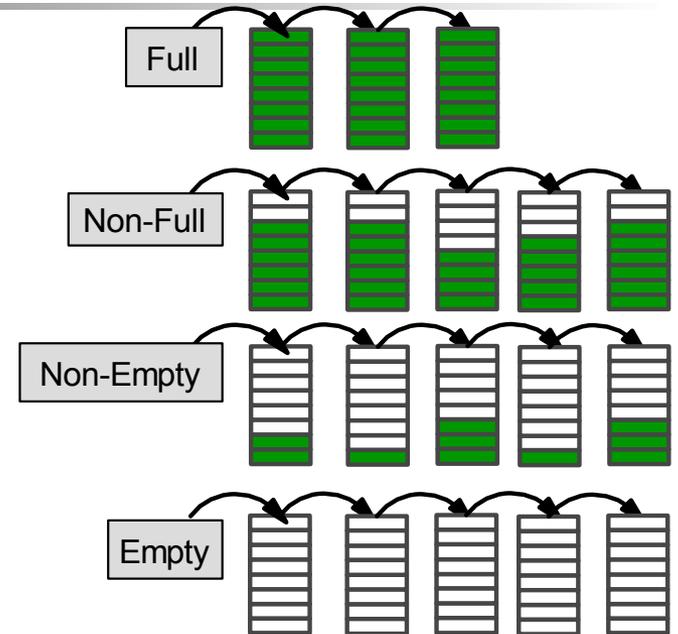
Work Packets

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



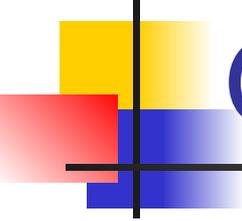
Work Packets

- Each thread keeps an input packets and an output packet.
- Pop an object from input work-packet.
- Mark children and push to 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.



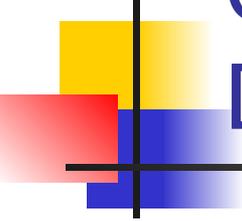
Advantages of WorkPackets

- Easy work distribution
- Packet size determines task size
- Simple detection of termination (all packets in empty list)
- Good performance in practice.



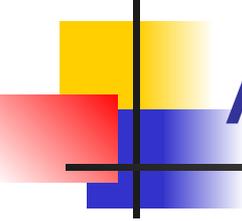
Outline

- 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
- Debug version cannot reproduce Release bugs
- Problems surface only occasionally
- Behavior is machine dependent



About 40% verification code

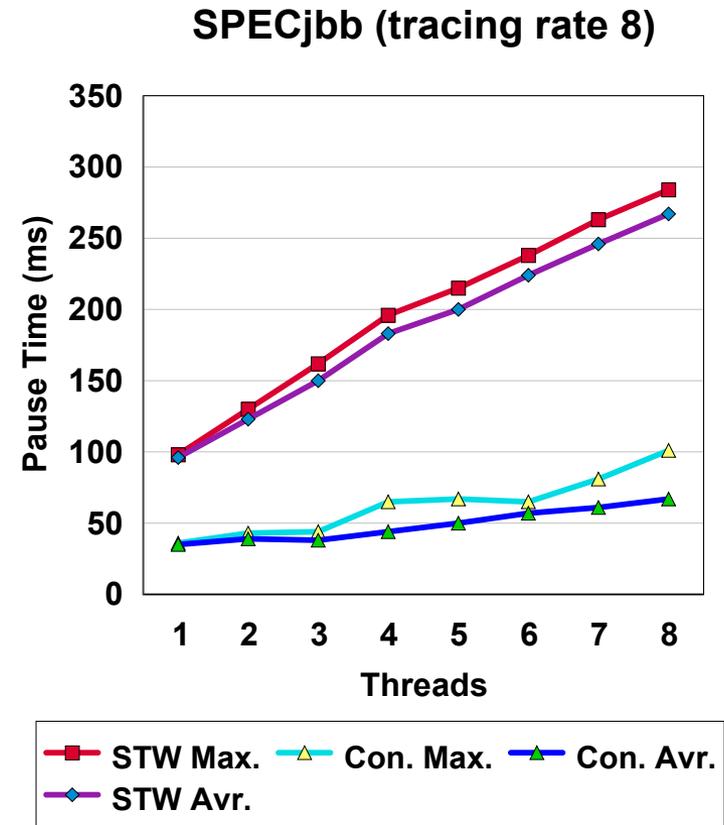
- Sanity checks
 - Asserts, consistency checks
- Logging of collection activity, state, and history
 - Shadow heap, for tracing history
 - Shadow card table, for card state and treatment
 - Code to use the above for printing detailed information.

Outline

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

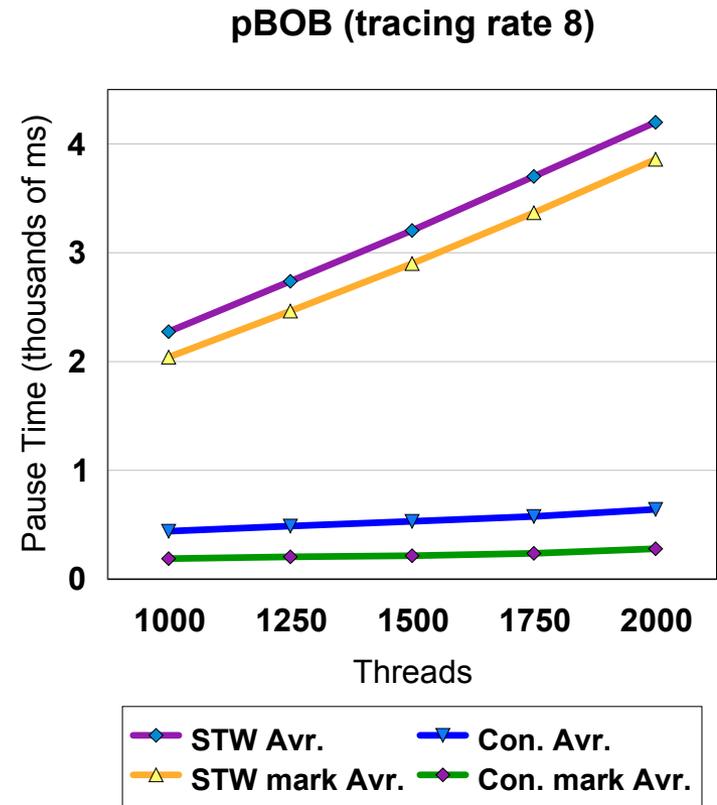
Comparison with STW GC

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



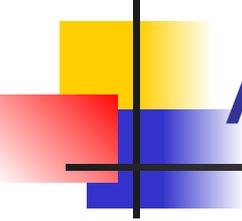
Comparison with STW GC (cont.)

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



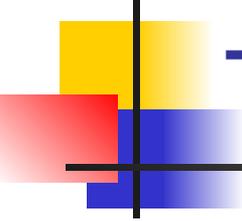
Outline

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



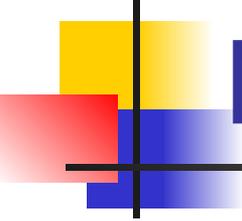
Algorithm Revisited

- [Barabash-Ossia-Petrank 2003]
- Two simple algorithmic improvements.



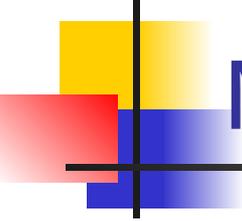
The Repetitive Work Problem

- **An observation:**
 - Suppose a card is dirtied.
 - Reachable objects on this card are marked and traced by the collector.
 - All these objects are later traced again during card cleaning.
 - Outcome: repeated tracing
- **First Improvement:** Don't trace through dirty cards



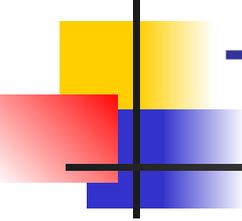
Moreover ...

- On typical benchmarks:
 - Specific “hot” objects get modified frequently.
 - (Almost) all hot objects reside on dirty cards.
 - They point to objects that become unreachable later.
 - Since we trace these hot objects twice, we increase work and floating garbage.
- Not tracing through dirty cards avoids this waste.



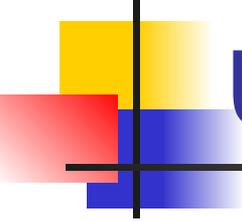
Method & Properties

- While tracing, do not scan an object on a dirty card.
- **Advantages:**
 - Less (repeated) tracing work
 - Reduced floating garbage
- Thus, substantial throughput improvements
- **Disadvantage:** increased pause time



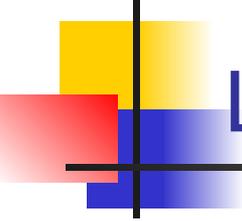
Timing of Card Dirtying

- Observation: dirtying a card tells us that previous tracing does not suffice.
- Therefore: no need to dirty a card if nothing was traced on it.
- **Second Improvement:** Undirty cards with no traced objects
 - Undirtying by scanning cards
 - Undirtying by checking local allocation caches



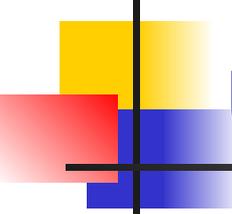
Undirtying via Scanning

- Undirtying periodically on the whole card table
 - Uses a second “traced” card table.
 - The card is marked when first object on it is traced.
- During the scan: undirty any card with no traced objects
 - Very effective in undirtying cards (cuts cleaning by 65%)
 - Scan imposes a cost.
- Frequency is a parameter:
 - Frequent scans: increases chances of catching cards before their objects are traced, but increase overhead



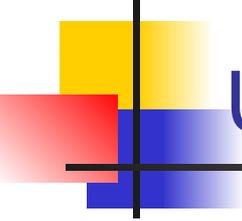
Local Allocation Caches

- An allocation optimization used by various modern JVMs
- The idea: each program thread obtains an “allocation cache” on which it allocates small objects with no synchronization
- Benefits:
 - Reduced synchronization
 - Better locality



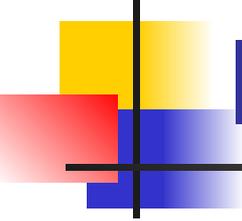
Undirtying via Local Allocation Caches

- Typically, newly allocated objects are immediately initialized, thus their cards are typically dirty.
- But new objects are usually not traced immediately after creation.
- Idea: delay tracing in allocation caches, undirty “after initialization”
- Collector: don’t allow tracing on allocation caches
 - if needed, record pointers to be traced later. This hardly ever happens
- Allocator: after finishing allocation on a local cache, undirty all included pages.
- Cuts the amount of dirty cards by more than 35%, at (almost) no cost.



Undirty Cards with No Traced Objects

- Advantages
 - Reduces the work on scanning dirty cards.
 - This reduces the final stop-the-world phase.
 - Thus, significant impact on pauses.
- Disadvantages
 - Time overhead.

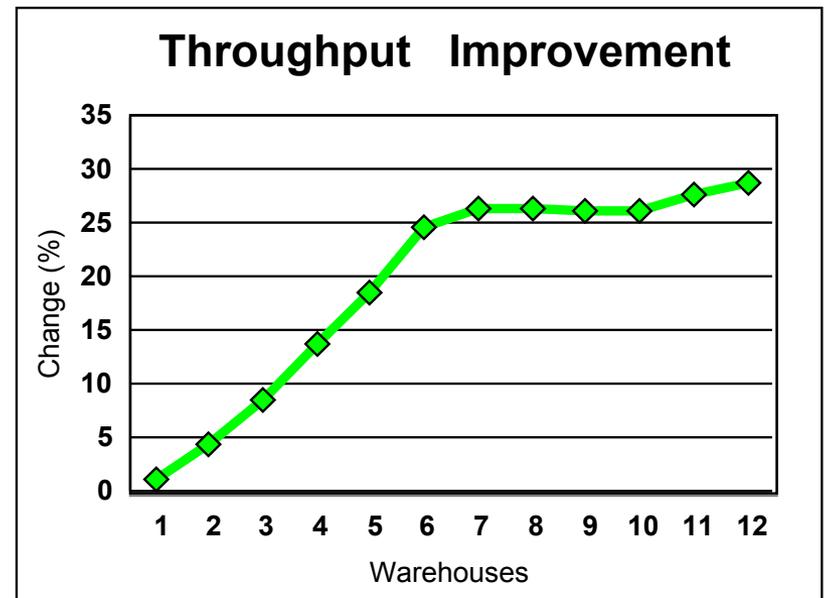


Implementation and Tests

- Implementation on top of the mostly concurrent collector that is part of the IBM production JVM 1.4.0.
- Platforms: tested on both an IBM 6-way pSeries server and an IBM 4-way Netfinity server
- Benchmarks: the SPECjbb2000 benchmark and the SPECjvm98 benchmark suite
- Measurements: performance of the base mostly concurrent collector vs. the improved version

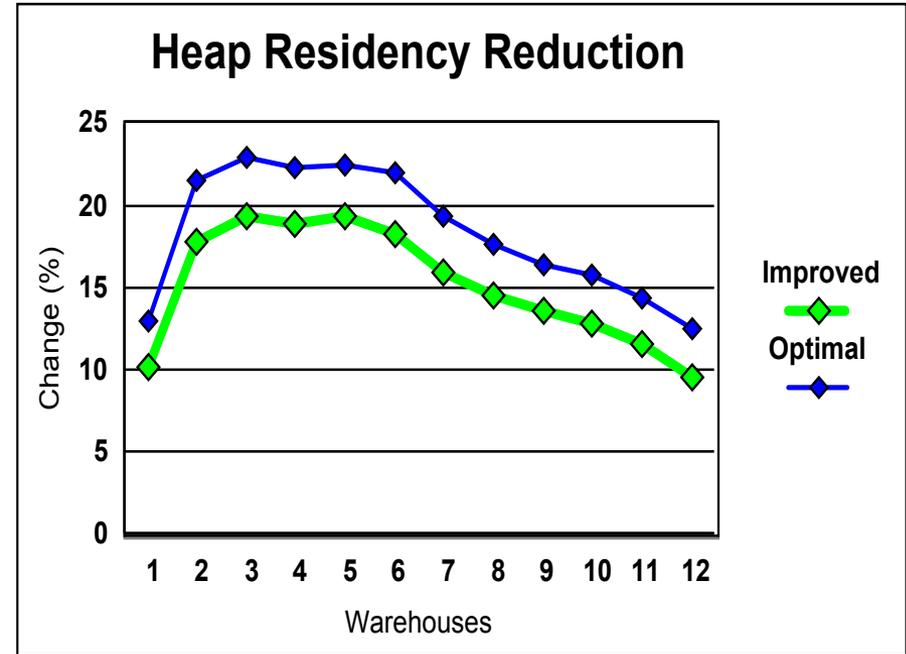
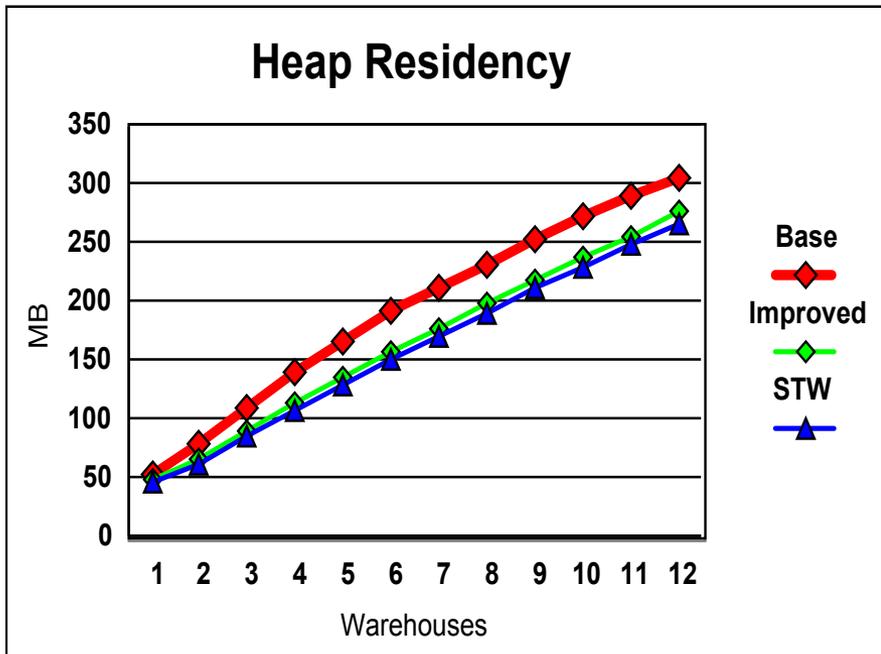
Results - Throughput Improvement

- SPECjbb. 6-way PPC. Heap size 448 MB
- 26.7% improvement (at tracing rate 1)



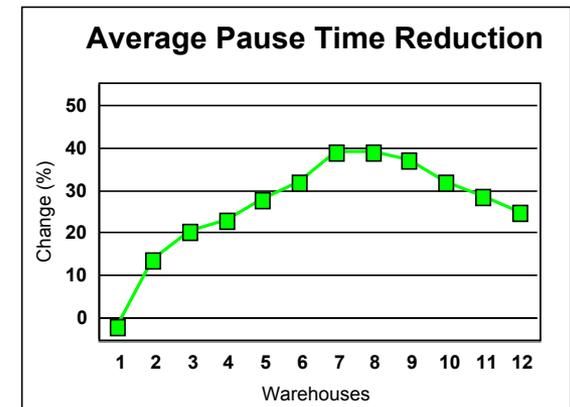
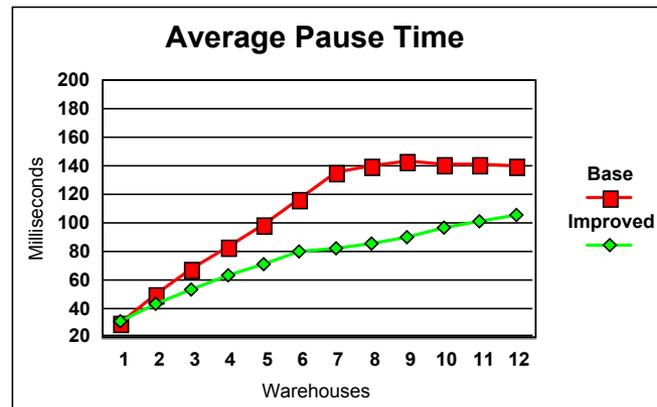
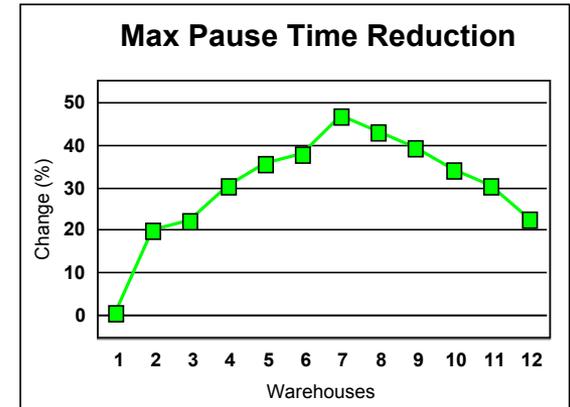
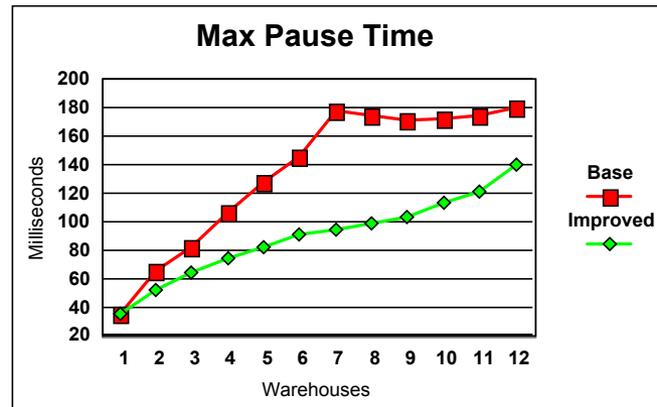
Results - Floating Garbage Reduction

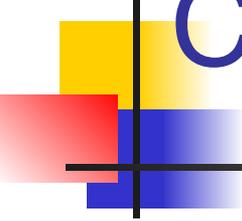
- SPECjbb. 6-way PPC. Heap size 448 MB
- 13.4% improvement in heap residency (at tracing rate 1)
- Almost all floating garbage eliminated



Results - Pause Time Reduction

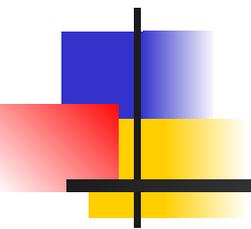
- SPECjbb. 6-way PPC. Heap size 448 MB
- 33.3% improvement in average pause
- 36.4% improvement in max pause





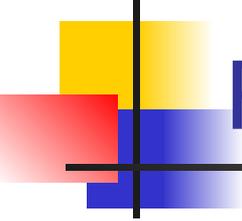
Conclusions

- Two improvements on the popular mostly concurrent algorithm
 - Reduce repetitive work (don't trace through dirty cards)
 - Reduces number of dirty cards (undirty cards with no traced objects)
- Substantial improvement
 - Improved benchmark throughput by 26%
 - Almost eliminated floating garbage (heap residency reduced by 13%)
 - Reduced average pause time by 33%



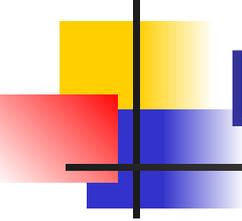
Reducing Pause Times for the Mostly Concurrent Collector

Endo-Taura 2002



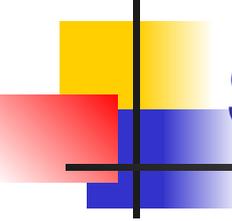
Reducing Pause Times

- Pause time is determined by the final stop-the-world phase in which cards are finally cleaned while the program is halted.
- Manage work to achieve short-pauses:
 - Limit the number of dirty cards
 - If the halt becomes long, let the program threads resume.



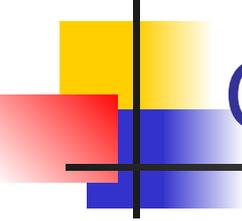
Limit the number of dirty cards

- If the number of dirty cards exceeds a predetermined threshold, immediately scan one card.
- Advantage: limited number of cards will be scanned during the final phase.
- Disadvantage: a card may be rescanned repeatedly a large number of time.



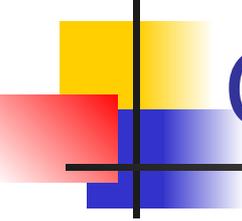
Stop a Halt

- If a halt scans too many objects, then go back to concurrent mode.
- Advantage: Limited pause time.
- Disadvantage: threads may exhaust heap before collection ends.
 - Triggering becomes more complicated.



Conclusion

- Ideas may be used to reduce the pause times considerably.
- Throughput may be harmed.



Conclusion – Mostly Concurrent

- Most of the work done concurrently with program.
- Program 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.
- One of the most popular commercially
 - Used (at least by) IBM, SUN (Oracle), BEA.