Myths and Realities: The Performance Impact of Garbage Collection

Tapasya Patki

February 17, 2011

1 Motivation

Automatic memory management has numerous software engineering benefits from the developer's point of view. Garbage collection frees the developer from the burden of tracking memory, prevents security violations and facilitates modularity. However, it incurs a space and runtime performance penalty, and the performance impact of garbage collection remains a matter of debate. This paper analyzes the behavior of whole heap and generational garbage collection algorithms and attempts to answer the following questions:

- Is GC a good idea?
- How often should we garbage collect?
- How should we garbage collect (Whole-heap or generational)?
- Does GC affect instruction throughput and locality?
- Does the underlying processor architecture influence GC performance?
- How sensitive are collectors to heap size?

2 Background

Each GC algorithm is comprised of an allocation strategy and a collection strategy. There are two possible allocation strategies:

- Contiguous allocation, in which we allocate a new object by appending a bump pointer by the size of the object, and
- Free-list allocation, which divides memory into size classes, and allocates the object in the smallest size class. This is similar to binning.

Contiguous allocation is faster as it does not have to scan multiple lists before allocating. Also, it has higher spatial locality of reference. Free-list allocation can lead to internal fragmentation, but has the advantage that it can be compacted faster. Collection strategies describe how allocated memory is reclaimed. The standard implementations are:

- Tracing collection, in which we identify the "live" objects by taking a transitive close of the roots and the remembered-set. Memory is reclaimed by using copying live data out of the space. This is similar to the idea of copying collection.

- Reference Counting, in which the number of references each object are counted, and the objects with no references are freed.

The tables below illustrate how various GC algorithms (whole-heap and generational) can be implemented using the aforementioned allocation and collection strategies.

Algorithm	**Allocator**	**Collector**
SemiSpace	Contiguous	Tracing
MarkSweep	Free-list	Tracing
RefCount	Free-list	Reference Counting

Algorithm	**Nursery**	**Mature**
GenSS	SemiSpace	SemiSpace
GenMS	SemiSpace	MarkSweep
GenRC	SemiSpace	RefCount

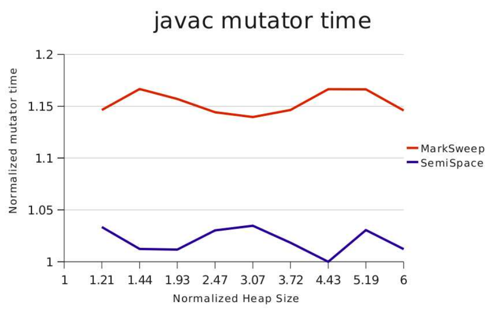
In case of generational garbage collectors, a *write-barrier* needs to be inserted for every pointer store to keep track of pointers that go from mature-space to the nursery. This incurs an extra overhead.

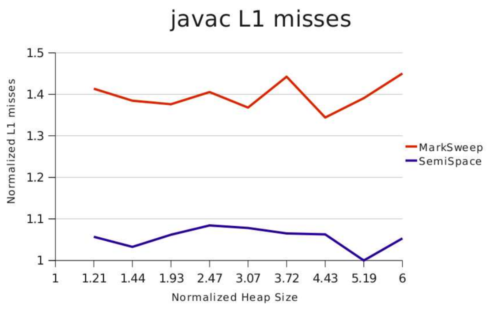
3 Methodology

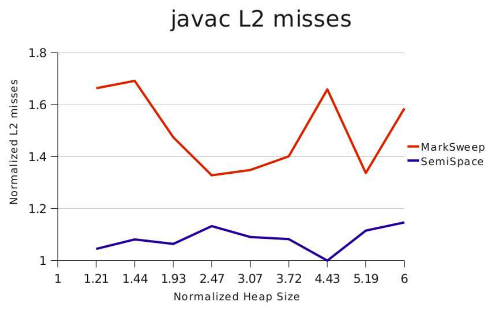
Each program is divided into two phases- a *mutator* phase which runs the application code, and a *garbage collection* phase. The application code may consist of some memory management activities like allocation sequences and write barriers. Authors use the Java Memory Management Toolkit (MMTk) in the IBM Jikes RVM environment. IBM Jikes RVM supports high-performance pseudo-adaptive compilation and is a Java-in-Java design. MMTk implements garbage collection policies using the allocation and collection strategies described in the last section, and supports inlining of write-barriers. This environment provides for fair experimentation and an apples-to-apples comparison of garbage collection algorithms. Authors measure the performance impact of the six algorithms described in the last section on the three architectures shown in the table below. They use the SPEC JVM benchmarks for the experiments.

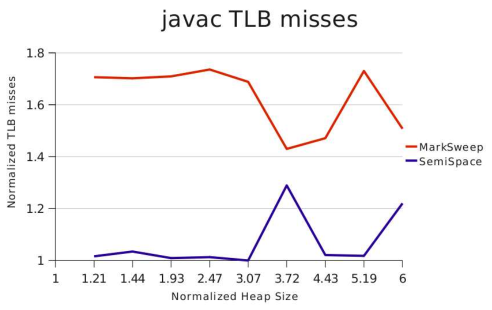
Arch	**CPU Freq**	**RAM**	**L1**	**L2**
Athlon	1.9Ghz	1GB	64K both	512K
P4	2.6GHz	1GB	8K D, 12K I	512K
PPC	1.6GHz	768MB	32K D, 64K I	512K

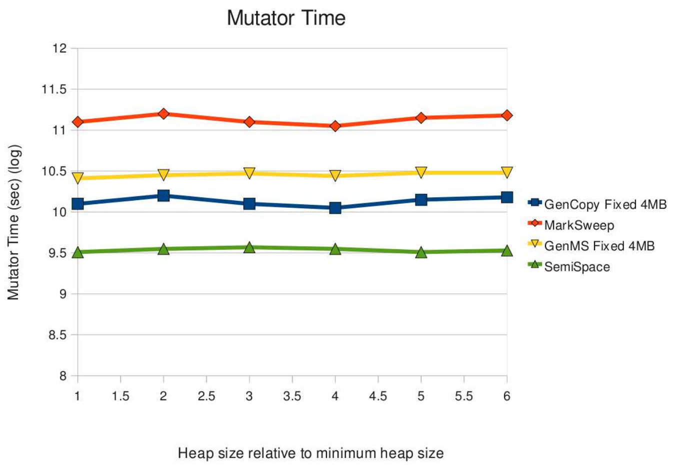
4 Results and Observations





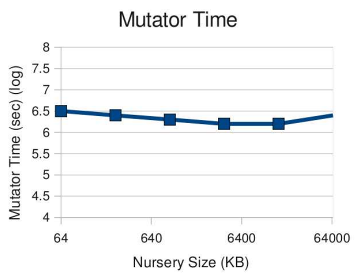


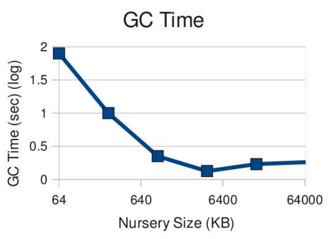




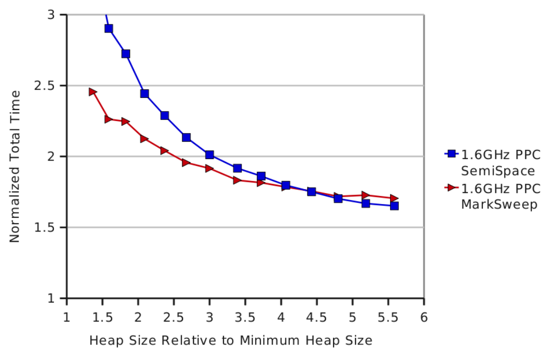
- Contiguous is better than Free-list
  - Allocation is 11% faster, total improvement 1%
  - Locality improves mutator performance by 5-15% (SS vs MS)
- Tracing is usually better than Reference Counting
  - Only live objects are touched
  - Locality improves
  - RC can be useful for mature objects, when most of the heap consists of live objects
- Generational better than Whole-heap

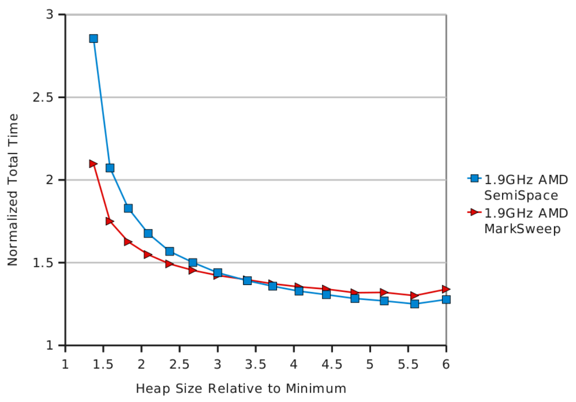
- Write-barrier over head is 1-14%, 3.2% average
- Collection time benefits outweigh the write-barrier overhead
- Locality of nursery: spatial
- Locality of mature objects: temporal

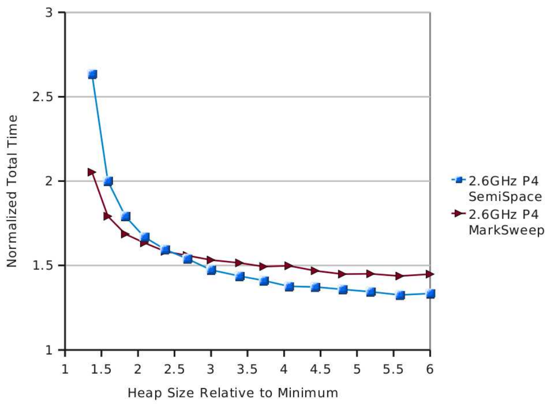




- Nursery Size
  - Fixed overhead of scanning roots (about 64KB)
  - Need not be matched to L2 cache size (512KB)
  - Should depend on fixed overhead and not L2 (4-8MB is ideal for SPEC JVM)
  - If large, collection cost degrades performance (>8MB)
- Sensitivity to Heap Size
  - Determines collection frequency
  - Small heaps: MarkSweep, Modest to large heaps: SemiSpace







- Influence on Architecture
  - Crossover point: When SemiSpace outperforms MarkSweep
  - Limited space versus Locality tradeoff