Analysis of pure methods using Garbage Collection

Authors:

Erik Österlund and Welf Löwe

Linnaeus University, Sweden

Motivation

- CPU clock rates are not increasing
- Need for other ways to increase performance
- Parallelization is a promising option

Motivation (cont.)

- Very good potential for parallelization in hardware
- Less good potential for parallelization in software in practice
- Parallel programs are inherently complex and time consuming
- Automatic parallelization is easy to use but had little success in object oriented programming

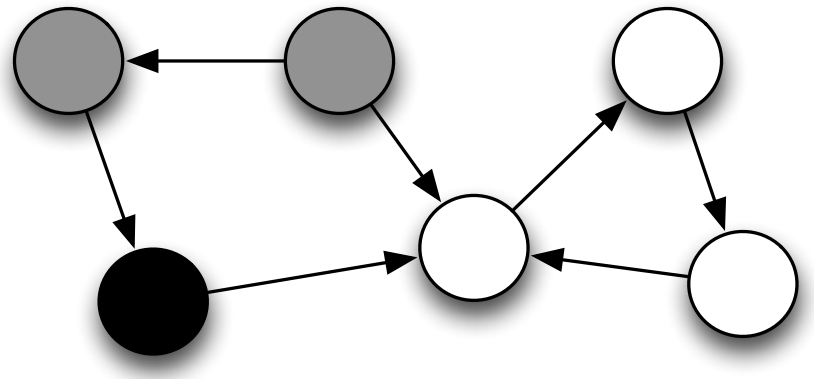
Contents

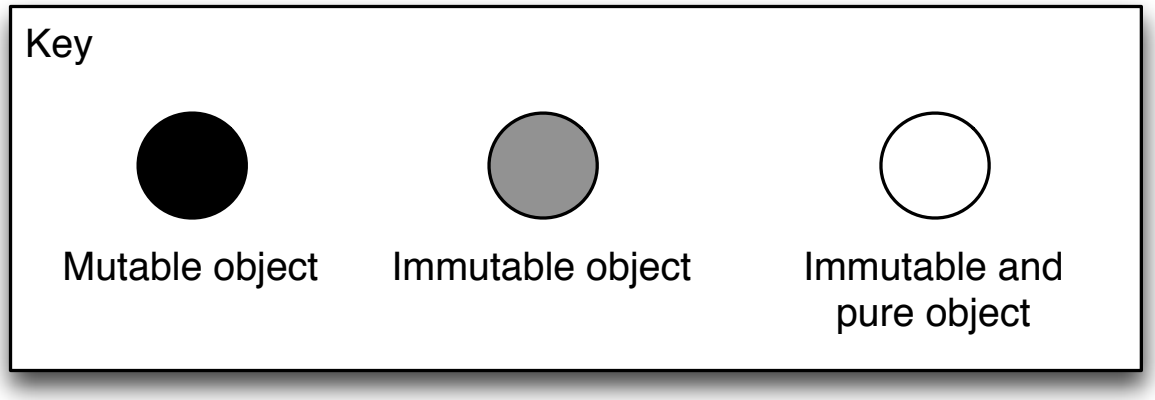
- Basic idea and notions
- Pure object analysis
- Garbage collection and traversal strategies
- Final solution
- Evaluation
- Conclusion and Future work

Pure objects

- An immutable object does not change its attributes
  - I/O operations count as mutations of object's state
- A **pure object** is immutable and can only transitively reach other immutable objects
- Insight:
  - A pure object can not change global state
  - Method invocations on pure objects are called **pure methods** and can run in parallel

Pure Objects





Pure object analysis

- Existing automatic parallelization – based on something like purity analysis – is mostly done statically, and has not succeeded for OOP
- My hypothesis:
  - Analysis of OOP has to regard too many dependencies if done conservatively.
  - Optimistic, dynamic analysis does not over-approximate dependencies and is more precise
  - Allows for better parallelization

Basic idea

- Use a garbage collector to guess pure objects (optimistic, dynamic analysis)
  - Pure for some time, not necessarily always
- Roll back if guess was wrong using careful write-protection
- Idea: merge 3 algorithms
  - Classic GC algorithm
  - Tarjan's algorithm
  - Purity detection algorithm
- Test of the idea:
  - Proof of concept implementation for evaluation

Notions

- Strongly connected components (SCC)
  - partitioning of graph in min set of nodes so all nodes can reach every other node in its set
- Objects and references, cells and pointers, nodes and edges
- Cell properties: mutable, dirty, pure

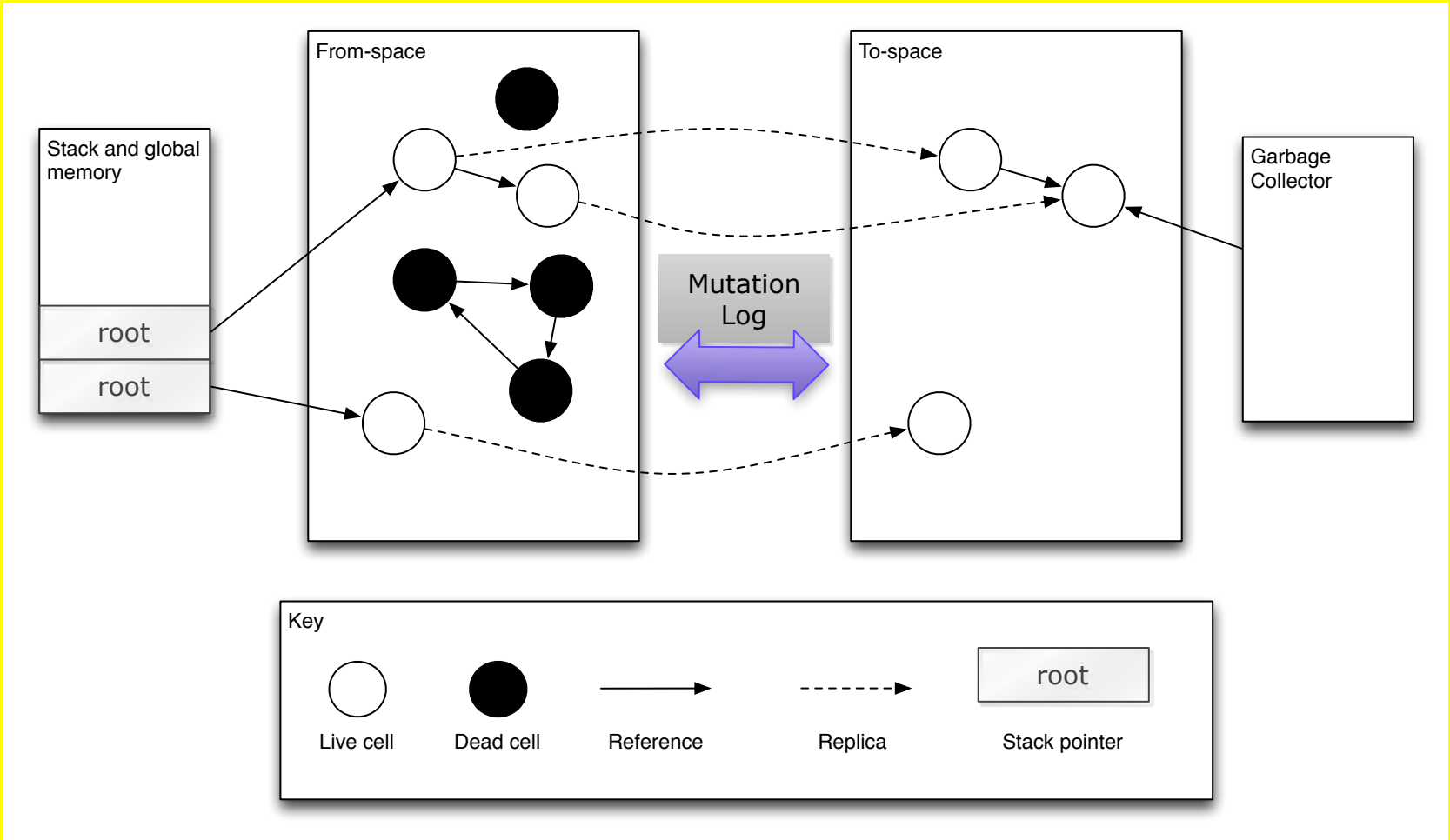
Pure object analysis

- Condense object graph to SCCs
  - directed acyclic graph
  - Modified Tarjan's algorithm to find SCCs
  - Need a linear O(|E|+|V|) algorithm
- Traverse the condensed graph in DFS order, propagate "dirty" property up towards roots from mutable nodes
- Nodes that are not dirty are guessed to be pure

Garbage collection

- Replicating garbage collector
  - Variant of copying GC
  - Mutators access from-space
  - Collector synchronizes from- and to-space during GC
- Why replicating GC
  - Allows for pointer reversal in to-space (required for analysis)
  - Traverse DFS order, needed for merging Tarjan's algorithm with GC
  - Replicating live objects (between semi-spaces)
- Parallel garbage collection
  - Fast, low mutator delay due to GC
  - Needs a mutation log for synchronization (thread-local)
  - Write barrier for all fields, not only pointers
  - Exploited by analysis – guess mutable objects (major reason why replicating GC algorithm was chosen

Replicating GC



Traversal strategies

- Pointer reversal vs stack-based traversal
  - Pointer reversal – no stack overflow, impossible for some cells for technical reasons.
  - Stack – faster in some VMs because of technical reasons, lower memory footprint
  - Hybrid currently chosen using stack mostly, but can fall back to pointer reversal

Tarjan modifications

- Stack (for determining nodes in same SCC)
  - Removed, flag bit used instead
- Index field (indicating DFS order)
  - Removed, memory address of cell is in DFS order with replicating GC
- Lowest field (first node in SCC DFS order)
  - Removed; shared with replica pointer needed by replicating GC

Memory allocation

- Contiguous heap
- Contiguous memory allocation
- Thread local allocation buffers
- Small synchronization times (no locks)
  - Soft real-time GC (not going to prove hard real-time!!!)
- Allocation speed fast

Final solution

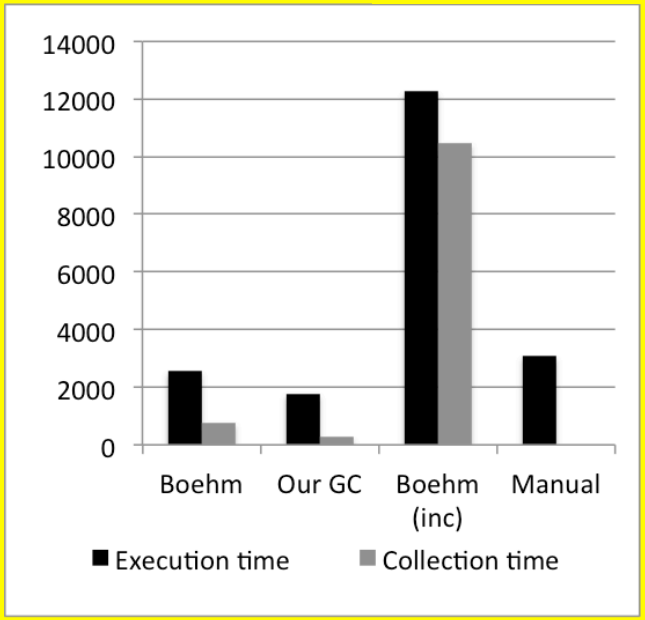
- Optimistic purity analysis
  - Replicating garbage collector
  - Merged with Tarjan's algorithm
  - Purity analysis on the way
- Low overhead in time and memory
  - Total 1 DFS pass O(|V|+|E|)
  - Overhead of analysis (purity analysis + Tarjan's) insignificant
  - 1 pointer word memory overhead per cell using smart optimizations applicable only to replicating GC

Evaluation

- Proof of concept in C (with runtime-system)
- GCBench
- Homegrown parallelization benchmark using our GC
- Claims are toned down in this paper; major contribution is the concept

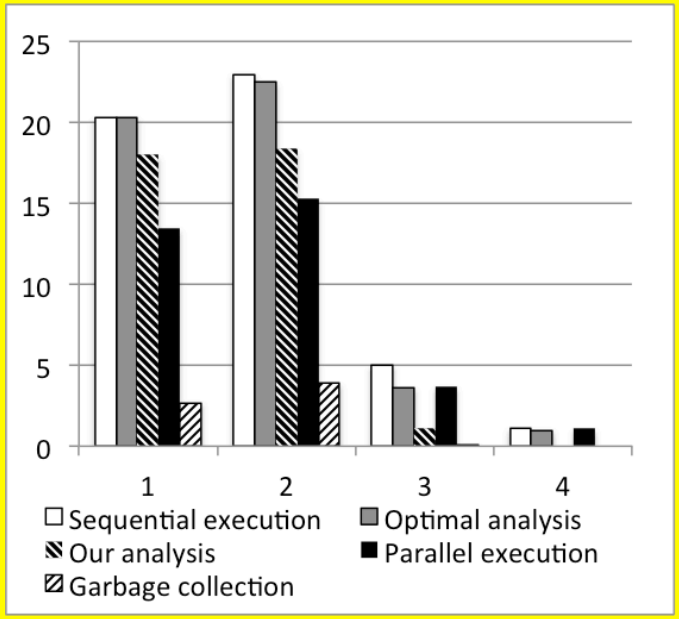
GCBench

GC	Execution time	Collector overhead
Boehm	2,555 ms	774 ms (30 %)
Our GC	1,781 ms	263 ms (13 %)
Boehm (inc)	12,255 ms	10,474 ms (86 %)
Manual	3,083 ms	N/A



- Compared to Boehm's STW collector
- Heap size: 512 MB
- Boehm's collector overhead measured as difference between Boehm's STW and mutator time of our GC
- **257 ms collector overhead without analysis (almost the same)**
- Limitation: Suboptimal to compare conservative mark & sweep against parallel replicating GC

Parallelization bench



- Traverse tree BFS order, mutations in all nodes except leaves
- Leaves multiply matrices together
- Different runs with different tree sizes
- Occasionally mutates state if derivative deviates due to lacking floating point precision (which static analysis can not find easily)
- Dynamically parallelizes and rolls back if wrong

Conclusion

- Implemented GC and purity analysis
- Performance of proof of concept GC is comparable to Boehm's GC
- Optimistic purity analysis does not take extra time and is directly accessible for parallelization
- Scales well with multiple threads

Future work

- Detecting independent sub-graph pairs
- Generational garbage collection
- JIT parallelization transformations using analysis
- Integrate to VM for better evaluation (currently in progress, integrated to OpenJDK 7 hotspot JVM)
- In general let runtime system exploit GCs for more than collecting garbage

Linnéuniversitetet

Questions?

Linnéuniversitetet