Algorithms for Dynamic Memory Management (236780)

Lecture 10

Lecturer: Erez Petrank
Last Week

- Compressor
- Reference Counting
Topics Today

- Parallel GC
- Cache consciousness
- Real-time
Recall Terminology

Stop-the-World

Parallel

Concurrent

On-the-Fly

program
GC
Part I: Parallel Mark & Sweep
Motivation

- Concurrent GC is not scalable enough
  - Parallelism needed.
- In 1997 this was the first work to study mark-sweep scalability.
- A 64-way SMP is used.
- Speedup reported.
What do we want?

- **Goals:**
  - Scalability, load balancing, locality of reference, simplicity

- **Main problem:**
  - Static work partitioning not good enough, needs dynamic load balancing.
  - **Idea:** over-partition the work.
Why Bitmap Sync is Required

Two threads mark two objects concurrently. Both objects are mapped to the same bitmap word.

1. Read bitmap word
2. Set the bit
3. Write bitmap word

T1

T2
Parallel Sweeping

- Each process grabs a part of the heap and handles it
  - Use of Bohm’s collector that employs blocks
  - Take 64 blocks at a time and process all blocks’ free space.
  - After locally processing the blocks, a lock is taken and the space is added to the global free list.
Naïve Implementation

- This naïve implementation results in hardly any speedup.
- Thus, improvements required.
Introducing Load Balancing

- Problem: Consider the following shared tree:

  The process that first marks the tree's root will have to mark the rest of the tree!
Stealable Mark Queues

- Each processor keeps a “stealable mark queue”.
- Once in a while, if stealable mark queue empty, move $\frac{1}{2}$ of the jobs from markstack to stealable mark queue.
- If processor idle - search for jobs in stealable mark queues. (Steal $\frac{1}{2}$ of a s.m.q.)
  - Idle processors help the busy ones
  - Sync on stealable mark queues
  - Tougher termination detection
Termination

- Keep a global counter with the number of empty stacks and empty queues.
- Counter is updated whenever a processor fills its mark queue, becomes idle, or obtains a task.
- When counter reaches twice the number of processors - GC ends.
Empirical Evaluation

- With stealable mark queues, the algorithm exhibits at most \(12x\) speed-up on a 64-way SMP.
- Next, 4 improvements.
1: Split Large Objects

- A process that marks a **large object** (e.g. 400KB) is tied up for a long time (**load imbalance**)
- **Solution**: Break objects into 512-byte **chunks** before inserting into mark stack
2: Skip Locked Queues

- Sometimes many processes attempt to steal from the same queue and must wait to enter the critical section \(\text{(contention)}\)
- **Solution:** If lock can’t be acquired on first try, give up and go to the next queue.
3: Markbits Test

- Sync. (lock) is used to mark objects
- However, many times the object is already marked!
- **Improvement**: Read the bit first without sync. If not set, use sync to set it.
4: Improve Termination Detect

- Global counter was used to keep track of empty mark and steal queues (*contention*)

- **Improvement:** “Mark Stack Empty” and “Steal Stack Empty” flags kept on each processor. Terminate if all flags are set and global “interrupted” flag is clear.
Empirical Evaluation

- Desired: run various number of processors on the same snapshot of the heap.
- Problem: snapshot depends on number of processors. Not clear how to continue with varying number of processors from the same snapshot.
- Approach: Devise a formula for workload and compare workload/time ratios.
Workload and Speed-up

- Workload of a collection is
  \[ W = a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_4 + a_5 x_5 \]
  
  \( x_1 = \) no. marked objects marked and scanned
  
  \( x_2 = \) no. visits to already marked obj’s
  
  \( x_3 = \) no. visits to objects with no pointers
  
  \( x_4 = \) no. empty heap blocks
  
  \( x_5 = \) no. non-empty blocks

- Weights were determined by experiment
  
  \( a_1 = 0.50, a_2 = 0.16, a_3 = 0.02, a_4 = 2.0, a_5 = 1.3 \)

- Speed = \( W / t \), (for \( t = \) GC elapsed time)

- Speed-up on \( N \) proc’s = \( (W_n / t_n)/(W_1 / t_1) \)
Benchmark Apps

- BH -- simulates motion of N moving bodies (here N = 5000)

- CKY -- context-free grammar parser (run on 67 sentences, 10-40 words per sentence).
Mark Speed-up Results

Figure 5: Average marking speed-up in CKY.

Figure 6: Average marking speed-up in BH.
Indiv. Improvement Effect

No-SLO = w/o splitting large objects, No-SLQ = w/o non-blocking mark queues search, No-NSB = w/o improved termination detection, No-TCS = w/o non-blocking bitmap reads

Figure 7: Effect of each optimization in CKY.

Figure 8: Effect of each optimization in BH.
Summary

- Parallelizing a mark-and-sweep collector is possible.
- With the ideas raised in the paper the authors got speed-up around 30 with 64 processors. (Very good!)
Parallel Collection for a Copying Collector

Halstead 84
Imai-Tick 91
Kolodner-Petrrank 99
Flood-Detlefs-Shavit-Zhang 01
Copy Algorithm: Reminder

- Divides heap: to-space and from-space
- On GC, stop all threads, scan reachable objects and copy them into to-space
Parallel Copying

- Similar to mark-sweep.
- One difference: synchronize allocations in to-space.
- Second difference: work load-balance.
Main (new) Problem: Allocation

- Concurrent allocation in to-space (by several collectors):
  - **Option 1**: (Halstead): Every processor will write in a separate memory area
    - Problem: Uneven allocations cause wasted space and fragmentation
  - **Option 2**: Compete (via synchronization) on a shared free pointer in to-space.
    - Problem: [FDSZ] tried that and got too much contention.
Parallel Allocation (cont’d)

- **Option 3** (Halstead): Allocate memory blocks to each processor; when filled, allocate another.
  - Problem: doesn’t solve fragmentation

- **Option 4** (Imai & Tick): Each processor has several blocks for allocations. Block i for allocation of sizes $2^{i-1}$ to $2^i$.
  - Problem: Complicated block management, and doesn’t completely solve fragmentation.
Parallel Allocation (cont’d)

- Option 5 [KP]: delayed allocation:
  - Collectors do not really copy the objects.
  - Instead, they record intents to copy.
  - When record large enough, allocate space and execute the copy.
- No fragmentation
- Reduced synchronization.
Delayed Allocation

from-space

<table>
<thead>
<tr>
<th>total</th>
<th>object addr</th>
<th>parent addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>B</td>
<td>S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

© Erez Petrank
Delayed Allocation

```
from-space
  S
    B
      C 10
        D
      E
        4
    F
      6
```

<table>
<thead>
<tr>
<th>total</th>
<th>object addr</th>
<th>parent addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>B</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>C</td>
</tr>
</tbody>
</table>

© Erez Petrank
Delayed Allocation

From-space

<table>
<thead>
<tr>
<th>object addr</th>
<th>parent addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>S</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
</tr>
</tbody>
</table>

Total: 20

- B: 10
- C: 6
- E: 4
- F: 2
Partitioning the work

- Option 1 [KP]: the space in to-space that requires scanning is partitioned to jobs.
- Option 2 [FDSZ]: work with a markstack that contains all objects to be scanned.
  - Problem reduced to mark and sweep.
  - Use stealable mechanisms for load balancing.
- Option 3 (recall IBM’s mostly concurrent work): work with a markstack that consists of work packets.
Termination Detection

- Recall algorithm:
- Flags “Mark Stack Empty” & “Steal queue Empty” for each process. Flag interrupted is global.
- When a process obtains work it sets interrupted.
- When a process cannot get work it executes a check:
  - Clear interrupted,
  - go over all local flags to check if all queues and stacks are empty.
  - If all are empty and interrupted is clear - termination has been detected.
Termination Detection is Flawed

- Problem: more than one process may decide to detect termination.
- Thus, more than one thread may clear interrupted...
Bad Detection Scenario

P1

P3
Checking

P2

P4
Idle

Interrupt Flag OFF
Bad Detection Scenario

Checking

Interrupt Flag ON
Bad Detection Scenario

Checking

Interrupt Flag OFF
Bad Detection Scenario

P1

P2

P3

P4

Checking

Checking

Interrupt Flag OFF
Bad Detection Scenario

Check completed with finding P4 idle, Report termination

Solution: Add the detector ID to the flag
Modified Detection

- Flags “Mark Stack Empty” & “Steal queue Empty” for each processor. Flag interrupted is global. A field process-id is global.
- When a process obtains work it sets interrupted.
- When a thread cannot get work it execute a check:
  - Writes its thread id in process-id,
  - clears interrupted,
  - goes over all local flags.
  - If all are clear, and interrupted is clear, and process-id was not modified - termination has been detected.
Summary of Parallel Copying

- Copying is similar to mark-and-sweep except for allocation in to-space.

- Issues:
  - space allocation on to-space.
  - Deal with work distribution.
  - Deal with termination detection.
Cache Conscious
Memory Management
Cache-consciousness

- Cache systems overview
- Improving program behavior via GC
  - [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)

- μProc 60%/yr.

- Processor-Memory Performance Gap:
  (grows 50% / year)

- DRAM 7%/yr.
Memory Hierarchy

Processor (with L1 cache 2-5 ns) → External Cache (Kbytes to Mbytes 10-20 ns) → Main Memory (Mbytes to Gbytes 50-100 ns) → Virtual Memory (Gbytes and up 10-100 ms)

Numbers change day to day...
Cache structure

• Large memory divided into lines.
• Small cache - k blocks.
• Mapping of memory lines to cache lines.
• Cache hit.
• Cache miss.
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.
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.
- 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
  • [Huang, Blackburn, McKinley, Moss, Wang, Cheng 04]

• Improving GC behavior:
  • Boehm’s prefetching and lazy sweeping;

• Limitations on the ability to improve data placement
  [Petrark-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 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\((p)\)
Prefetching gray objects

An observation (by measurements): \( \sim \frac{1}{3} \) 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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Mark Time</th>
<th>Sweep Time</th>
<th>Eager Sweep Slowdown</th>
<th>Prefetch Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>gc_bench.java</td>
<td>39%</td>
<td>3%</td>
<td>0%</td>
<td>11%</td>
</tr>
<tr>
<td>gc_bench</td>
<td>49%</td>
<td>3%</td>
<td>0%</td>
<td>13%</td>
</tr>
<tr>
<td>holes_gc_bench</td>
<td>57%</td>
<td>12%</td>
<td>7%</td>
<td>17%</td>
</tr>
<tr>
<td>ptc</td>
<td>27%</td>
<td>0%</td>
<td>0%</td>
<td>4%</td>
</tr>
<tr>
<td>ghostscript</td>
<td>44%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>incremental_ghostscript</td>
<td>39%</td>
<td>9%</td>
<td>17%</td>
<td>4%</td>
</tr>
<tr>
<td>large_ghostscript</td>
<td>8%</td>
<td>6%</td>
<td>3%</td>
<td>1%</td>
</tr>
</tbody>
</table>

## Table 2: HP PA-RISC Relative Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Mark Time</th>
<th>Sweep Time</th>
<th>Eager Sweep Slowdown</th>
<th>Prefetch Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>gc_bench</td>
<td>45%</td>
<td>3%</td>
<td>-1%</td>
<td>11%</td>
</tr>
<tr>
<td>holes_gc_bench</td>
<td>40%</td>
<td>36%</td>
<td>34%</td>
<td>9%</td>
</tr>
<tr>
<td>ghostscript</td>
<td>26%</td>
<td>4%</td>
<td>3%</td>
<td>8%</td>
</tr>
</tbody>
</table>
Roadmap

• Cache systems overview
• Improving program behavior via GC
  • [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].
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, $f:O\rightarrow\mathbb{N}$.
- **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 $\approx$ 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 ≈ 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 $\varepsilon$ 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 $>2$ lines)
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:**
  - $t \cdot k$ 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…
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.)
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: $n^{(1-\varepsilon)}$, Possible: $\frac{n}{c \log n}$
Comparison to Other Problems

• Data Placement:
  • Inapproximable
    • $\frac{n}{c \log n}$-approximation algorithm

• Famous problems with similar results:
  • Minimum graph coloring
  • Maximum clique