Algorithms for Dynamic Memory Management (236780)

Lecture 11 ++

Lecturer: Erez Petrank

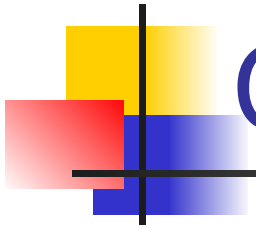
Topic last week

- Allocation Techniques
- Parallel Garbage Collection

Topics this week

- Cache conscious memory management
- Real-time memory management

Cache Conscious Memory Management

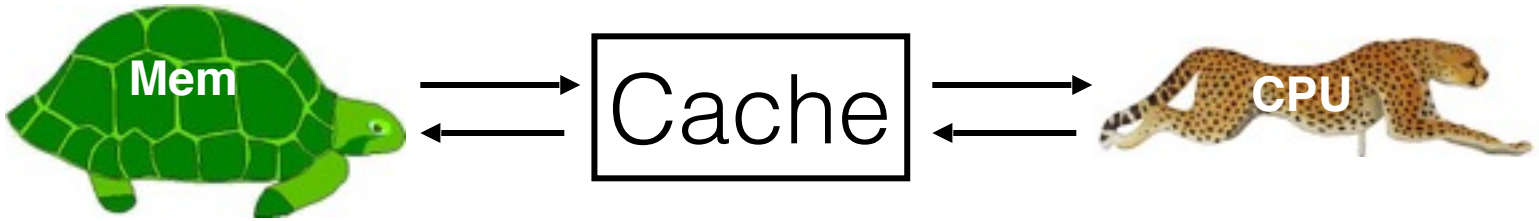


Cache-consciousness

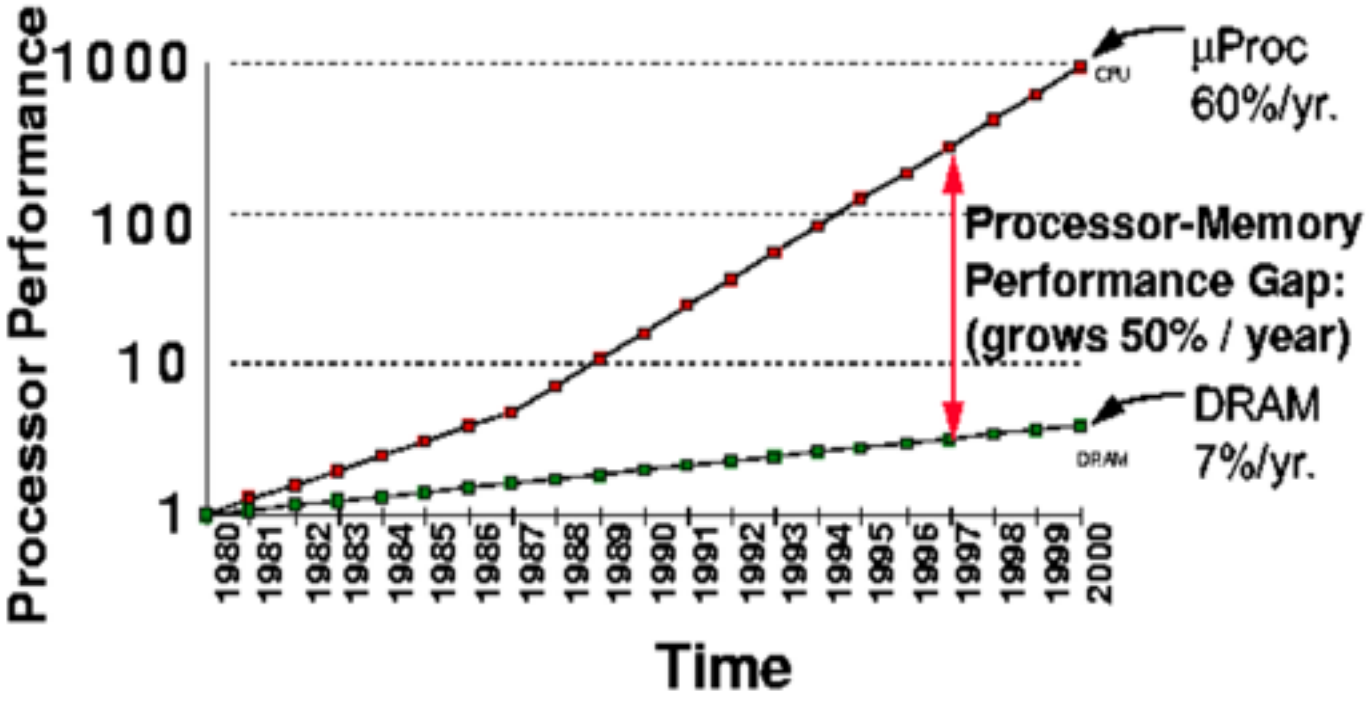
- Cache systems overview
- Improving program behavior via GC
  - [Chilimbi-Larus 98]
  - [Huang, Blackburn, McKinley, Moss, Wang, Cheng 04]
- Improving GC behavior:
  - Boehm's prefetching and lazy sweeping;
- Limitations on the ability to improve data placement [Petrank-Rawitz 02].

Computers today

- Memory speed falls behind processor speed.
- Solution: a fast cache between memory and CPU.
- Implication: significant impact on program efficiency.

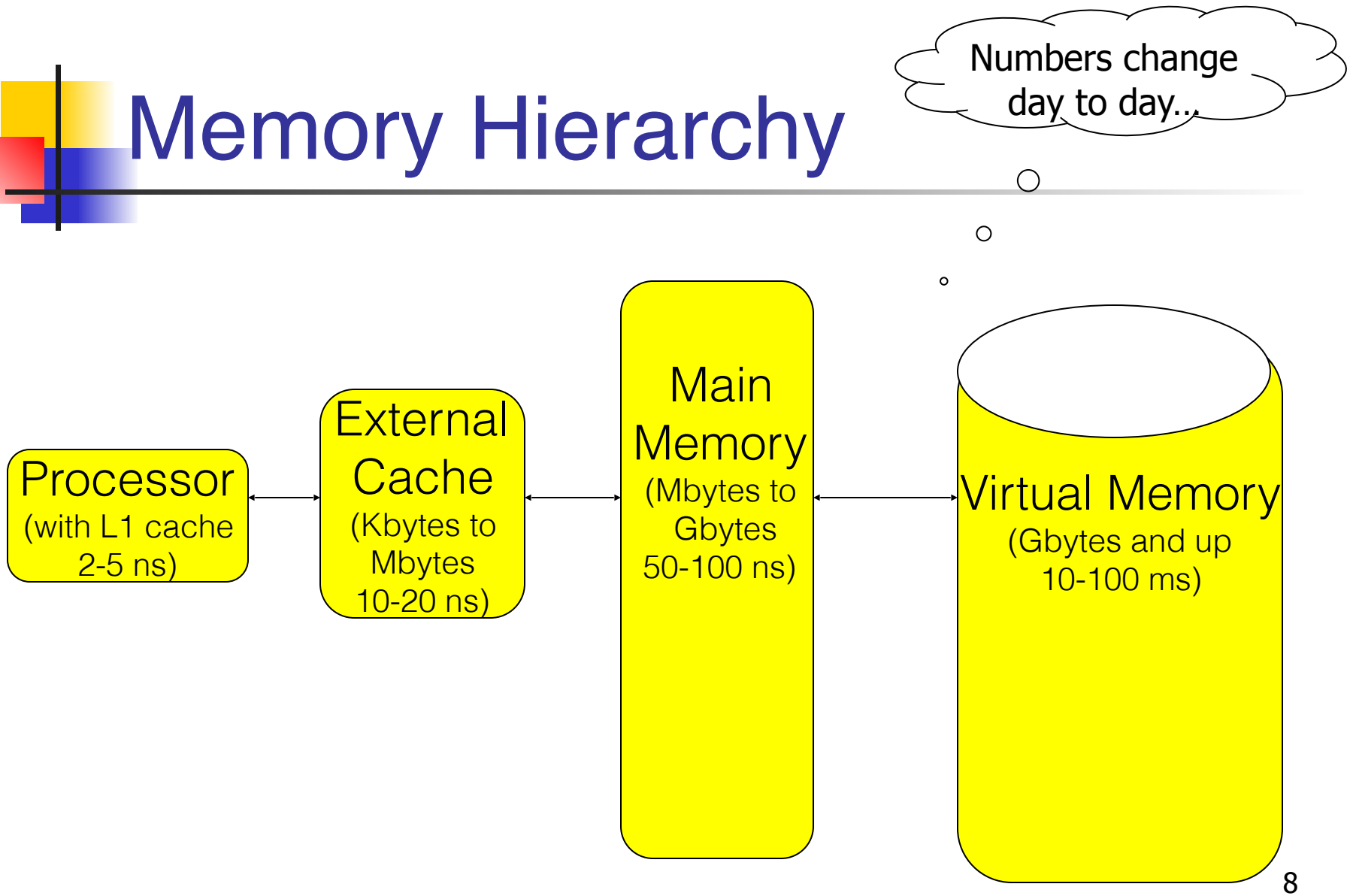


Processor-DRAM Gap (latency)



4

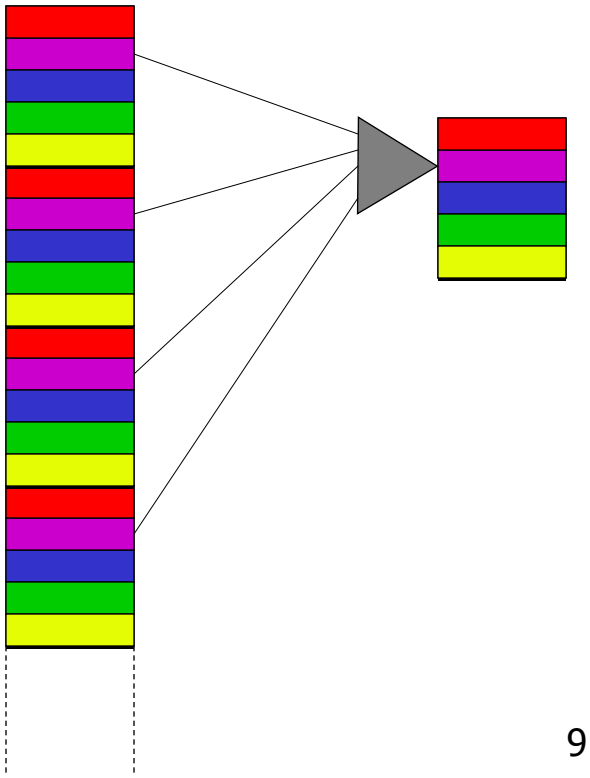
Memory Hierarchy



Cache structure

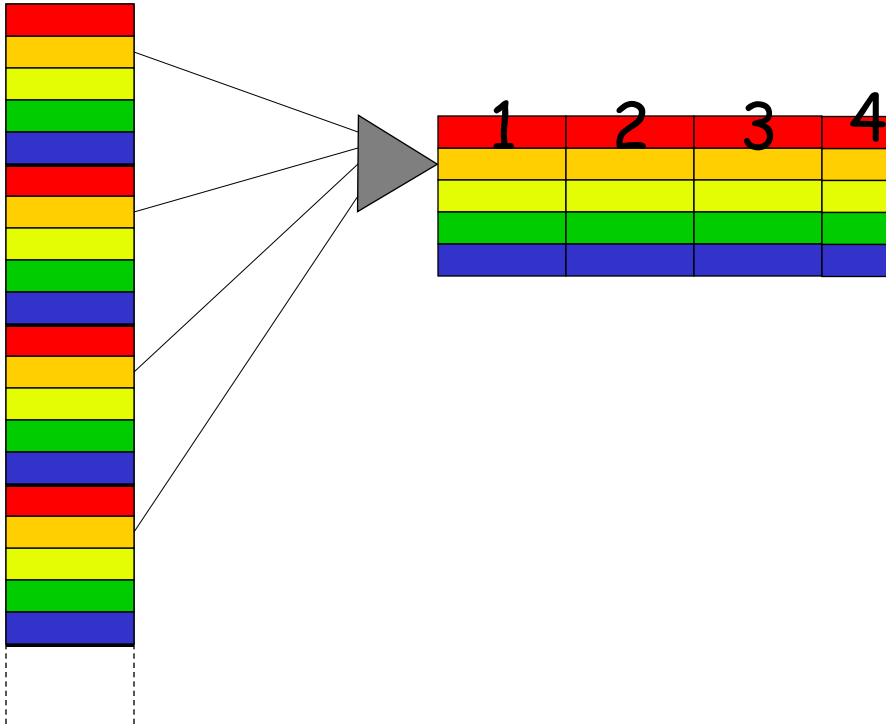
- Large memory divided into lines.
- Small cache - k blocks.
- Mapping of memory lines to cache lines.
- Cache hit.
- Cache miss.

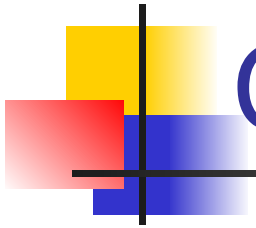
Direct mapping



Associative cache:

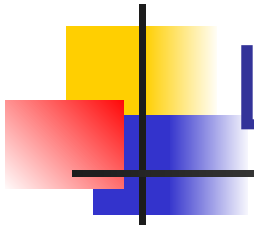
- t-way Associative caches:
- t·k lines in cache, k sets of t lines.
- memory line mapped to a set.
- Inside a set: a replacement protocol.





Cache Architectures

- Unified cache vs. splitted data/instructions cache
- Cache's locations:
  - Primary – on die
  - Secondary (more than one level is possible) – usually packaged in a separate chip
- Relevant parameters:
  - Size
  - Number of lines
  - More: write strategy, etc.

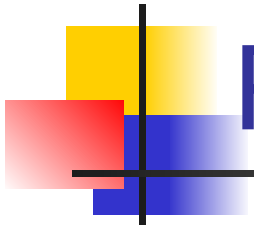


Line Size

- Ranges between 16-128 bytes
- Large lines:
  - Reduce miss rate if a program has good spatial locality
  - Tougher miss penalty (fetch time)
  - Less blocks – easier handling.

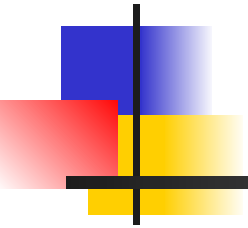
General Ways to Improve Cache Performance

- Hardware:
  - larger cache, more cache levels (use L2, L3).
- Software:
  - Write wiser algorithms.
  - Data arrangement to reduce hits.
  - Prefetching lines from memory.
- System: match parameters to cache.



Placing Data Appropriately

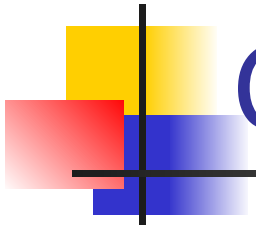
- [Chilimbi and Larus 98]
  - Place objects with high temporal affinity near each other
  - Idea: they will share a cache line.
  - Two misses become one.
- [Calder et al 98]
  - Do not let objects with high temporal affinity collide in cache.
  - Idea: reduce collision misses.
- [Huang, Blackburn, McKinley, Moss, Wang, Cheng 04] coming up…



The Garbage Collection Advantage: Improving Program Locality

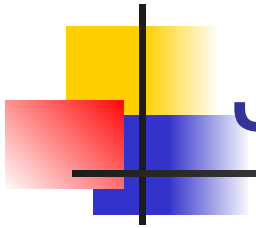
Huang, Blackburn, McKinley, Moss, Wang, Cheng

OOPSLA 2004



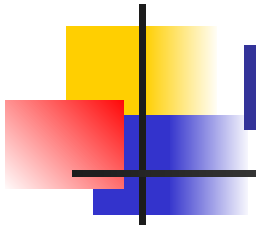
General Idea

- Copying GC May rearrange objects to improve performance.
- Idea: on-line detect which pointers are “hot”.
- During collection, let hot pointers have their descendants close-by.



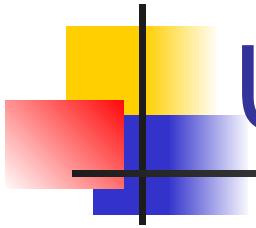
Java Virtual Machine

- Java code is translated into bytecode (javac).
- The JVM runs the bytecode.
- Initially: interpreter. Now: Just In Time (JIT) compilation.
- During the run, the JVM decides which methods to compile into native code and what degree of optimization to use.



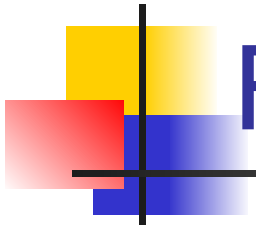
Identifying Hot Pointers

- The Jikes RVM identifies hot methods using time-driven sampling and recompiles them with higher optimization levels.
- Back to the current work:
  When a method is detected hot, an additional mechanism marks pointers accessed in this method as hot.



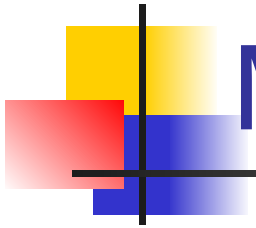
Using the Info During GC

- While scanning an object, enqueue its hot descendants on the hot queue and its cold descendants on the cold queue.
- Scan until hot and cold queues are empty, always prefer to take an object from the hot queue.



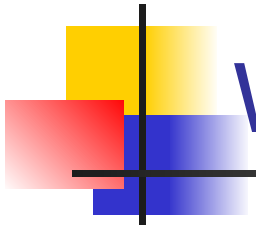
Further Optimizations

- Decay heat to respond to phase changes.
  - Hot methods should be periodically caught by the sampler. If they are not – the heat decays.
- Exclude cold code-blocks from reordering analysis using Jikes' static analysis.
- Group together objects of hot classes in a separate space.



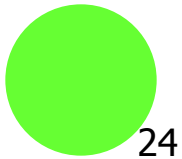
Measurements

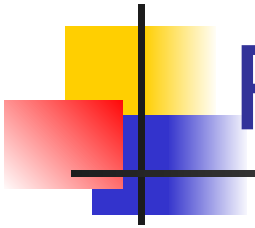
- Overhead of reordering analysis is ~2%.
- Most bm's vary by ~4% due to copy order.
- Four programs are more sensitive (up to 25%)
- Methods compared: BFS, DFS, and partial DFS, using the first two children.
- Online object reordering matches or improves upon the best class-oblivious ordering.



What about optimal arrangements?

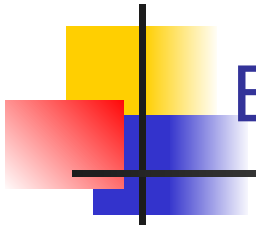
We will see later that determining the best placement is extremely difficult…





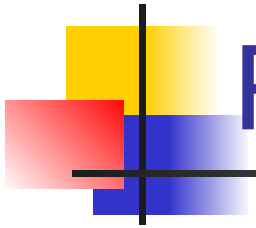
Roadmap

- Cache systems overview
- Improving program behavior via GC
  - [Chilimbi-Larus 98]
  - [Huang, Blackburn, McKinley, Moss, Wang, Cheng 04]
- Improving GC behavior:
  - Boehm's prefetching and lazy sweeping;
- Limitations on the ability to improve data placement [Petrank-Rawitz 02].



Boehm: Reduce Cache Misses for Tracing GC

- During the mark phase: Prefetch on Grey.
- During the sweep phase: Lazy Sweeping.



Prefetch(x)

- Hint to hardware: start fetching data at addr x into the cache.
- Never stalls the processor, but can be ignored.
- When successful, the next load will hit the cache.
- When unsuccessful, next load may miss the cache, but program still executes correctly.
- Most platforms provide a prefetch instruction.

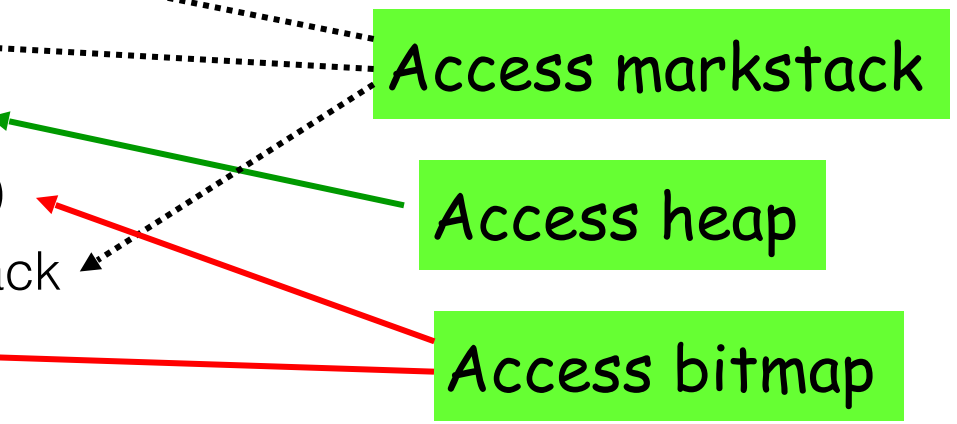
Dealing with the Mark Phase

Recall mark phase:

Ensure that all objects are unmarked.

Mark and Push to makrstack addresses of all objects pointed to by a root.

```
while (markstack not empty)
    pop object g
    For each pointer p in g
        if ( obj(p) not marked )
            push p to markstack
            mark obj(p)
```

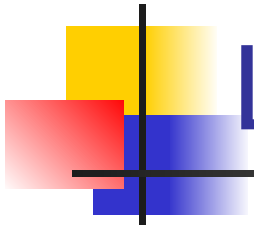


Prefetching gray objects

An observation (by measurements): ~1/3 of marker time is initial load of first pointer in an object.

The “prefetch” instruction.

As soon as object g is pushed to markstack a prefetch is issued on the first cache line of g.



Lazy Sweeping technique.

- Recall lazy sweep
- Initial motivation: lazy sweep reduces pause times.
- However, lazy sweep also reduces page faults and cache misses.
- A cache line is reallocated shortly after being swept. Thus, two cache misses may become one.

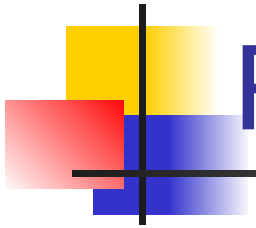
Measurement details

Table 1: Pentium II/500 Relative Performance

Benchmark	Mark Time	Sweep Time	Eager Sweep Slowdown	Prefetch Speedup
gc_bench_java	39%	3%	0%	11%
gc_bench	49%	3%	0%	13%
holes_gc_bench	57%	12%	7%	17%
ptc	27%	0%	0%	4%
ghostscript	44%	5%	5%	5%
incremental_ghostscript	39%	9%	17%	4%
large_ghostscript	8%	6%	3%	1%

Table 2: HP PA-RISC Relative Performance

Benchmark	Mark Time	Sweep Time	Eager Sweep Slowdown	Prefetch Speedup
gc_bench	45%	3%	-1%	11%
holes_gc_bench	40%	36%	34%	9%
ghostscript	26%	4%	3%	8%

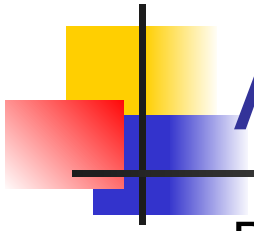


Roadmap

- Cache systems overview
- Improving program behavior via GC
  - [Chilimbi-Larus 98]
  - [Huang, Blackburn, McKinley, Moss, Wang, Cheng 04]
- Improving GC behavior:
  - Boehm's prefetching and lazy sweeping;
  - Zorn's recommendations.
- Limitations on the ability to improve data placement [Petrank-Rawitz 02].

The Hardness of Cache Conscious Data Placement

[Petrank-Rawitz 2002]

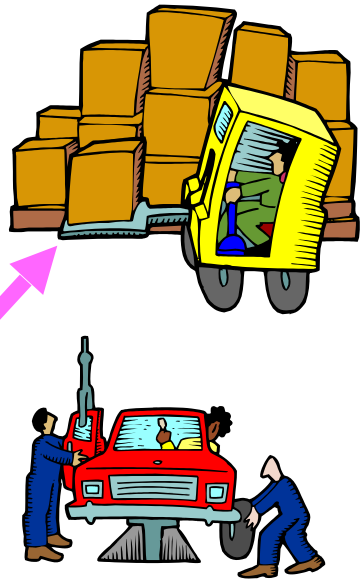


Agenda

- Background & motivation
- The problem of cache conscious data / code placement is:
  - extremely difficult
  - in various models
- Positive matching results (weak…).
- Some proof techniques and details
- Conclusion

What can we do to improve program cache behavior?

- Arrange code / data to minimize cache misses
- Write cache-conscious programs



We now concentrate on the first.

How do we place data (or code) optimally?

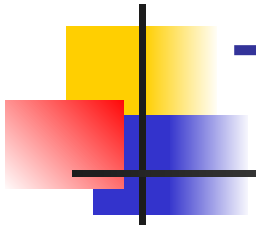
- Step 1: Discover future accesses to data.
- Step 2: Find placement of data that minimizes the cache misses.
- Step 3: Rearrange the data in memory.
- Step 4: Run program.

- Some "minor" problems:
  - In Step 1: We cannot tell the future
  - In Step 2: We don't know how to do that

Step 1: Discover future accesses to data

- Static analysis.
- Profiling.
- Runtime monitoring.

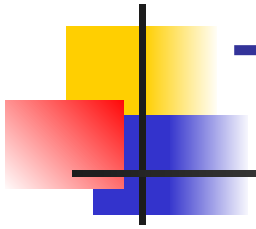
We will next show:
Even if future accesses are known exactly,
Step 2 (placing data optimally) is extremely difficult.



The Problem

- Input: a set of objects $O=\{o_1,\ldots,o_m\}$, and a sequence of accesses $\sigma=(\sigma_1,\ldots,\sigma_n)$.
  E.g. $\sigma = (o_1,o_3,o_7,o_1,o_2,o_1,o_3,o_4,o_1)$.
- Solution: a placement, $f:O\rightarrow N$.
- Measure: number of misses.

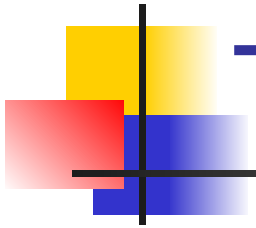
We want: placement of $o_1,\ldots,o_m$ in memory that obtains minimum number of cache misses (over all possible placements).



The Results

Can we (efficiently) find an optimal placement?

No! Unless, P=NP.



The Results

Can we (efficiently) find an “almost” optimal placement?

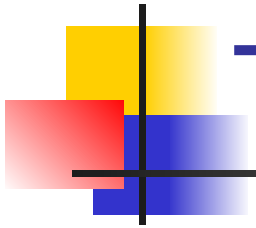
Almost = # misses ≈ twice the optimum

No! Unless, P=NP.

Can we (eff.) find “fairly” optimal placement?

Fairly = # misses ≈ 100 times the optimum

No! Unless, P=NP.



The Results

Can we (eff.) find a “reasonable” placement?
reasonable = # misses ≈ log(n) the optimum

No! Unless, P=NP.

Can we (eff.) find an “acceptable” placement?
Acceptable = # misses ≈ $n^{0.99}$ times the optimum

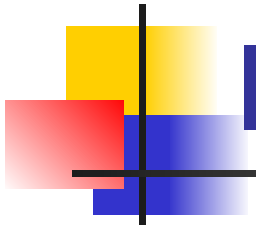
No! Unless, P=NP.

The Main Theorem

Let ε be any real number, $0<\varepsilon<1$.
If there is a polynomial time algorithm that finds a placement which is within a factor of $n^{(1-\varepsilon)}$ from the optimum, then P=NP.

(Theorem holds for caches with > 2 lines)

[Rahman Lavaee 2016]: This is also true for 1 line (relevant for paging)

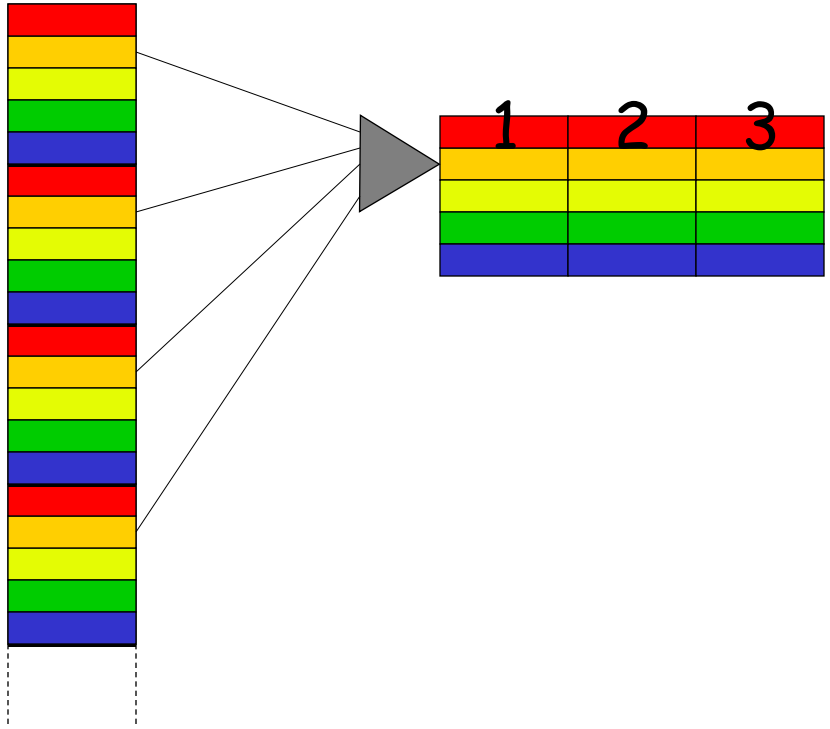


Implications:

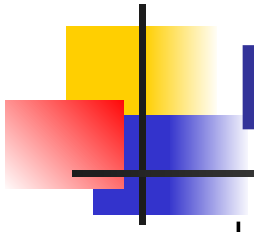
- We cannot hope to find an algorithm that will always give a good placement.
  We must use heuristics.
- We cannot estimate the potential benefit of rearranging data in memory to the cache behavior.
  We can only check what a proposed heuristic does for common benchmarks.

Extend to t-way Associative Caches

- t-way Associative Caches:
- t·k blocks in cache, k sets, t blocks in a set.
- memory block mapped to a set.
- Inside a set: a replacement protocol.



Theorem 2: same hardness holds for t-way associative cache systems.



Result is “robust”

- Holds for a variety of models. E.g.,
  - Mapping of memory block to cache is not by modulus,
  - Replacement policy is not standard,
  - Object sizes are fixed, (or they are not),
  - Objects must be aligned to cache blocks, (or not),
  - Etc…

A Simple Observation

- Input: Objects $O=\{o_1,\ldots,o_m\}$, and accesses $\sigma=(\sigma_1,\ldots,\sigma_n)$.
- Any placement yields at most n cache misses.
- Any placement yields at least 1 cache miss.
- Therefore, any placement is within a factor of n from the optimum.
- (Recall: a solution within $n^{(1-\varepsilon)}$ is not possible.)

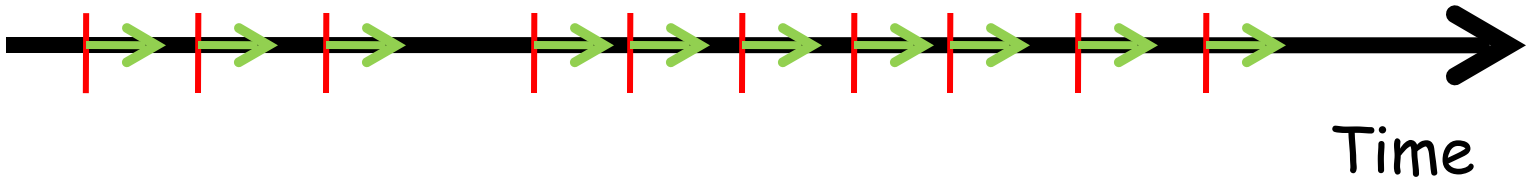
Real-Time GC

Agenda

- Real-time requirements
- What's bad with Baker's collector ? (Work-based versus time-based collection)
- IBM's Metronome
- Microsoft's concurrent real-time collectors
- Schism

Real-Time

- Respond on time to events.
  - Flight control, telecommunication, audio processing, stock commerce, network switches, etc.
- Predictable response more important than efficiency.
  - But it would be nice not to act slowly.
  - Main parameters: latency of response, throughput overhead.



Real-Time for Managed Code

- Historically, real-time was written in low-level code, requiring some sort of verification.
  - Difficult to create meaningful software.
- Garbage collection is a major obstacle for C# & Java.
  - So are Jitting and dynamic class loading.
- Real-Time Specification for Java (RTSJ): real-time threads do not use the garbage collected heap.
- With time, real-time programs get more complicated.
  - A reliable and scalable development environment is needed.

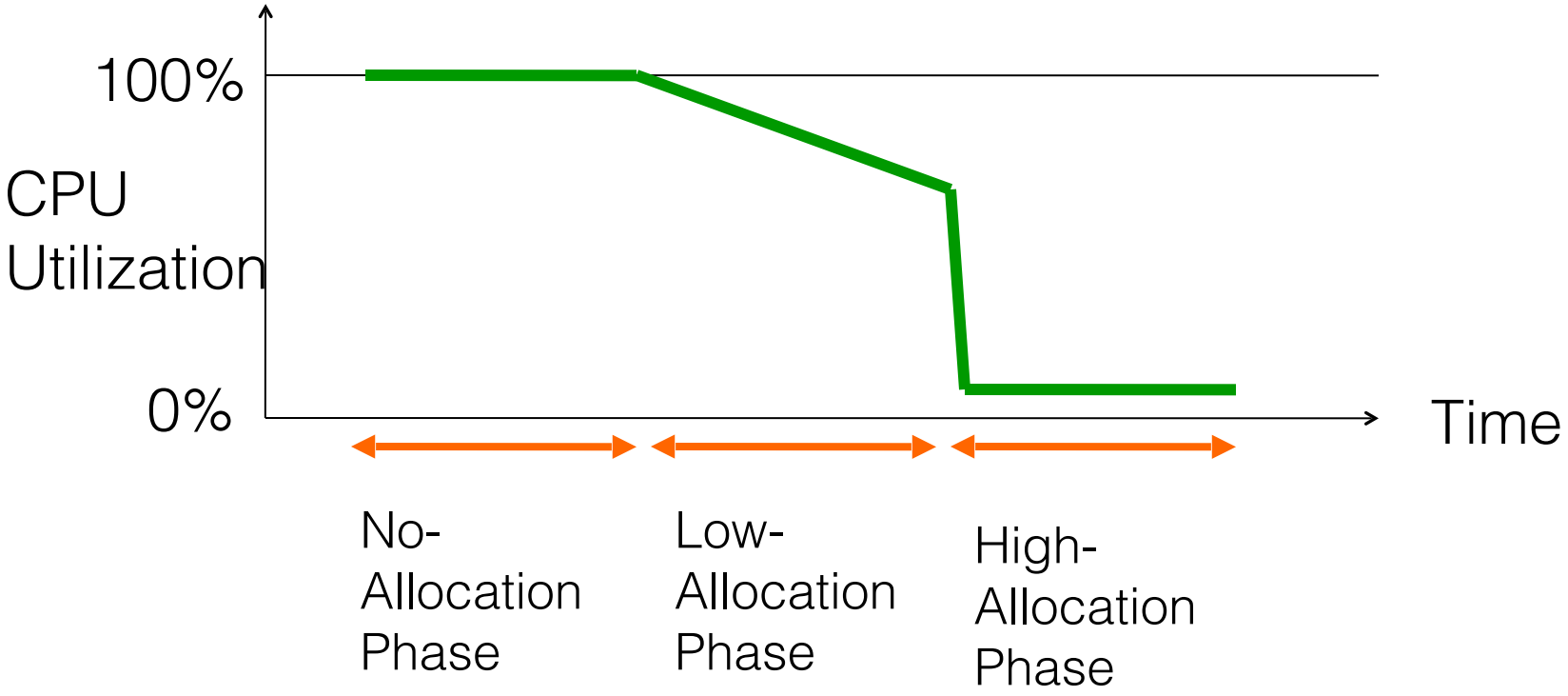
Real-Time Garbage Collection

- Must be either incremental or concurrent.
- Pauses must be bounded !
- Pauses should be evenly distributed to allow mutator to respond.
- Memory overhead should be within provided resources
  - Cannot just allocate without collecting
  - Cannot ignore fragmentation.

Work-Based versus Time-Based Triggering

- Baker designed a work-based incremental copying collector for real-time purposes.
- The idea: perform some incremental collector work during each allocation.
- Advantages:
  - Fairness: a thread that allocates a lot pays more.
  - Termination: The work of the collector ends before heap exhausted
- Disadvantage:
  - In program phases that allocate frequently, mutator gets a low percentage of the CPU.

Mutator CPU Utilization

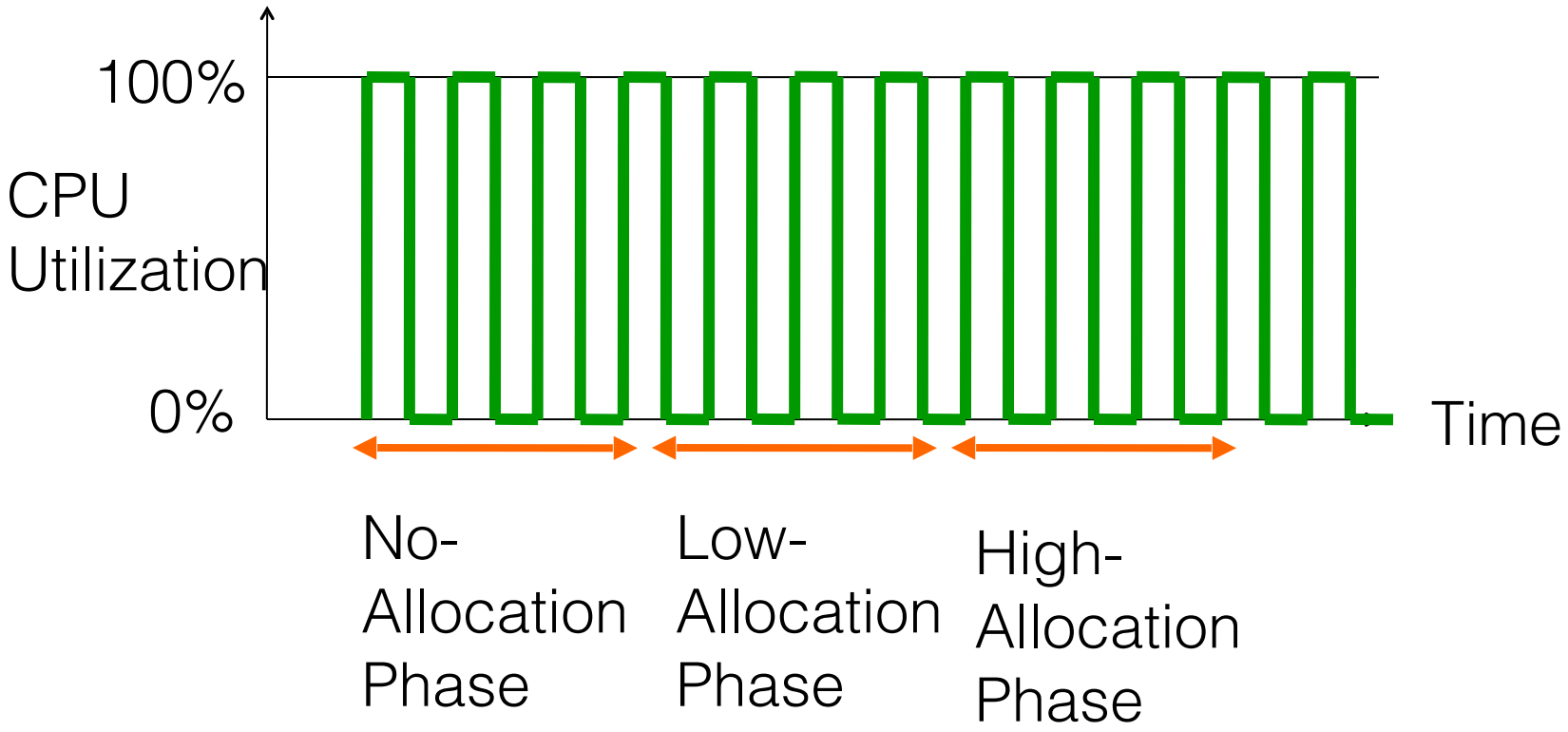


During high allocation phase, a program may miss a real-time deadline !

Time-Based Triggering

- A time-based collector provides a percentage of the time to the collector.
- E.g., mutator runs for 1ms, GC for 2ms, repeatedly.
- The idea: mutator guaranteed to get 1ms every 3 ms.
- Advantages:
  - Predictable response.
- Disadvantage:
  - No guaranteed termination. Must ask the user to supply parameters (such as rate of allocation and max live space)
    - Parallelism may change rate of allocation…
  - Required mechanism for stopping threads on time and switching to collection.

Time-Based Collectors

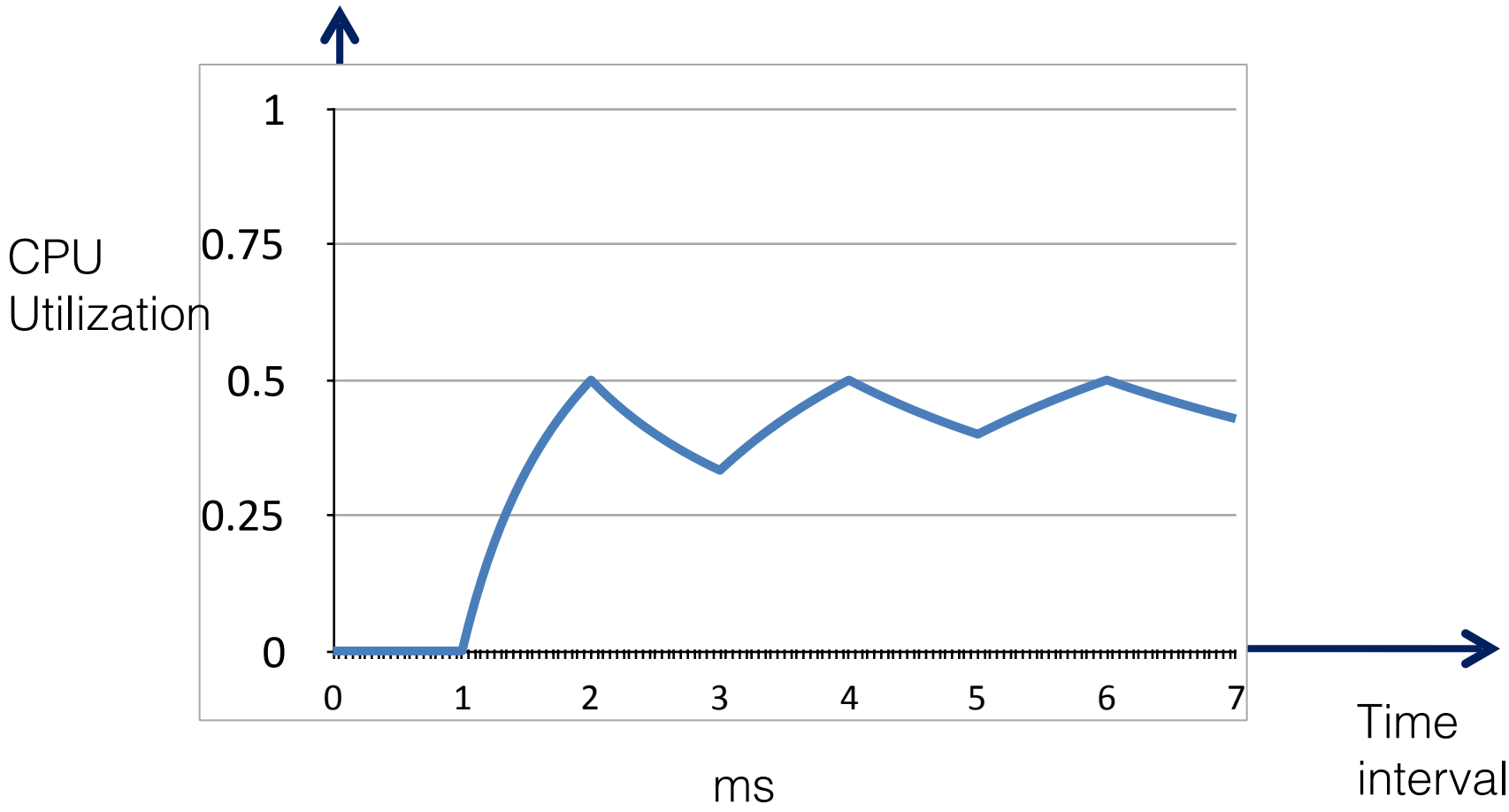


During any (sufficiently long) phase, a program gets a fixed percentage of the CPU.

Minimum Mutator Utilization

- Note that we said: “sufficiently long phase”.
- A more informative measure is the minimum mutator utilization graph (MMU).
- X-axis: time intervals
- Y-axis: given a time interval length, what is the worst mutator CPU utilization during the run.
- Example, if run is as in previous graph: collector and mutator get 1ms alternating.

The MMU Graph



Minimum Mutator Utilization

- Information that can be obtained:
  - Maximum pause time (how?)
  - For each possible time interval: what is the fraction of CPU time that the mutator is guaranteed to get in this interval length.
- A real run will not have such a simple line. E.g., the collection is not active all the time, and alternation ends, leaving the CPU to mutator.
  - If collection occurs half of the time, how will the above MMU graph change?

IBM's Metronome

David Bacon, Joshua Auerbach, Perry Cheng, Dave Grove, Mike Hind

(2003-2009)

Overview

- A real-time garbage collector for Java.
- Initially (2003) designed for a uniprocessor, thus, incremental.
- Time-based triggering.
- Mark-sweep (incremental) collection.
- Infrequent partial compaction to avoid fragmentation.
- Extensions include generations, parallelism, and concurrency.

Allocation

- Segregated free-list allocator, block-oriented:
  - Fixed-size pages
  - Each page only allocates fixed-size spaces.
- Objects allocated in smallest block that fits
- Sizes grow geometrically, limiting the inner fragmentation according to the growth factor (1+ρ).

Incremental Mark-Sweep

- Snapshot-oriented.
- Write-barrier saves overwritten (old values of) pointers in buffers.
- Buffered pointers become roots.
- Mark phase is used to fix pointers of objects that have moved (to be discussed later).

Handling Fragmentation

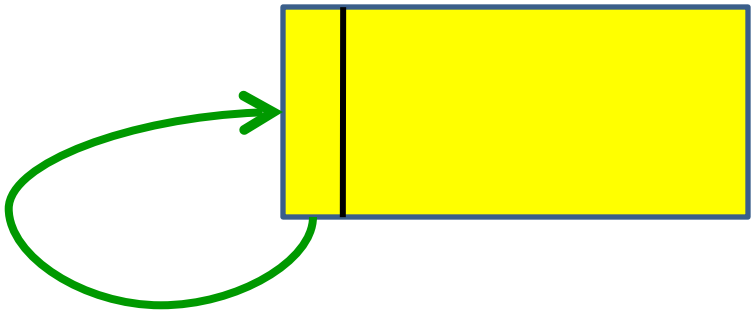
- A real-time garbage collector should handle worst-case scenarios.
- It must be able to handle fragmentation, otherwise, program will fail on space limit.
- Partial compaction: when fragmentation detected, move some of the objects.
- Hopefully, this happens seldom.
- Usually, no attempt is made to preserve objects order.

Partial Compaction

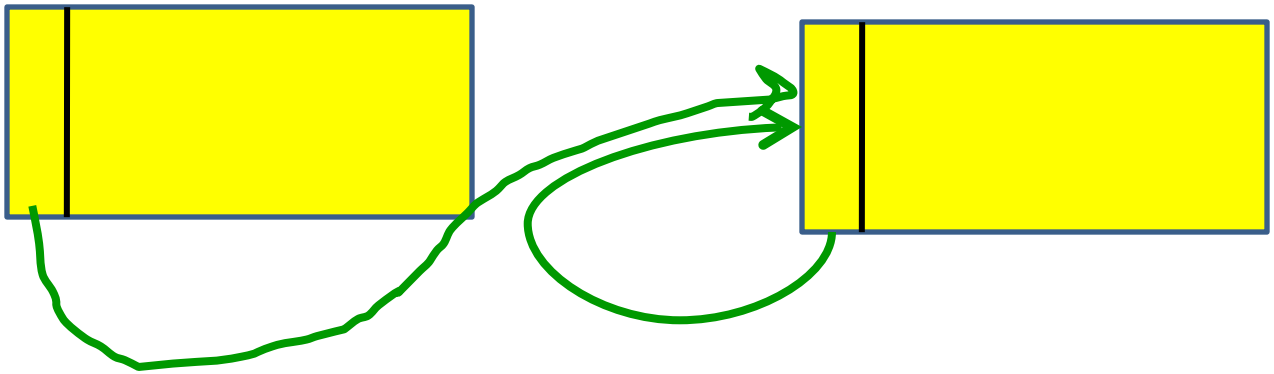
- Problem: when moving an object, a new copy is created.
  - At some point all program threads should start accessing the new copy.
  - If one thread writes the old copy after other threads read the new one, the write will be lost!
  - This switch is simpler on a uniprocessor.
- Main idea: indirect access to objects.
- Each object contains in its header a pointer either to itself or to its new copy.

Indirect Access (Brooks)

Case I: object not copied



Case II: object has a newer copy.



Object Access

- Each object access (read-barrier) goes through the indirect pointer.
- To move an object, the collector
  - Creates a new copy ;
  - Copies all data from old to new copy;
  - Sets the forwarding pointers (in both objects);
- The above operations must appear atomic to other threads (why?).
- Uniprocessor (or any non-concurrent) solution: CPU is not yielded in the middle of a move.

Reclamation of Old Copies

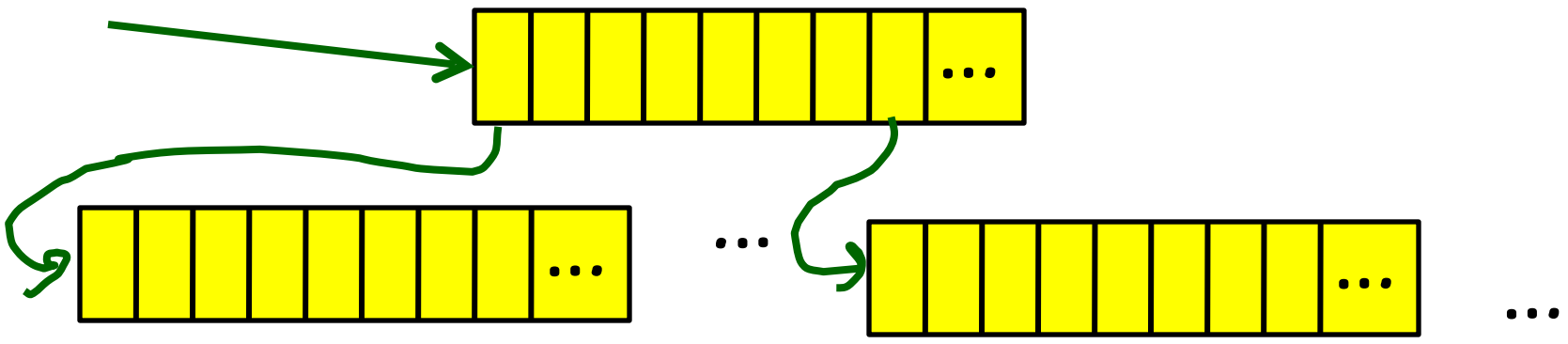
- An old copy can be reclaimed after nothing points to it.
- The mark procedure updates all reachable pointers.
- After that, old objects are not reachable and are automatically reclaimed during the next collection.

Picking Objects to Relocate

- Fragmentation indicators:
  - Free pages fall below limit for performing a GC
  - Dead slots are not re-used for a GC cycle
- In practice: 2-3% of reachable data is moved, allows heaps of size < 2 x size of live data.

Large Arrays and Objects

- Large objects are impossible to move in a short quantum.
- Solution: Arraylets.
  - Each array broken into arraylets.



Arraylets

- Advantage:
  - Can move bounded-size pieces.
- Disadvantage:
  - More indirection: time and space overhead.

Triggering

- Triggering is a major technical issue.
- Time triggering: after application runs time t(app), collector runs time t(col).
- To guarantee termination, t(app)/t(col) is set according to allocation rate, live space, heap size, etc.
- The quantum t(col) is upper bounded by the response time of the application, it is also lower bounded by technical constraints
  - At time required for the atomic sections (scan the stack, move a large object) + time to stop threads, if made parallel.
- We do not get into the (tedious) formulas.

Microsoft's Real-Time Concurrent Garbage Collection

Filip Pizlo, Daniel Frampton, Gabi Kliot, Erez Petrank, Bjarne Steensgaard

Real-Time Garbage Collection

- Mtronome's response time within ~1ms (at 2007, now hundreds of μ-sec).
- Can we respond within 10 μ-sec and less?

Real-Time GC on Parallel Platforms:
What's Expected?

- Short (or no) GC pauses
  - support responsiveness.
- Program never waits for the collector
  - lock freedom support: GC does not add locks.
- Partial compaction
  - to avoid worst-case fragmentation.
- Low overhead and multiprocessor scalability

- This work was the first to provide all of these requirements simultaneously.

Improving Responsiveness

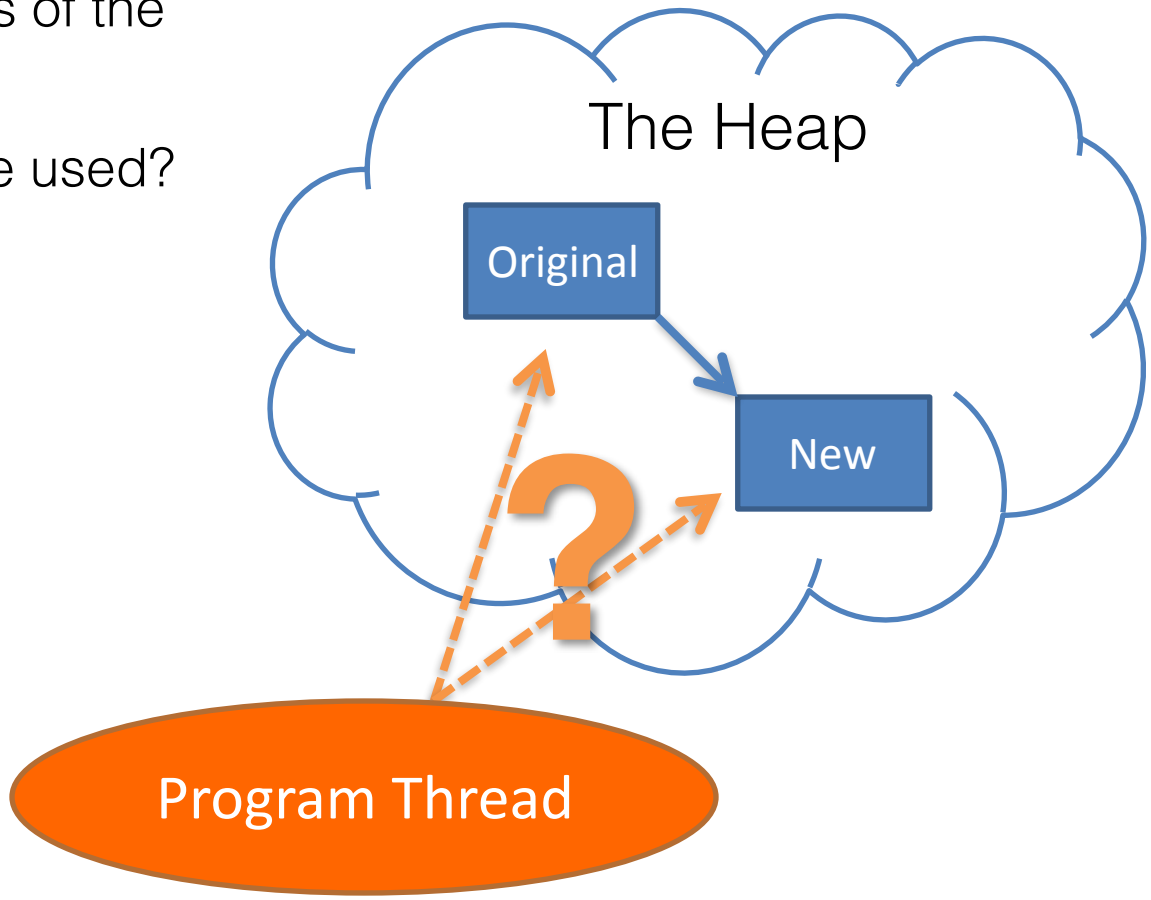
- How do we get responsiveness faster than Metronome?
- A simple idea: use concurrent GC.
- Run GC on one processor/core and the (parallel) program on the others.

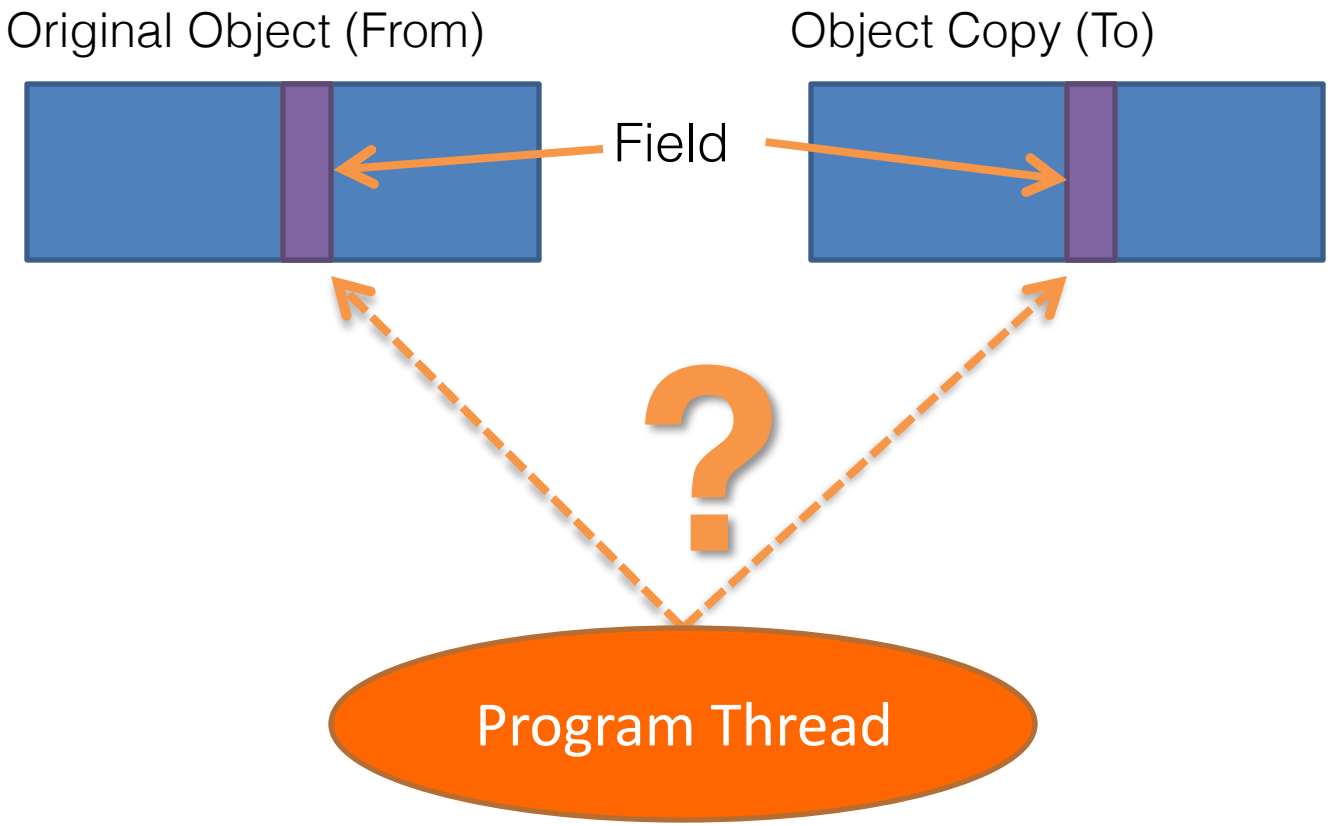
- Obtain high responsiveness.
- Problem: need new technique.

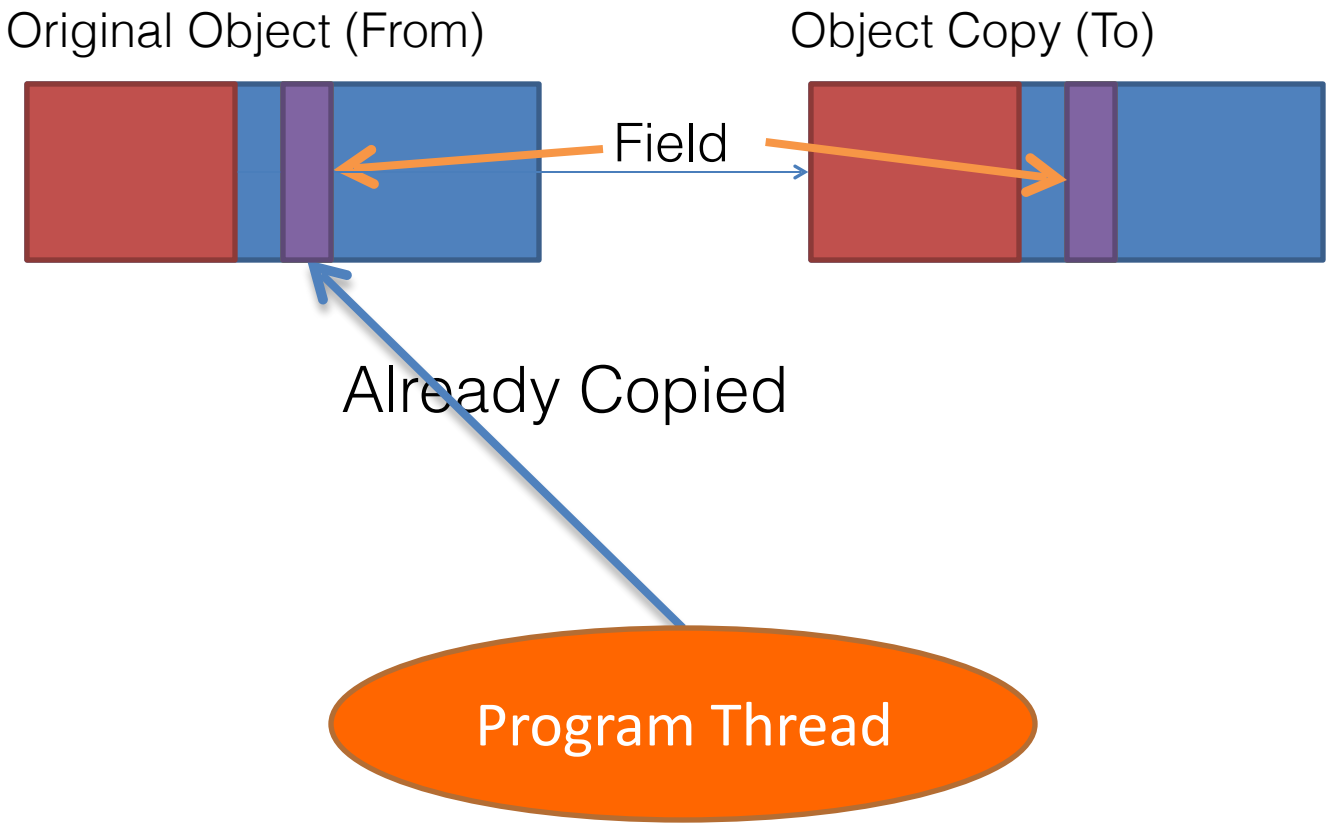
- Main problem: move objects concurrently

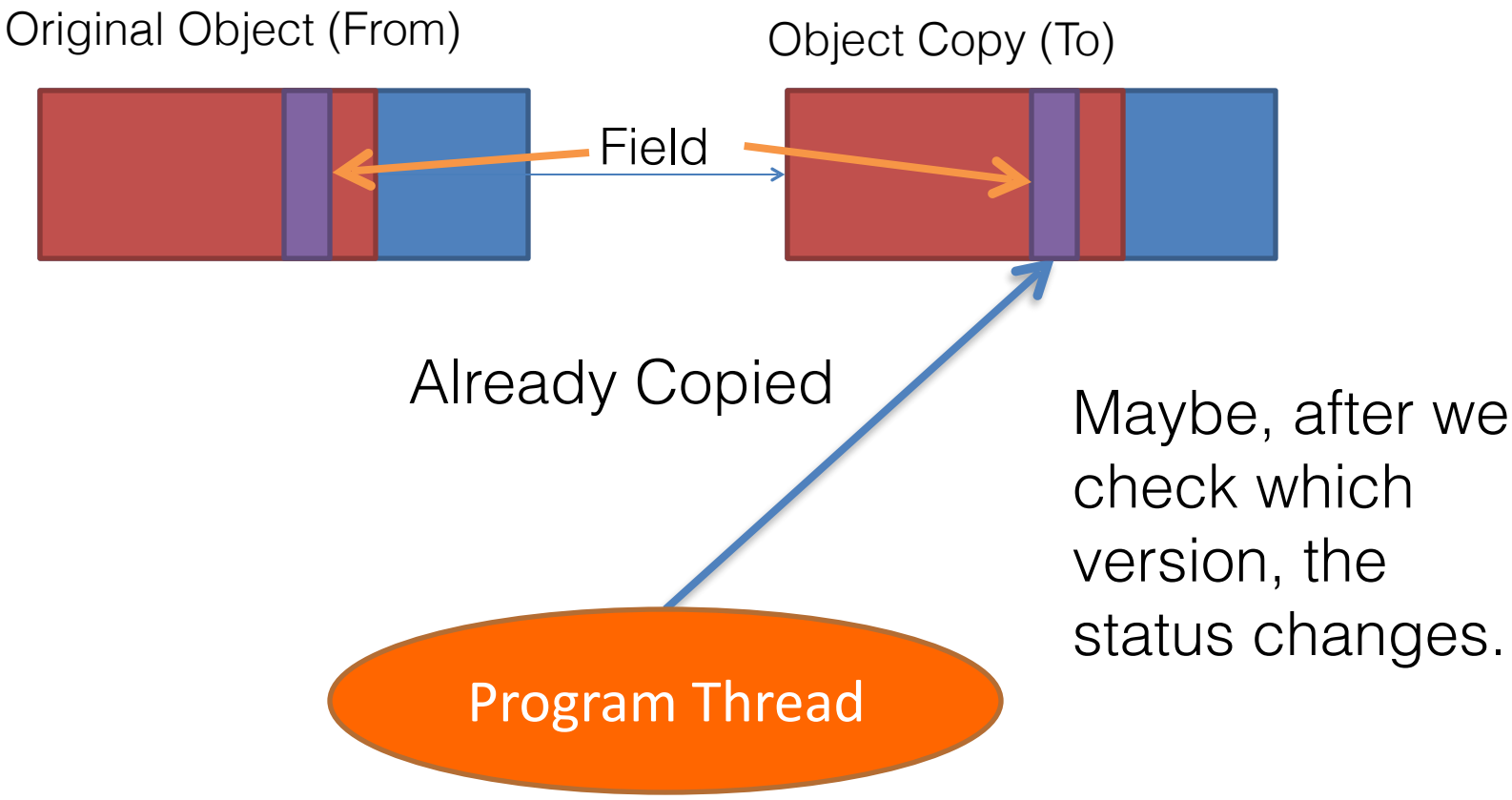
Moving Objects Concurrently

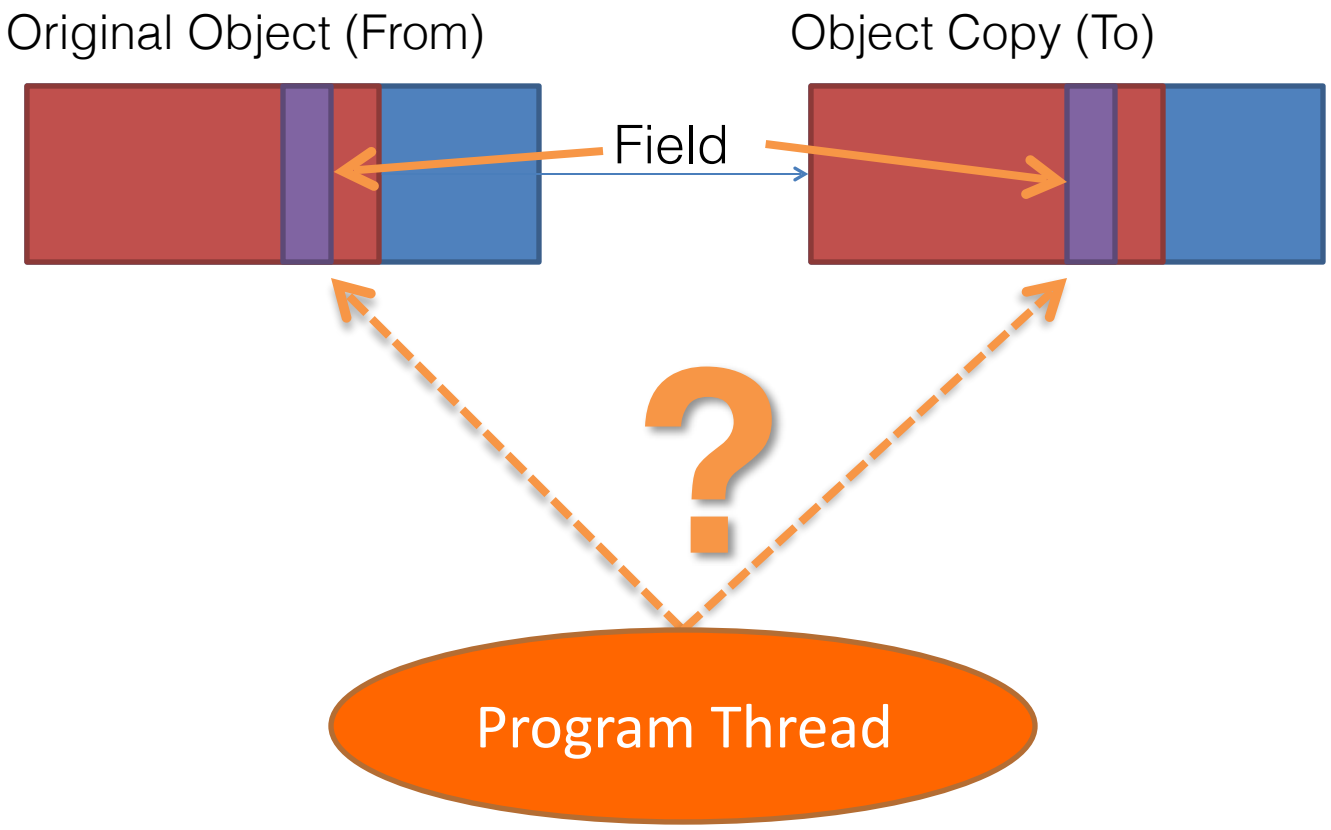
- The problem: two copies of the same object.
- Which version should be used?




Original Object (From)
Object Copy (To)
Field
?
Program Thread


Original Object (From)
Object Copy (To)
Field
Already Copied
Program Thread


Original Object (From)
Object Copy (To)
Field
Already Copied
Maybe, after we check which version, the status changes.
Program Thread



Need to atomically access two locations:
field and status.

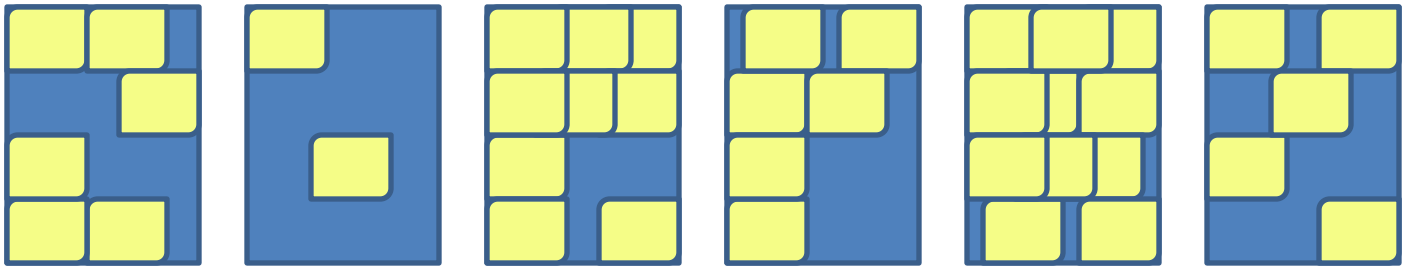
The Three Collectors

- Stopless: employs an intermediate long object.
- Clover: locks with extremely small probability.
- Chicken: aborts copying when program interferes.

- Concentrate on concurrently moving objects.

The Overall Picture

- Concurrent mark-sweep
  - Lock-free data structures & care for mark-stack overflow.
- Partial Concurrent Compaction
  - Move objects only when heap fragmented.
  - Evict fragmented pages.
  - Concurrently move objects using Stopless, Clover, or Chicken.
  - Fix pointers during next trace, or via special pass.
- Both tasks can run concurrently and in parallel



Chicken

- Idea: if program touches an object during the move then the move is aborted.

- Collector copies objects one by one in the background.
- After copying an entire object, if it was not modified, atomically mark it “moved”.
- Objects are copied as a whole.

- A program write starts by marking the object dirty.
- Then, checks if the object moved.
- Finally, it accesses it according to its determined location.

Chicken

- Advantage: no intermediate objects, no fall-back to locking, quick memory barriers.
- Disadvantage: may fail to move objects.

Conclusion

- Allowing development of real-time software on high-level languages is important.
- A major obstacle is garbage collection pauses.
- Need partial compaction, lock-freedom, highly responsive collector.
- We've discussed IBM's Metronome, Microsoft's concurrent solutions, and Schism.