

# Algorithms for Dynamic Memory Management (236780)

## Lecture 11

---

Lecturer: Erez Petrank



# Topic “last” week

---

- Cycle Collection (for Reference Counting)
- Non-Intrusive Compaction



# Topics this week

---

- The Compressor:
  - Parallel version
  - Concurrent version
- Cache conscious memory management

# The Compressor

(continued from last week)

# The Generic Task

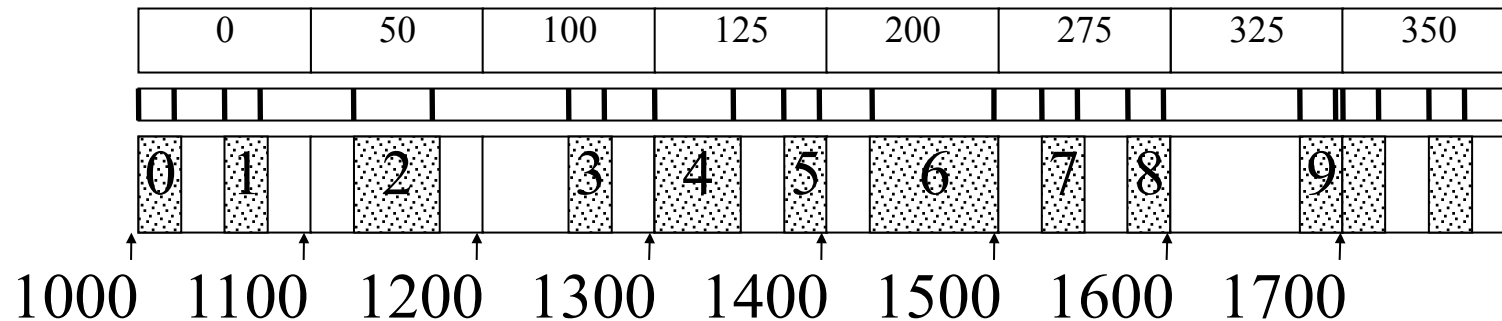
- Assume live objects are marked (i.e., the mark phase is done).
- The algorithm **moves** objects to one (or a small number of) areas in the heap
- Pointers are **modified** to reference the new locations.

# Compressor - Overview

- Compute locations of new objects (markbits pass)
- Move objects + fix their pointers (heap pass)
- Assume a mark-bit exists providing a bit for each beginning of heap object, and each end.
- Typically, such a bit-map is produced by a mark-sweep collector.
- A pass over this mark-bit vector, allows computing a target address for each object.

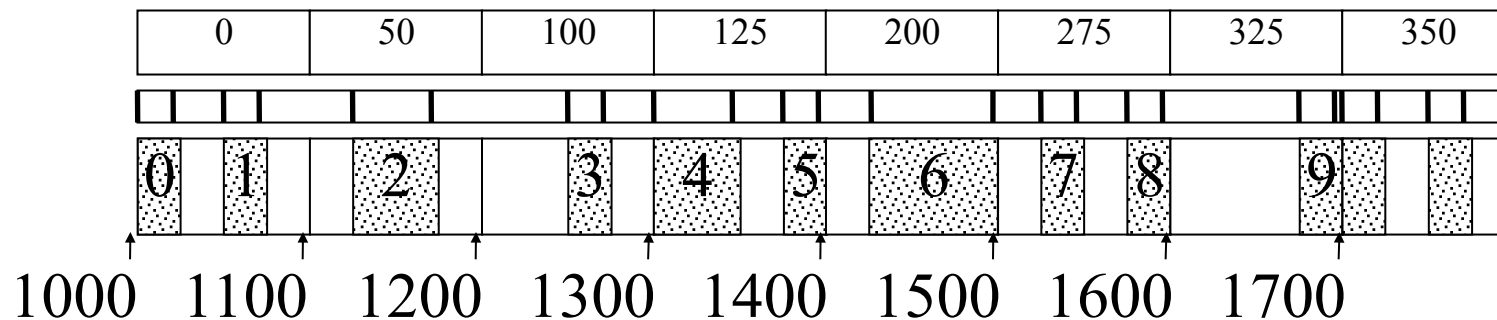
# Compressor - Overview

- Use a table to store relocation info succinctly.
- Heap partitioned to blocks (typically, 512 bytes).
- Start by computing for each block the total size of objects preceding that block (**the offset vector**).
- Requires a single pass over the mark-bit vector.
  - Assume: a **markbit vector** with a bit set for beginning and ending word of each object.



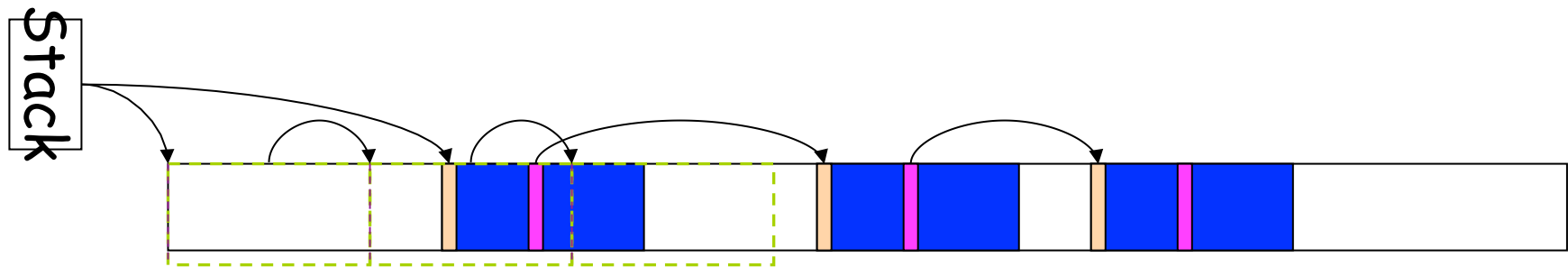
# Using the Offset Vector

- Computing an address from the offset vector:
- The new address of an object  $O$  in block  $B$  is the offset vector entry of  $B$  plus the size of the objects in  $B$  preceding  $O$ .
- For object 5:  $Fix-up(1375) = 1000 + 125 + 50 = 1175$ .
- (Give up the additional IBM structures.)



# The Basic Compressor (Single-Threaded Compaction)

- Stop application threads.
- Calculate Offset vector.
- Fix roots.
- For every object in address order:
  - Move object
  - Fix its pointer.
- Resume application threads.



# Properties

- One table pass, one heap pass = efficient.
- Small space overhead (the offset vector).
- But - Single threaded.
- The main supporting invariant:
  - “Old object version is not stepped on before it moves”.
  - Due to ordered move of objects.
- This invariant cannot be guaranteed when parallel threads move objects in parallel.

# Parallelization

- If we had two spaces ...
- The Compressor could compact the objects from one space (**from-space**) into the other (**to-space**).
- Advantage: each object can be moved independently of the others → Simple parallelization
- Problem: space overhead.

# Virtual Memory Reminder

- A process has a large **virtual** (memory) address space, maybe larger than physical memory.
- This space is divided into virtual pages (typically 4KB)
- Some of the virtual pages are kept on physical memory while the rest reside on the hard disk.
- The system keeps a map from virtual to physical pages.
- Due to locality of computation, **page faults** (pages that are not available in the memory and must be retrieved from the disk) are seldom.

# Virtual Memory Primitives

- A virtual page is *mapped* into a physical page when a space on that page is allocated.
- Virtual pages consume “real” resources (such as space on disk and memory) when mapped.
- A virtual page can be *released* or *unmapped* when the process does not need the data on it anymore.
- A physical page can be mapped from two different virtual pages.
  - Changing one virtual address will affect the other...

# Virtual Memory Primitives

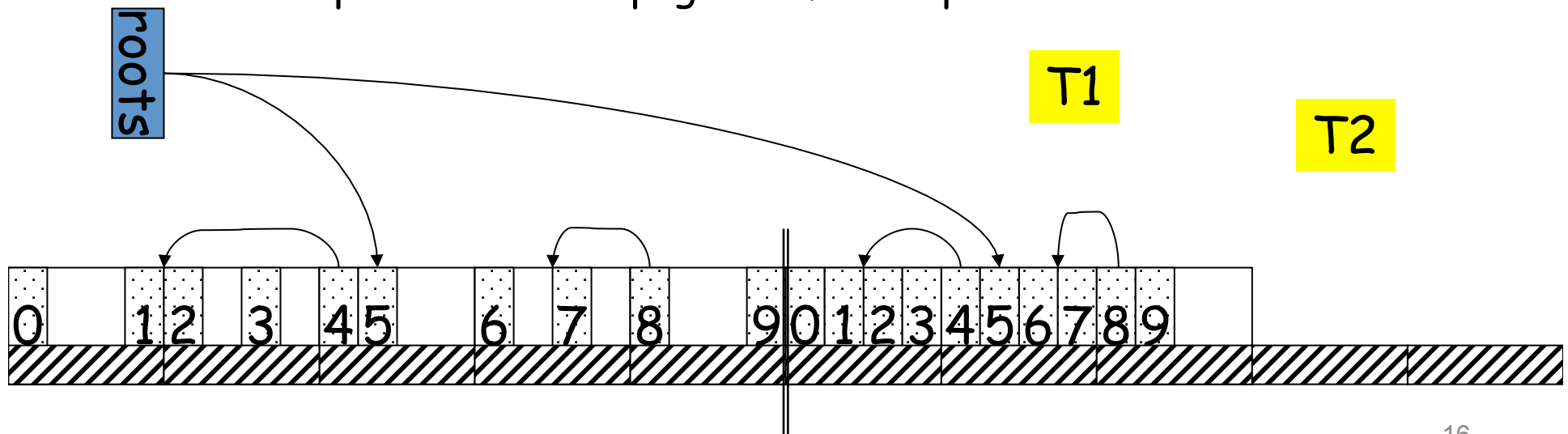
- A virtual page can be *read-protected* or *write-protected*.
- When the process tries to read (or write) from a read- (or write-) protected page, a **trap** springs.
- A **trap handler** code is invoked at that time.

# Parallel Compressor without Double-Space Overhead

- Initially, **to-space** is not mapped to physical pages.
  - It is a virtual address space.
- Execute in parallel: for every (virtual) page in **to-space**:
  - Map the virtual page to physical memory.
  - Move the corresponding **from-space** objects & fix pointers.
  - Unmap the relevant pages in from-space.

# Parallel Compressor without Double-Space Overhead

- Initially, **to-space** is not mapped to physical pages.
  - It is a virtual address space.
- Execute in parallel: for every (virtual) page in **to-space**:
  - Map the virtual page to physical memory.
  - Move the corresponding **from-space** objects & fix pointers.
  - Unmap the relevant pages in from-space.



# Parallel Compressor Algorithm

- Stop application threads.
- Calculate Offset vector.
- Fix roots.
- Run in parallel: while there are pages in to-space to handle:
  - Map the page.
  - Move the objects to the page and fix their pointers.
  - Unmap the unnecessary pages in from-Space.
- Resume application threads.

# Problem with Concurrency

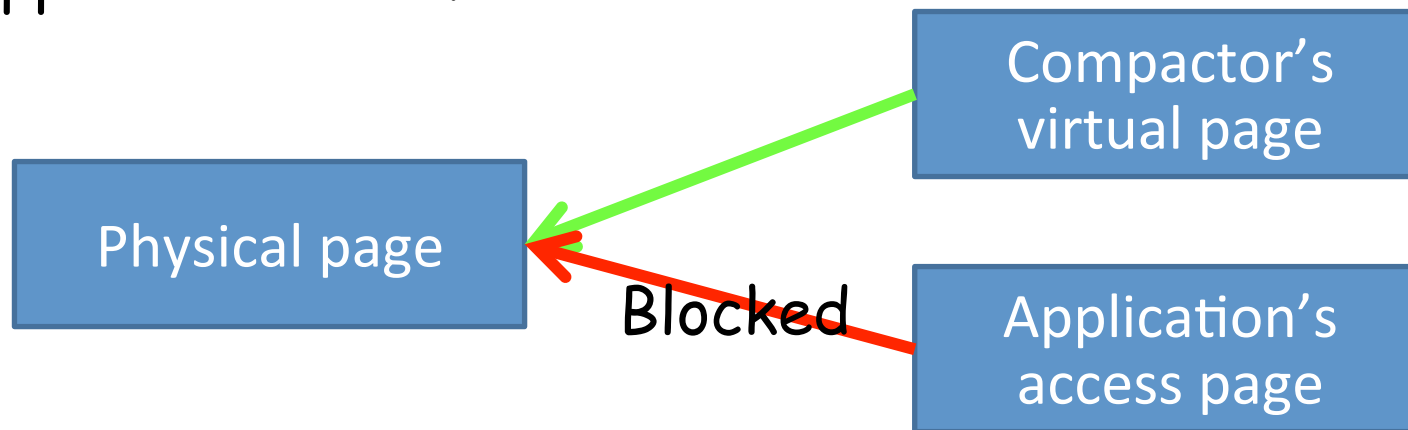
- If we move objects while application is running, two copies appear.
  - It is difficult to synchronize all threads to properly switch between using one copy or the other.

# Concurrent Compressor

- Solution: at a well-defined point in time, all application threads stop using old versions and start using only moved objects (in to-space).
  - Initialization: all threads are stopped, and
    - Roots are modified to point to to-space,
    - All to-space pages are read-protected, and
    - Application resumes.
  - Trying to touch a to-space page springs a trap.
  - The trap handler moves the relevant objects into the to-space page and lifts protection.

# Fixing To-Space

- How can the trap handler fix a protected to-space while it is still protected?
- Lifting the protection is not an option, because other threads may touch the non-ready page.
- Solution: double mapping. The handler reaches the page via a different virtual space, unprotected and mapped to the same location.



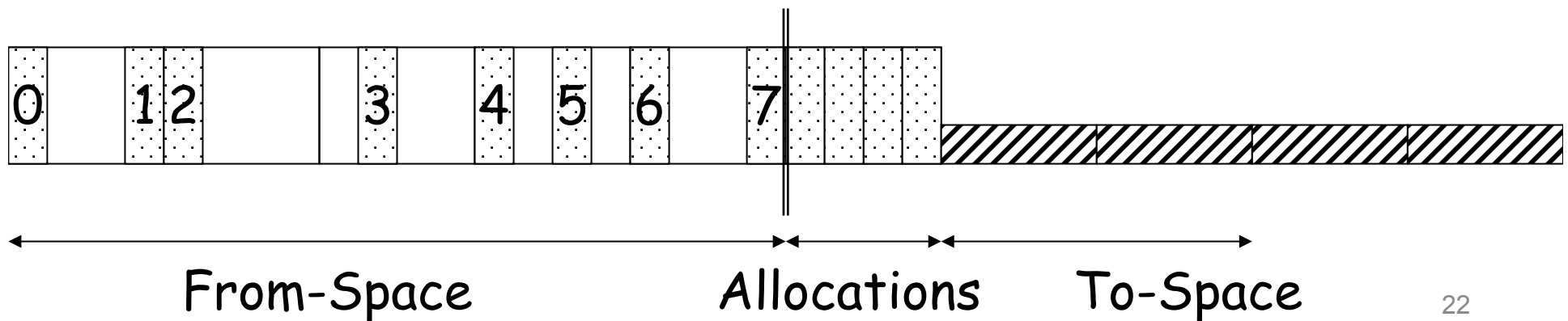
# A Simple Concurrent Solution

- Stop application threads.
- Calculate offset vector.
- Protect to-space.
- Fix roots.
- Resume application threads.
- Move objects via a trap handler.



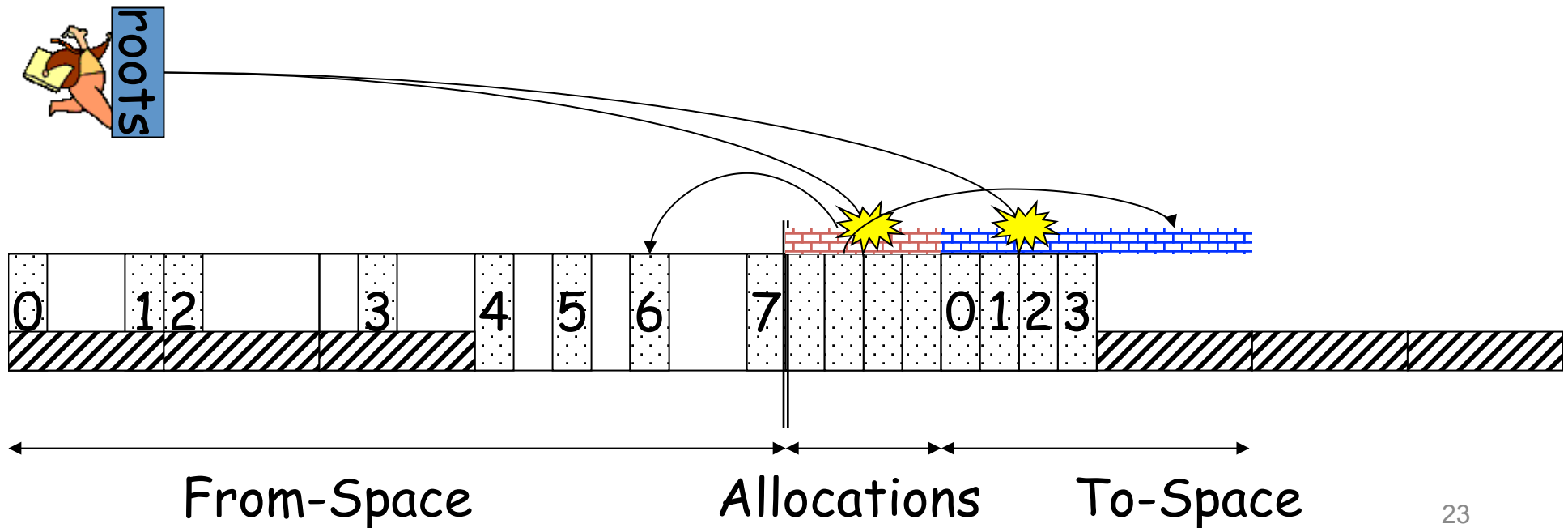
# Concurrent Solution - Offset Vector

- The offset vector can be calculated without stopping the application threads:
  - The calculation is done concurrently by the application threads.
  - Done only for old objects.
  - *New objects* are allocated in to-space.



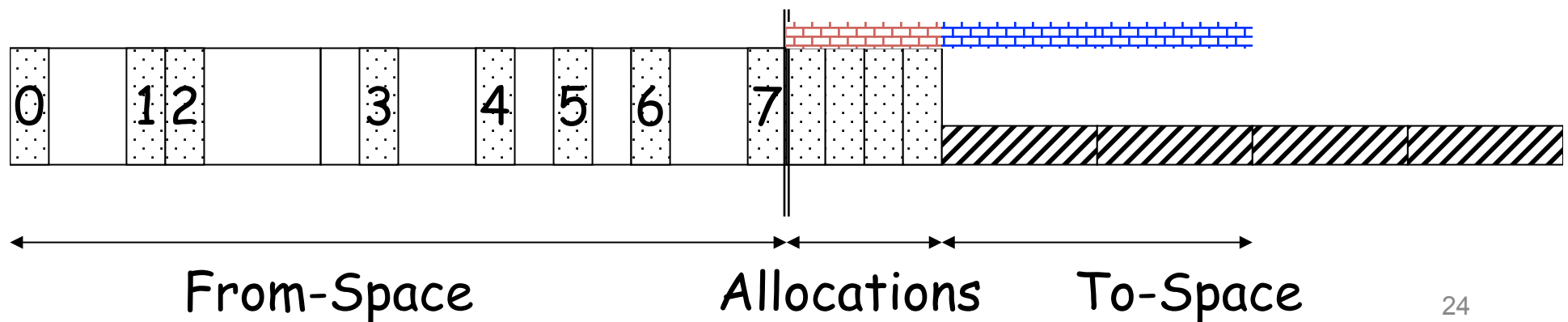
# Concurrent Solution - Offset Vector

- New objects may contain pointers to from-space.
- New object pages are later protected.
- Pointers in new objects are fixed via traps.



# Concurrent Solution - Protection

- Turning on protection of to-space pages can be done while the application runs, because these pages are not in use.
- Protection of pages with new objects must be done while the application is stopped.



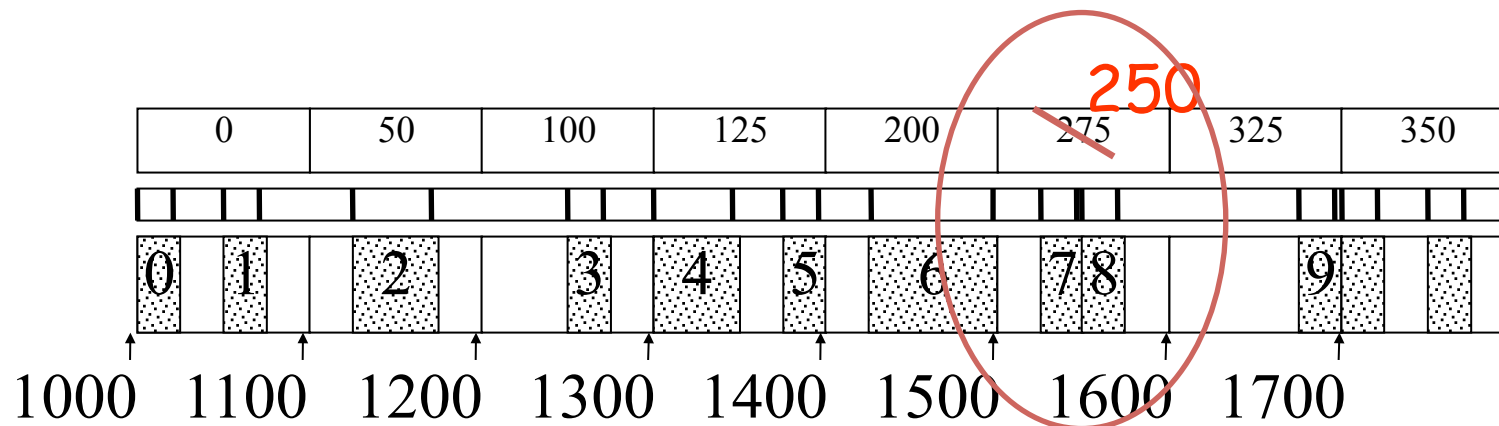
# Better Concurrent Solution scheme

- Stop application threads; “tell them” to start allocating in new object space; resume threads.
- Calculate offset vector.
- Turn on protection on to-space.
- Stop application threads.
- Fix roots.
- Protect the new objects space.
- Resume application threads.
- Move the objects via a trap handler.
- Fix the new objects via a trap handler.

# Fix-up function - an improvement.

In case the block is *dense*:

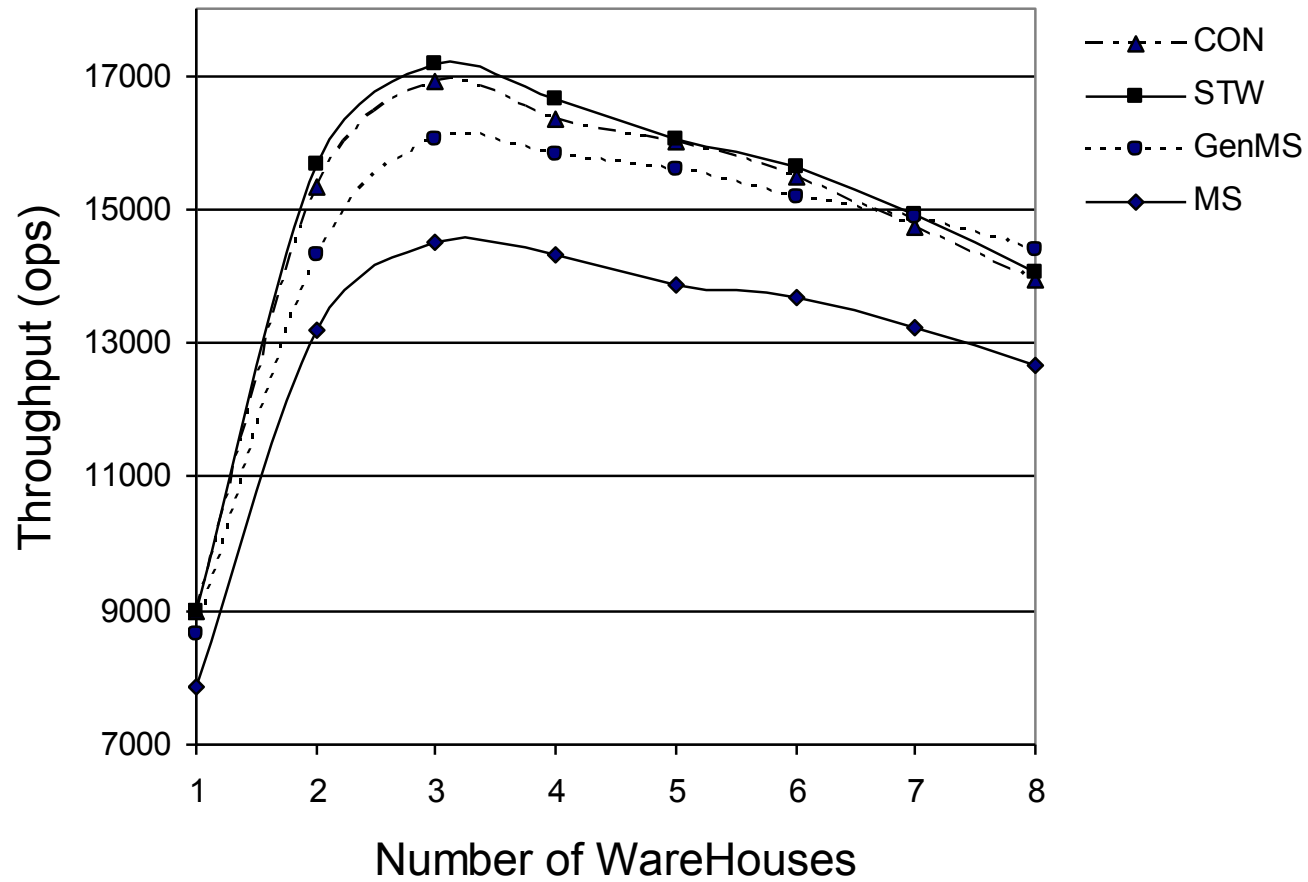
- Object 7:  $Fix-up(1525) = 1000 + 275 + 0 = 1275 (=1525 - 250)$ .
- Object 8:  $Fix-up(1550) = 1000 + 275 + 25 = 1300 (=1550 - 250)$ .
- In this block, new address = old address - 250, because it is dense. It turns out many blocks get dense with time.
- Instead of keeping the 275, keep 250.
- The LSB distinguish between the cases.



# Implementation & Measurements

- The Compressor was implemented on the Jikes RVM - a research Java Virtual Machine.
- The main benchmark is the Specjbb2000, A server application running one to eight threads.
- The Compressor performance was compared to the performance of Mark-Sweep and GenMS.
- Unfortunately, there were no concurrent nor compaction algorithms on the Jikes RVM, that could be compared.

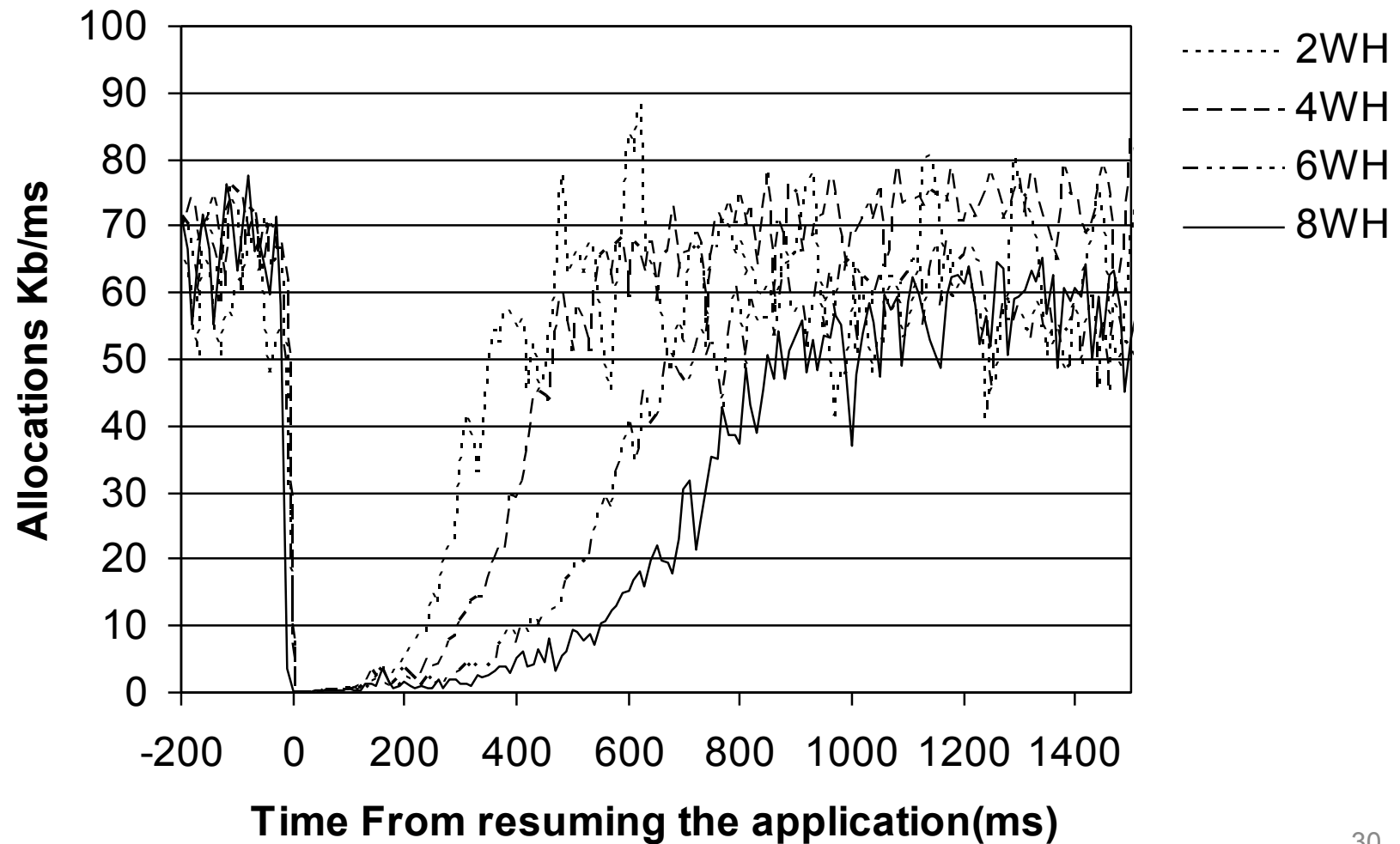
# Measurements - efficiency



# Measurements - pause time

	Parallel Compaction (Stop-The- World)	Mark and Sweep	generational Mark and Sweep (full collections)
jbb 2-WH	319.55	229.37	279.73
jbb 4-WH	516.89	287.32	323.64
jbb 6-WH	641.53	315.71	347.42
jbb 8-WH	770.14	372.46	374.41

# Measurements - Allocations per time



# Conclusion

## The Compressor:

- The first compactor with one heap pass (in addition to a table pass).
- Fully compact all the objects in the heap.
- Preserves the order of the objects.
- Low space overhead.
- Uses memory services to obtain parallelism.
- Uses traps to obtain concurrency.

# Conclusion --- Compaction

- Uniprocessor compaction:
  - Two fingers, Lisp2, Threaded (Yonkers)
- Parallel compaction:
  - Sun's compaction, IBM's compaction.
  - Parallel and Concurrent: the Compressor.
- Issues considered:
  - Efficiency, space overhead, parallelism, compaction quality, locality.



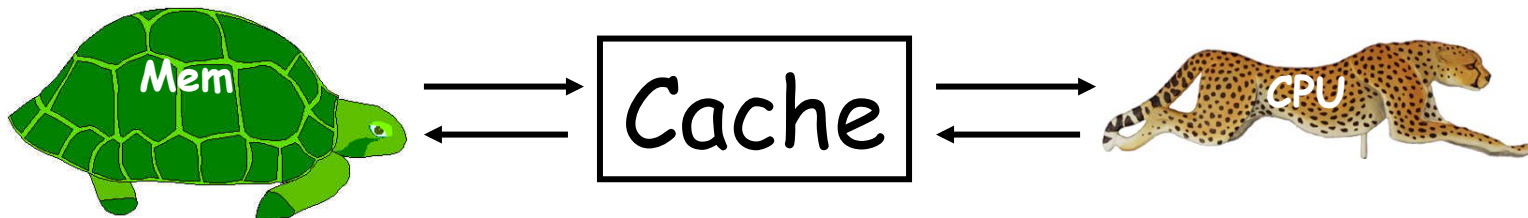
# Topics this week

---

- 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].

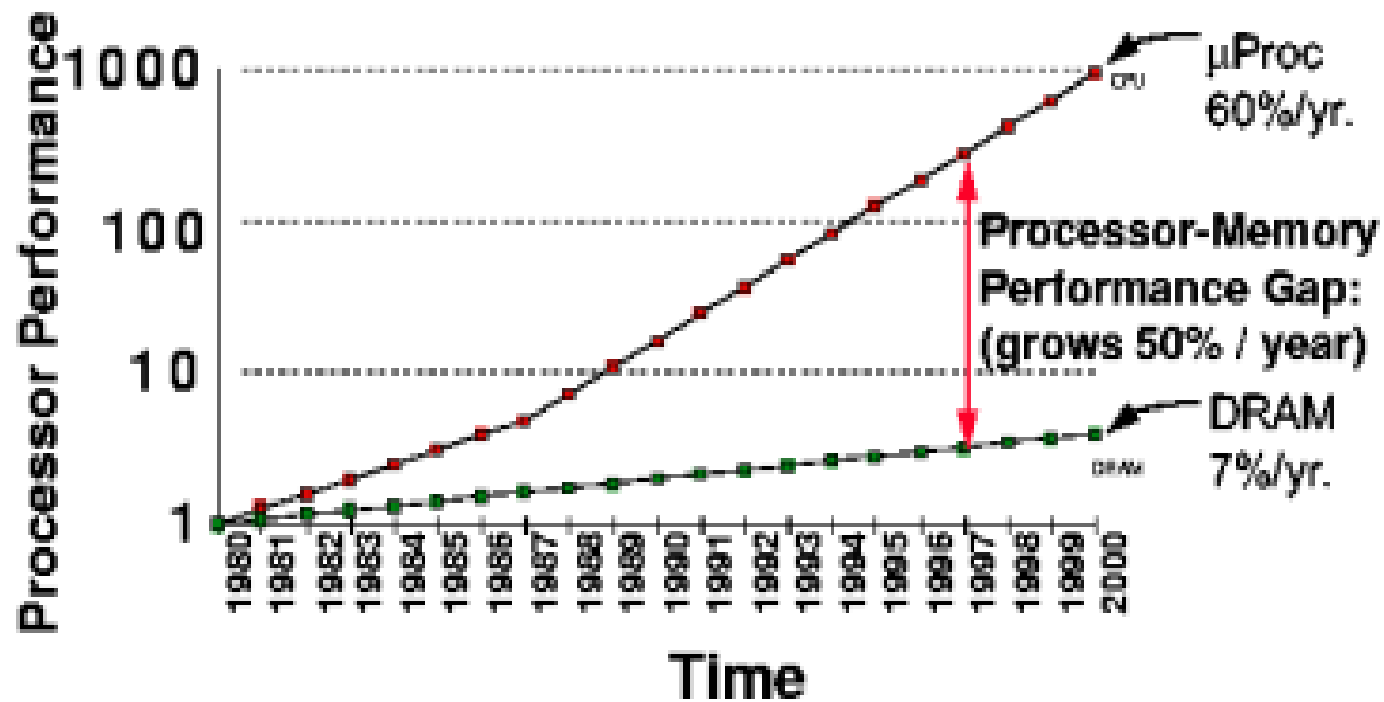
# Computers today

- Memory speed falls behind processor speed, and gap still increasing.
- Solution: use a fast cache between memory and CPU.
- Implication: program cache behavior has a significant impact on program efficiency.



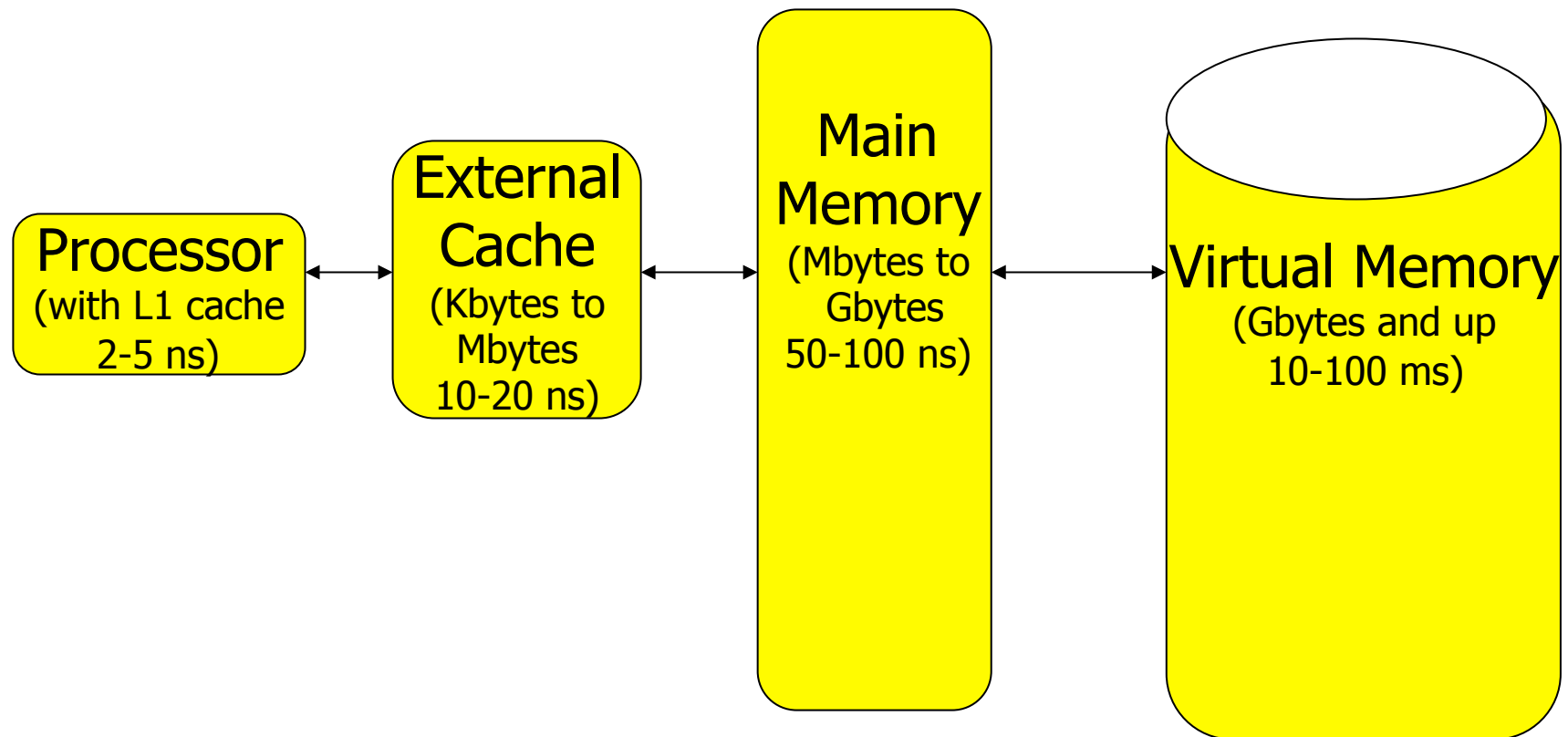


## Processor-DRAM Gap (latency)



# Memory Hierarchy

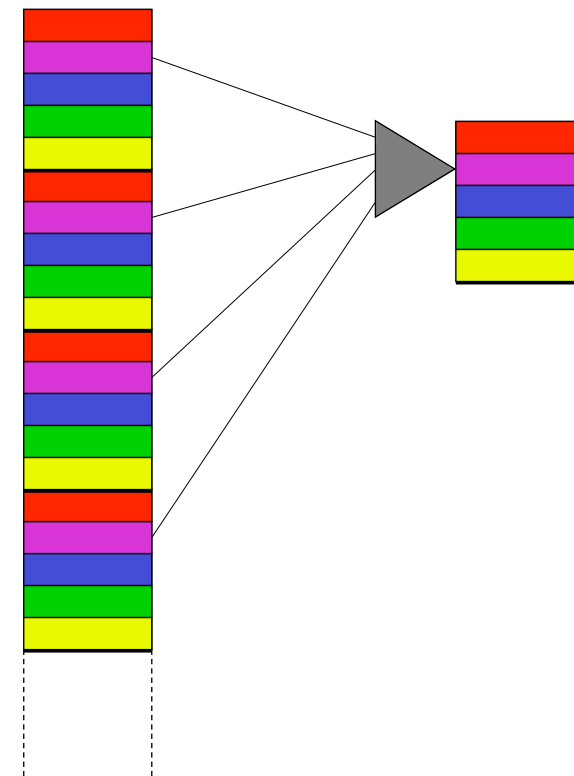
Numbers change  
from day to day...



# Cache structure

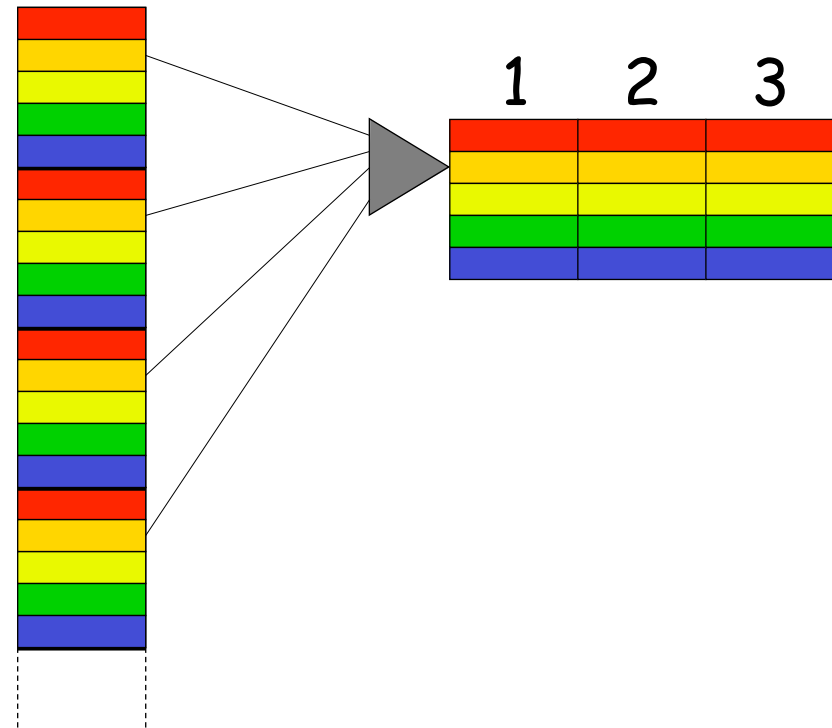
- Large memory divided into blocks.
- Small cache -  $k$  blocks.
- Mapping of memory blocks to cache blocks.
- Cache hit - accessed block is in the cache.
- Cache miss - required block must be read to cache from memory.

## Direct mapping



# Associative cache:

- t-way Associative caches:
- $t \cdot k$  blocks in cache,  $k$  sets,  $t$  blocks in a set.
- memory block 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.



# Block/Line Size

---

- Ranges between 16-128 bytes
- Large blocks:
  - 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 blocks from memory.
- **System: match parameters to cache.**



# 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].



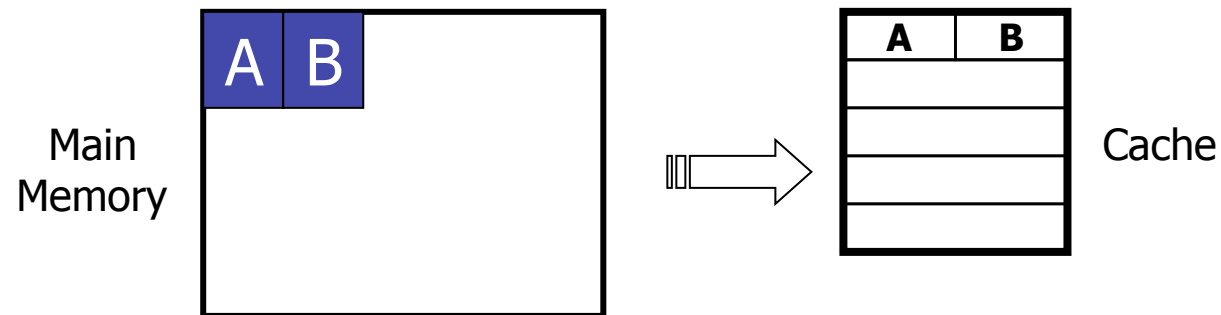
# Placing Data Appropriately

---

- [Chilimbi and Larus 98]
  - If objects have high temporal affinity: place them near each other
  - Idea: they will share a cache line!
  - Two misses become one.
- [Calder et al 98]
  - If objects have high temporal affinity: do not let them collide!
  - Idea: reduce collision misses.

# Chilimbi and Larus

- Goal: place “related” objects one after the other in memory so they map into the same cache line
- Produce a cache conscious object layout





# Two problems

---

- Problem 1: how do we know which objects will be “related”?
- Problem 2: how do we move objects?



# Getting the information

---

- Option 1: compiler analysis.
- Option 2: training. First run program to check what's happening, then use information for the second run.
  - requires extra runs. (preferably on typical programs.)
  - Hard to transfer consistent and usable names of objects from one run to another.
  - Behavior may be dynamic.



# Getting the information

---

- Option 3 (the one chosen): runtime monitoring: collect information while program is running.
  - Assume behavior is stable, so information gathered from previous epoch is useful to predict behavior in next epoch.
  - Must be low overhead.
  - A read barrier is too expensive.

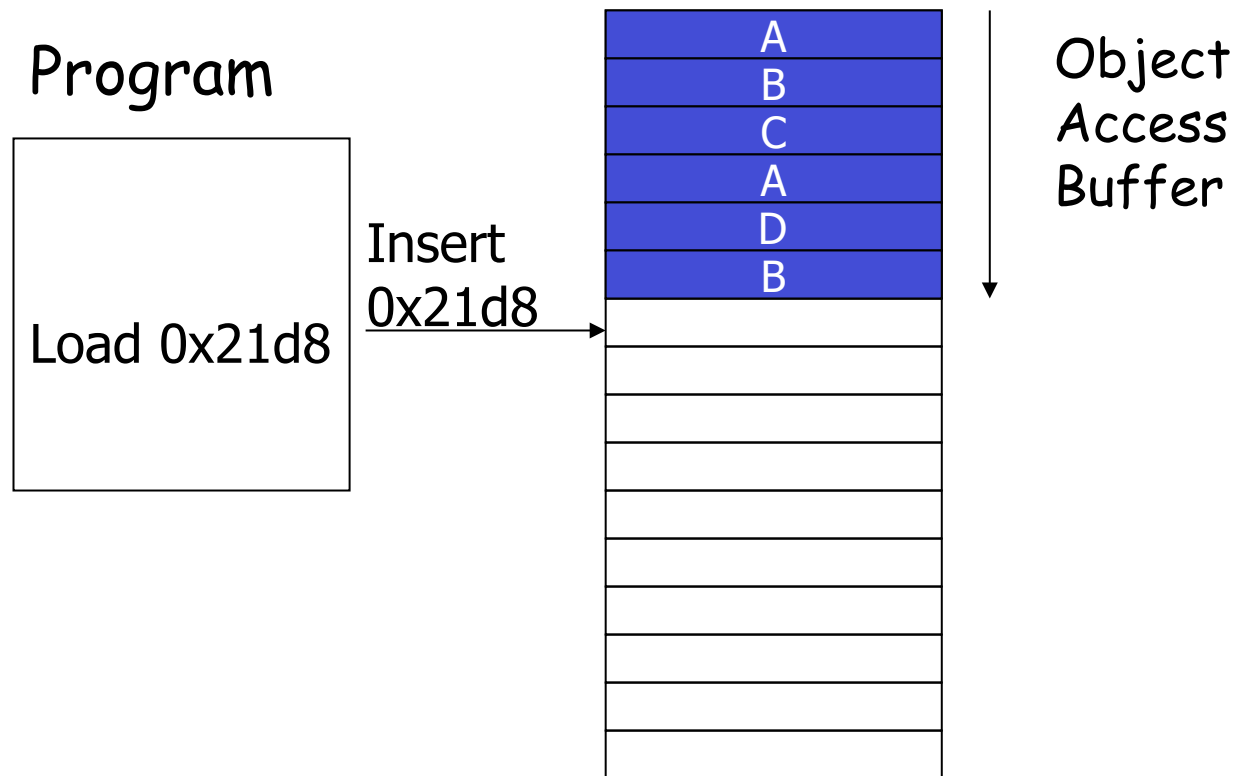


# The Object Access Buffer

---

- Solution: record only **object** access.
- Justification:
  - Objects are small, < 32 bytes mostly (in Cecil) thus, profiling at this resolution is enough.
  - Objects accesses are not lightweight, so recording object accesses is not too costly.
- Thus, monitoring loads of base object address is low overhead.

# Object Access Buffer



Records order of program's object accesses



# The Object Affinity Graph

---

- **Access buffer**: an ordered list of objects touched.
- We need to represent data from access buffer “compactly”.
- **Object affinity graph**: weighted undirected.
  - Each vertex represents an object.
  - An edge  $(A,B)$  of weight  $n$  implies that objects  $A$  and  $B$  were accessed closely in time  $n$  times.

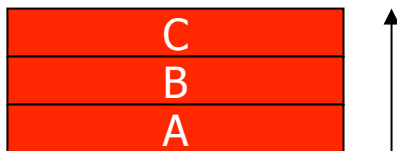
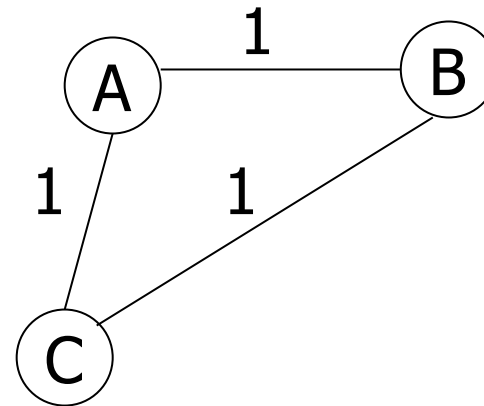
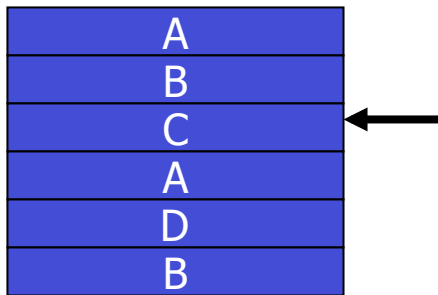


# What is “close”?

---

- Interpreting “close” with parameter  $w$ :
  - An access of object  $A$  is close to its  $w$  preceding accesses and  $w$  subsequent accesses.
  - Value of  $w$  used in this work:  $w=2$ .
  - Tradeoff between representing more information (when  $w$  large) and processing time.

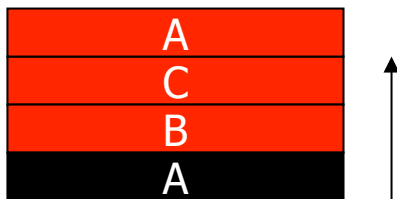
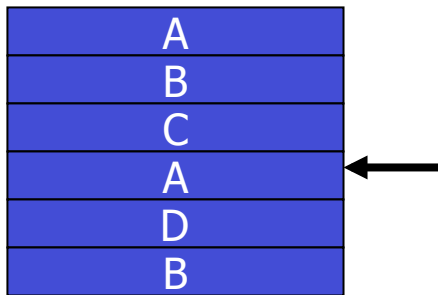
# Building the Object Affinity Graph



Queue (of size 3)

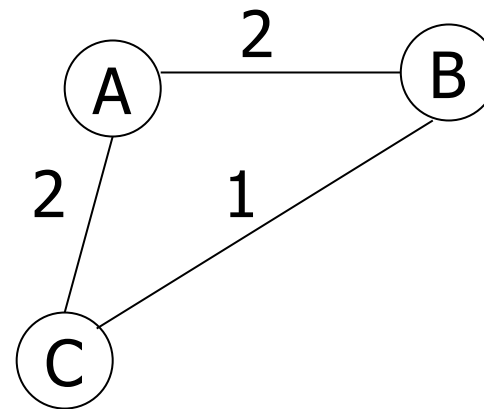
January 5, 2012

# Building the Object Affinity Graph

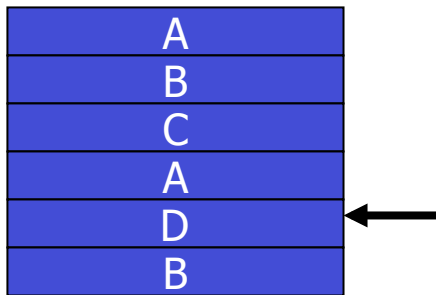


Queue (of size 3)

January 5, 2012

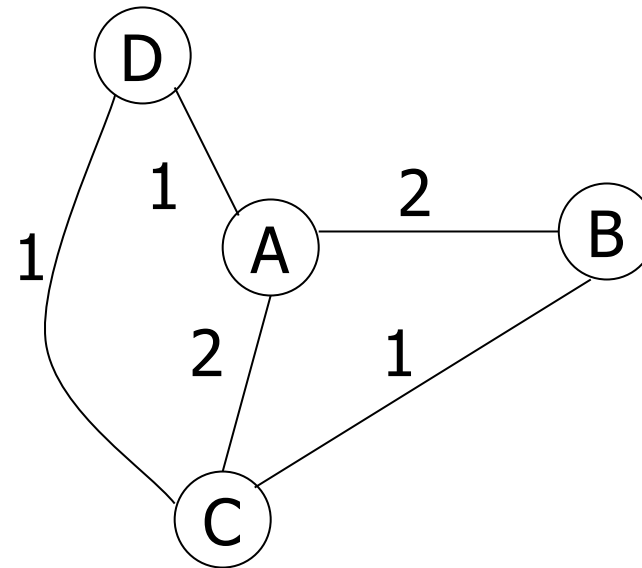


# Building the Object Affinity Graph

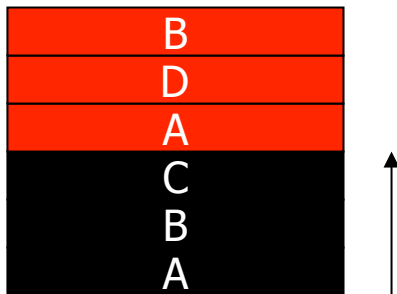
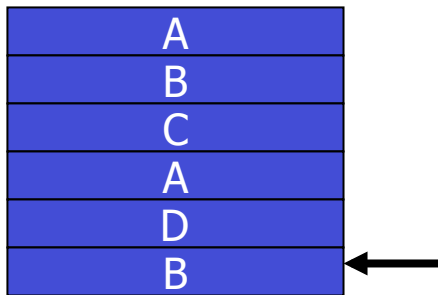


Queue (of size 3)

January 5, 2012

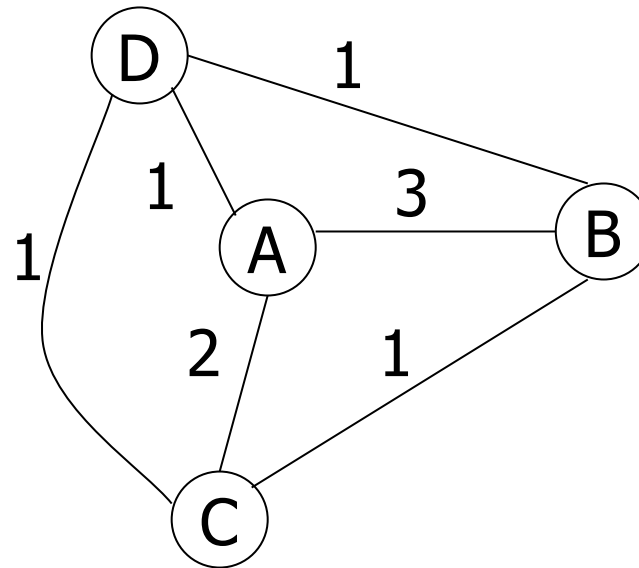


# Building the Object Affinity Graph



Queue (of size 3)

January 5, 2012





# Course of Collection

---

- During program run, access buffer is created.
- When GC starts, affinity graph is built (from scratch) according to the access buffer.
- The algorithm only works on the old generation.
  - Most objects in young generation are going to die...
  - Affinity graph is only built for the old generation.



# Using Affinity Graph to Collect

---

1. Start copying objects in the affinity graph via “Greedy DFS”:
  1. Start with the heaviest object.
  2. When choosing an edge go from heavy to light.
2. Go over to-space and update pointers (while perhaps copying objects not in the affinity graph)
3. Run Cheney on the rest of the objects.



# Remarks

---

1. Dead objects may be copied
  1. But will be collected during the next cycle.
2. Live objects will always be copied.



# Results: overheads

---

<b>Program</b>	<b>Original execution time (secs)</b>	<b>Instrumented program execution time (secs)</b>	<b>% overhead of instrumentation</b>
richards	0.202	0.213	<b>5.45</b>
deltablue	3.369	3.544	<b>5.19</b>
instr sched	3.518	3.683	<b>4.69</b>
typechecker	347.352	358.467	<b>3.20</b>
new-tc	391.250	403.378	<b>3.10</b>

**Table 4: Overhead of real-time data profiling.**



# Results: improvements

Program	L2 cache miss rate (base)	L2 cache miss rate (CCDP)	% reduction (L2 miss rate)	Execution time (base)	Execution time (CCDP)	% reduction (execution time)
richards	0.0131	0.0103	21.4	0.202	0.173	14.4
deltablue	0.0356	0.0240	32.6	3.369	2.578	23.5
instr sched	0.0543	0.0392	27.8	3.518	2.756	21.7
typechecker	0.0947	0.0591	37.6	347.352	238.179	31.4
new-tc	0.0979	0.0571	41.7	391.250	247.622	36.7

Table 5: Impact of our cache-conscious object layout.



# Summary

---

- A reduction in the L2 cache miss rate by 14%-37%.
- An improvement in program performance by 18%-31%
- Excellent results.



# Summary

---

- A reduction in the L2 cache miss rate by 14%-37%.
- An improvement in program performance by 18%-31%
- Excellent results.
- It is not clear that they are obtainable on other JVM's and other platforms
  - Results have not been reproduced since.



# 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

---

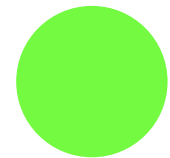
- Overhead of reordering analysis is ~2%.
- Most bm's vary by ~4% due to copy order.
- Four programs are more sensitive (up to 25%)
- Methods compared: BFS, DFS, and partial DFS, using the first two children.
- Online object reordering matches or improves upon the best class-oblivious ordering.



# 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;
  - Zorn's recommendations.
- Limitations on the ability to improve data placement [Petrank-Rawitz 02].



# Zorn's Remarks

---

- With generations, young generation had better fit in cache.
- Use of 2- and 4-way set-associative caches gave little performance gain over direct-mapped ones



# Boehm: Reduce Cache Misses for Tracing GC

---

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



# Prefetch(x)

---

- Hint to hardware: start bringing 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.
- Many machines now have some form of prefetch instruction.



# Dealing with the Mark Phase

Recall mark phase:

Ensure that all objects are unmarked.

Mark and Push to markstack 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)
```

Access markstack

Access heap

Access bitmap



# Prefetching gray objects



**An observation** (by measurements):  $\sim 1/3$  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.
- Reason: 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].



# Agenda

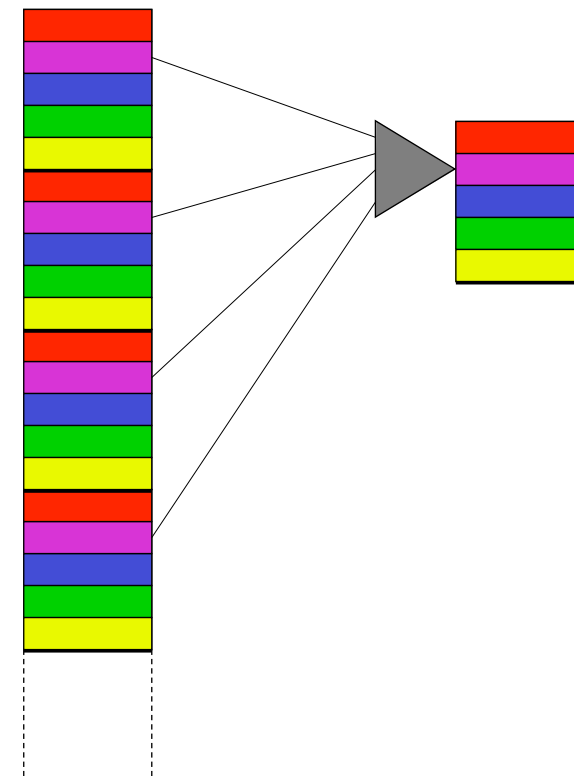
---

- 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

# Cache Structure

- Large memory divided into blocks.
- Small cache -  $k$  blocks.
- Mapping of memory blocks to cache blocks.  
(e.g., modulus function)
- Cache hit - accessed block is in the cache.
- Cache miss - required block must be read to cache from memory.

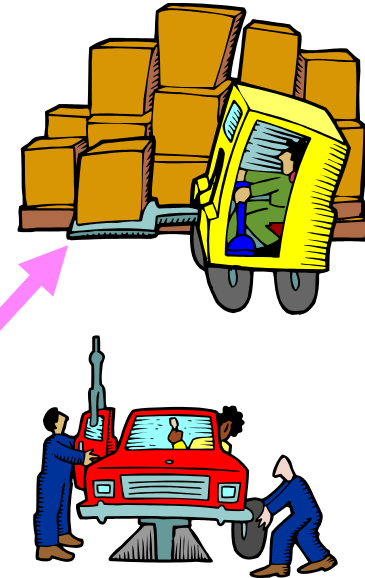
## Direct mapping



# 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  $O = \{o_1, \dots, o_m\}$ , and a sequence of accesses  $\sigma = (\sigma_1, \dots, \sigma_n)$ .  
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  $o_1, \dots, o_m$  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  $\approx$  **twice** the optimum

**No! Unless,  $P=NP$ .**

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

Fairly = # misses  $\approx$  **100 times** the optimum

**No! Unless,  $P=NP$ .**



# The Results

---

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

No! Unless,  $P=NP$ .

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

No! Unless,  $P=NP$ .



# The Main Theorem

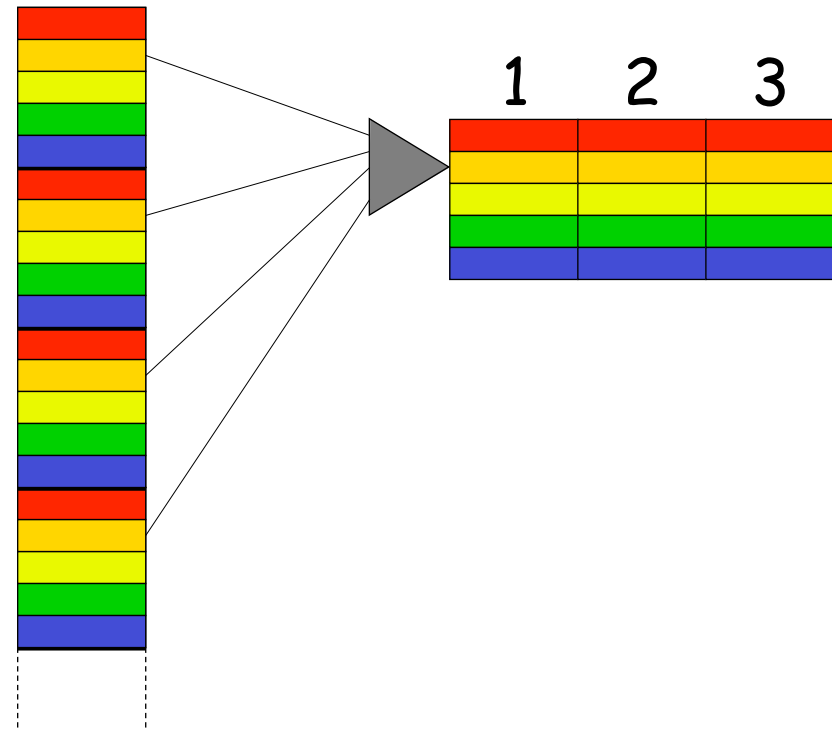
---

Let  $\varepsilon$  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  $> 2$  blocks)

# Extend to t-way Associative Caches

- t-way Associative Caches:
- $t \cdot k$  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...



# Keeping Pairwise Information

---

- A practical problem: sequence  $\sigma$  of accesses is long. Processing it is costly.
- Solution in previous work: keep relations between pairs of objects.
  - E.g., for each pair how beneficial is putting them in the same memory block.
  - E.g., for each pair how many misses would be caused by mapping the two to the same cache block



# Pairwise Information is Lossy

---

## Theorem 3:

There exists a sequence  $\sigma$  such that

$$\#misses(f) \geq (k-3) \cdot \#misses(f^*)$$

$f$  - optimal pairwise placement

$f^*$  - optimal placement

## Conclusion:

Even when given **unrestricted time**, finding an optimal pairwise placement is a bad idea (worst case).



# Pairwise Information: Hardness Result

---

Theorem 4: Let  $\varepsilon$  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 with respect to pairwise information, then  $P=NP$ .

Proof is similar to the direct mapping case.



# A Simple Observation

---

- Input: Objects  $O=\{o_1,\dots,o_m\}$ , and accesses  $\sigma=(\sigma_1,\dots,\sigma_n)$ .
- 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  $n^{(1-\epsilon)}$  is not possible.)



# What about positive results?

---

In light of the lower bound not much can be done in general. Yet...

Theorem 5: There exists a polynomial time approximation algorithm that outputs a placement (always) within a factor of  $\frac{n}{c \cdot \log n}$  from the optimal placement for any  $c$ .

Compare:

Impossible:  $n^{(1-\epsilon)}$ , Possible:  $\frac{n}{c \log n}$



# Comparison to Other Problems

---

- Data Placement:
  - Inapproximable
  - $n/c \log n$ -approximation algorithm
- Famous problems with similar results:
  - Minimum graph coloring
  - Maximum clique



# 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.*



# Some Proof Ideas (simplest - direct mapping)

---

**Theorem 1:** Let  $\varepsilon$  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  $P=NP$ .

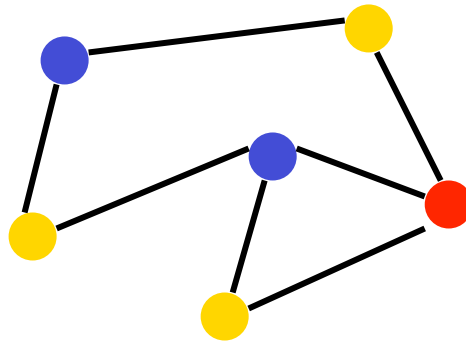
**Proof:** We show that if the above algorithm exists, then we can decide for any given graph  $G$ , if  $G$  is  $k$ -colorable.



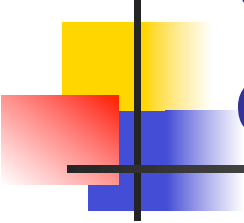
# The $k$ -colorability Problem

---

- Problem: Given  $G=(V,E)$ , is  $G$   $k$ -colorable?



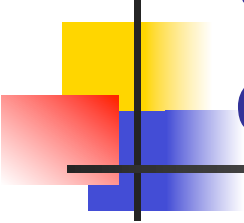
- Known to be NP-complete for  $k > 2$ .



# Reducing a Graph $G$ into a Cache Question

Graph	Cache Question
Vertex $v_i$	Object $o_i$
Edge $e=(v_i, v_j)$	Subsequence $\sigma_e=(o_i, o_j)^M$
Color	Cache line

Coloring  $\Leftrightarrow$  Placement



# Reducing a Graph $G$ into a Cache Question

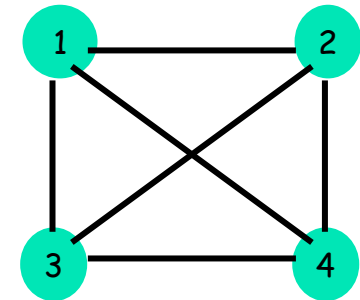
---

- A vertex  $v_i$  is represented by an object  $o_i$ :  
 $O_G = \{ o_i : v_i \in V \}$
- Let  $\ell = (3/\varepsilon) - 1$ . The edge  $(v_i, v_j)$  is represented by many repetitions of the two objects  $o_i, o_j$ :

$$\sigma_G = \prod_{(v_i, v_j) \in E} (o_i, o_j)^{\ell}$$

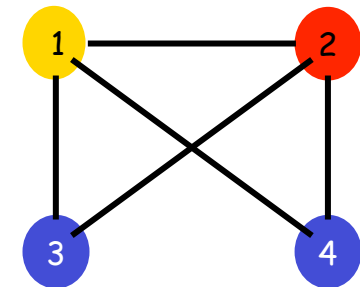
# Examples

- $G_1$  (not 3-colorable)



$$\sigma_1 = (o_1, o_2)^{6^l} (o_1, o_3)^{6^l} (o_1, o_4)^{6^l} (o_2, o_3)^{6^l} (o_2, o_4)^{6^l} (o_3, o_4)^{6^l}$$

- $G_2$  (3-colorable)



$$\sigma_2 = (o_1, o_2)^{5^l} (o_1, o_3)^{5^l} (o_1, o_4)^{5^l} (o_2, o_3)^{5^l} (o_2, o_4)^{5^l}$$



# Properties of the Reduction

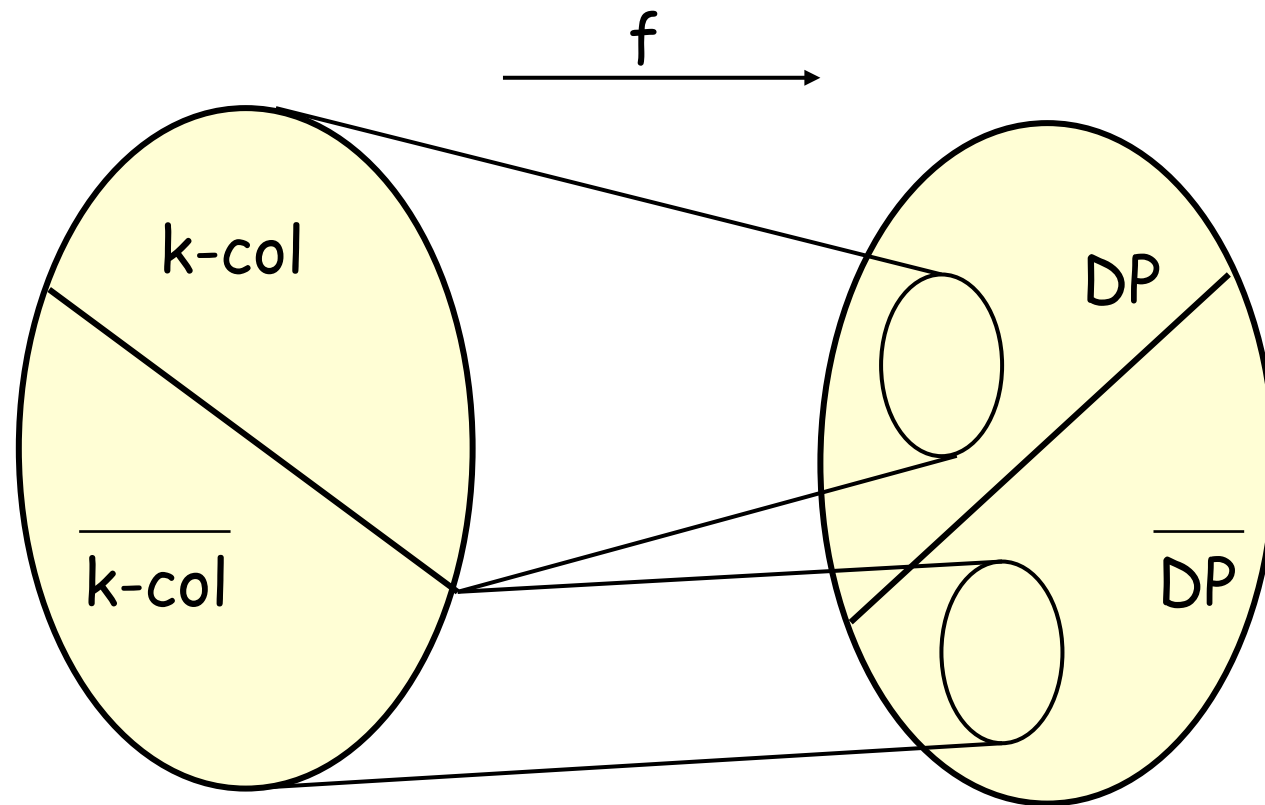
---

- Length of  $\sigma$ :  $n = O(|E|^{\ell+1})$
- Case I:  $G$  is  $k$ -colorable.  
Then,  $\text{Opt}(O_G, \sigma_G) = O(|E|) = O(n^{1/(\ell+1)}) = O(n^{\varepsilon/2})$
- Case II:  $G$  is not  $k$ -colorable.  
Then,  $\text{Opt}(O_G, \sigma_G) = \Omega(|E|^\ell) = \Omega(n^{\ell/(\ell+1)}) = \Omega(n^{1-\varepsilon/2})$

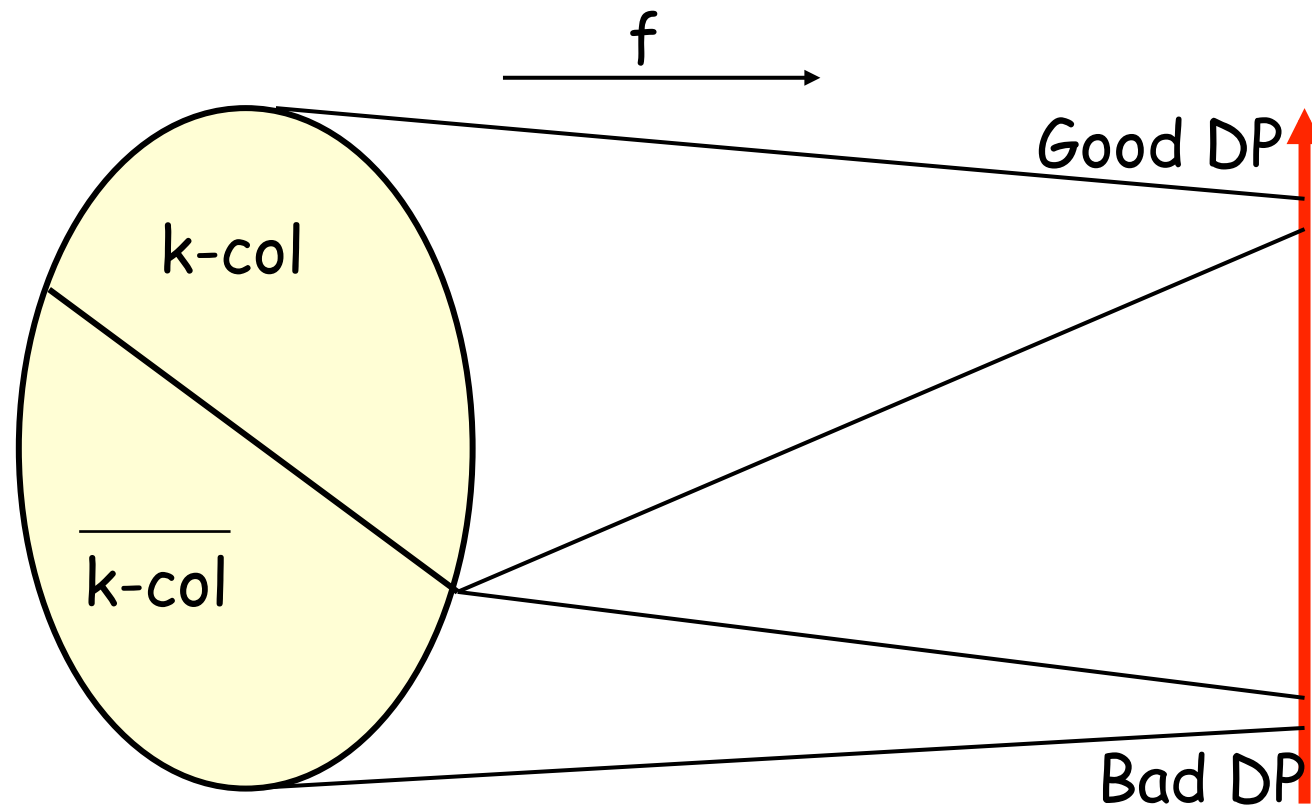
A polynomial reduction from 3-Colorability to data placement with an extra strength !



# A “Normal” Reduction



# A "Super" Reduction





# Hard to Find Good Placement

---

- Length of  $\sigma$ :  $n = O(|E|^{\ell+1})$
- Case I:  $G$  is  $k$ -colorable.  
Then,  $\text{Opt}(O_G, \sigma_G) = O(|E|) = O(n^{1/(\ell+1)}) = O(n^{\varepsilon/2})$
- Case II:  $G$  is not  $k$ -colorable.  
Then,  $\text{Opt}(O_G, \sigma_G) = \Omega(|E|^\ell) = \Omega(n^{\ell/(\ell+1)}) = \Omega(n^{1-\varepsilon/2})$

An algorithm that finds a placement within  $n^{(1-\varepsilon)}$  from the optimum can distinguish the two cases!



# Conclusion

---

- Computing the best placement of data in memory (w.r.t. reducing cache misses) is extremely difficult.
- We cannot even get close (if  $P \neq NP$ ).
- There exists a matching (weak) positive result.
- Implications:
  - using heuristics cannot be avoided.
  - We cannot hope to evaluate potential benefit.