Collections Classes for Real-Time and High-Performance Applications

July 4, 2005

Jean-Marie Dautelle

Introduction

The real-time specification for Java (RTSJ) aims to make your code more time predictable. But, this predictability can easily be ruined by using standard library classes. In this presentation we will look at potential problems with the standard Java collection framework and how these problems can be solved by using alternative implementations such as the one provided by the open-source *Javolution* library (http://javolution.org)

Standard Collections Overview

- Interface based.
  - **java.util.List**
  - **java.util.Map**
  - **java.util.Set**
  - **…**
- Interface approach allow for “drop-in” replacement.
- Generic since Java™ 5.0
- Oriented toward throughput
- Time predictability and RTSJ (Real Time Specification for Java) not taken into consideration by the implementation.

Standard Collections Timing Issues

- Users may encounter unexpected large delays due to:
  - Large arrays being allocated and copied (e.g. **ArrayList** capacity increase).
  - Sudden burst of computation (e.g. **HashMap** rehashing all its map entries).
  - Long garbage collection pauses (full GC) due to memory fragmentation when large arrays are allocated (e.g. **ArrayList**)

Standard Collections Memory Issues

- Memory allocation might be performed surreptitiously after the collection creation and cause RTSJ memory clashes:
  - **HashMap.put(key,value)** may create new entries allocated from the current memory area, resulting in memory clash when executing in ScopedMemory and the map is in ImmortalMemory/HeapMemory
  - Iterator objects may be internally allocated, e.g. **Collection.addAll(Collection), Map.putAll(Map)** causing memory leak when executing in ImmortalMemory
  - Objects may be allocated at first use only (lazy initialization), e.g. **Map.keySet(), Map.values()** causing further unexpected illegal access errors.

Javolution Collections Overview

- Provides very few classes but they are substitute for most of java.util.* collection implementations. For example:
    - **IdentityHashMap** would be a FastMap with an **identity** key comparator.
    - Both **HashMap** and **LinkedHashMap** can be replaced by a FastMap (same for **HashSet/LinkedHashSet** and FastSet)
- Implements standard collection interfaces.
- Generic when built with the ant target 1.5, but can also be built for any Java VM/Compiler or even J2ME.

Time-Predictability

- Collection capacity increases smoothly.
  - Small increments (avoid large arrays allocations).
  - Pre-allocation can be performed using profiling information from previous executions (Ref. **AllocationProfile**)
- No internal array resize or copy.
  - **FastTable** (random access list) uses multi-dimensional arrays to avoid resizing and copying its internal arrays.
  - Allocated arrays are small to avoid memory fragmentation.
- No rehashing.
  - Map entries have their own entry table. When the size increases beyond capacity, new (larger) tables are allocated for the new entries (the old entries are not moved and do not need rehashing).

Memory Allocation Policy

- No memory allocation ever performed unless the collection size exceeds its capacity which may be specified at creation.
  - Collections maintain internal pools of entries (map) or nodes (list). When an entry/node is removed from the collection, it is automatically restored to the pool.
  - The initial capacity determinates the number of entries/nodes allocated at creation.
  - Lazy initialization is forbidden.
  - Iterations can be performed without allocating iterator objects (direct record iterations)

RTSJ - Compliance

- No memory clash (e.g. IllegalAssignmentError) regardless of the current memory area or the operation being performed.
- Collections can be allocated in ImmortalMemory (e.g. global static collections) and accessed by all threads including NoHeapRealtimeThread
- Integrated with Javolution real-time framework which allows for transparent object recycling. Throw-away collections can be pre-allocated in ImmortalMemory (usable by all threads) and automatically recycled (necessary as ImmortalMemory is never garbage collected).

Concurrency and Synchronization

- Collection classes support concurrent access and iterations without synchronization as long as the collection records are not removed (e.g. FastMap look-up table).

- To keep read access unsynchronized when records are deleted, applications may either:
  - Replace the whole collection.
  - Or (better) set the record value to null instead of removing it.

- A read-only view on any collections is also provided (unmodifiable() method). This view is thread safe when the collection is thread-safe (records not removed).

Others.

- Support custom values comparator (collections) and key/value comparator (maps). For example, a **lexical** comparator can be used to retrieve objects based upon their character sequence content (regardless of their actual type).
- BSD-License allows applications to reuse/modify the collection source code as long as the copyright header is kept intact.

Conclusion

- Javolution collections behavior is highly time predictable (in the micro-second range).
- By favoring an incremental approach and by allocating small objects, the work and performance of concurrent / incremental garbage collectors is significantly improved.
- Finally, these collections are fast, very fast (as shown in the performance report in annex) proving that “real-time” can sometimes be “real-fast”.

Biography

- Jean-Marie holds a master degree of electrical engineering (Orsay University, France) and a post graduate degree in Information Processing and Simulation.
- Jean-Marie has been with Raytheon in Marlborough for 7 years.
- He is the project owner and main author of two open-source projects: ***Javolution*** (http://javolution.org) and **JScience** (http://jscience.org).
- Jean-Marie work has been published in several magazines such as Dr Dobbs Journal and the Java Developer Journal.

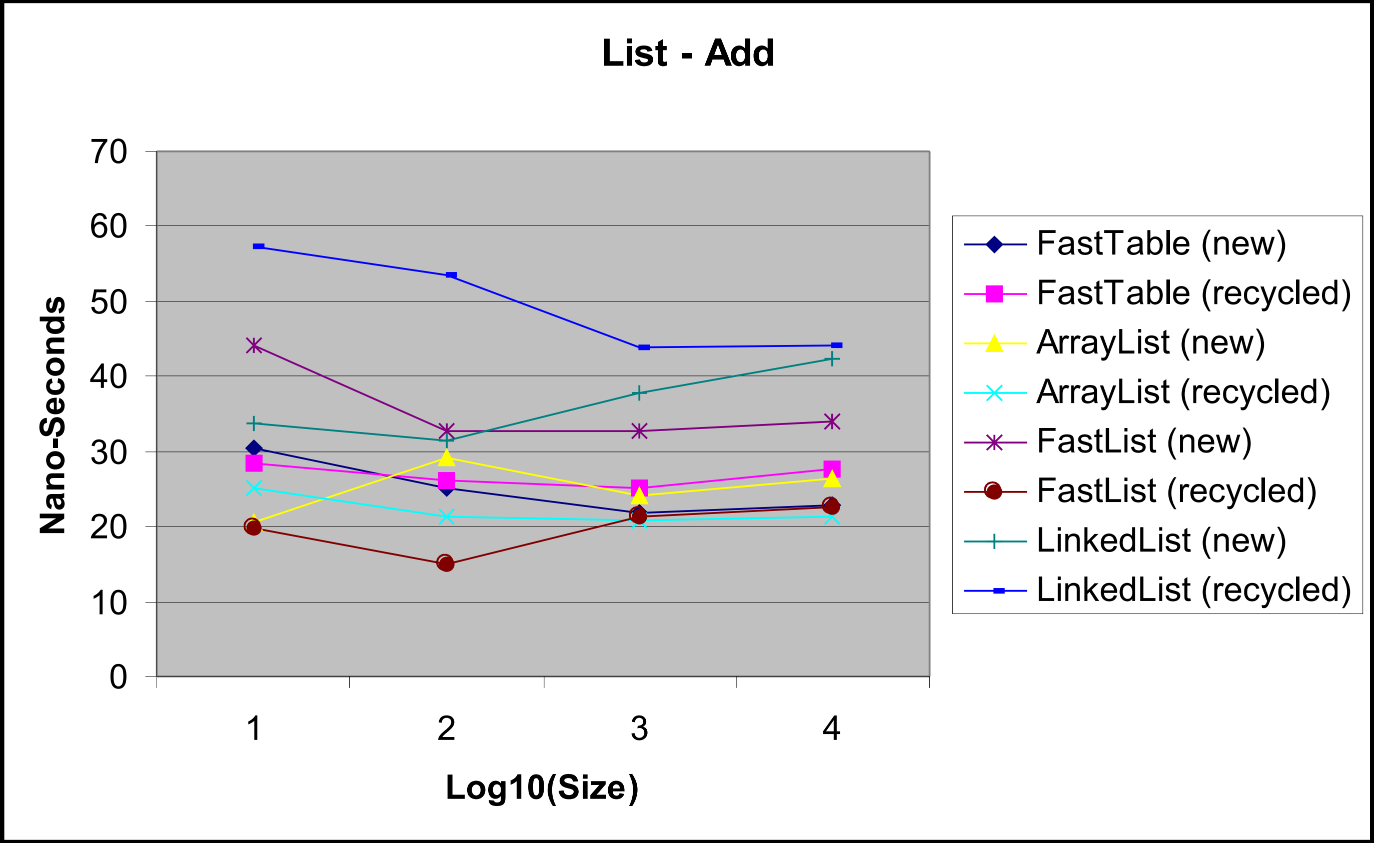
Annex: Performance Report

- Relative performance of Javolution Collections classes versus Standard Collections classes.
- The platform is a Single-CPU Intel Pentium 4 3.20GHz running Linux 2.4
- The benchmark source code and executable are available from the ***Javolution*** home page.

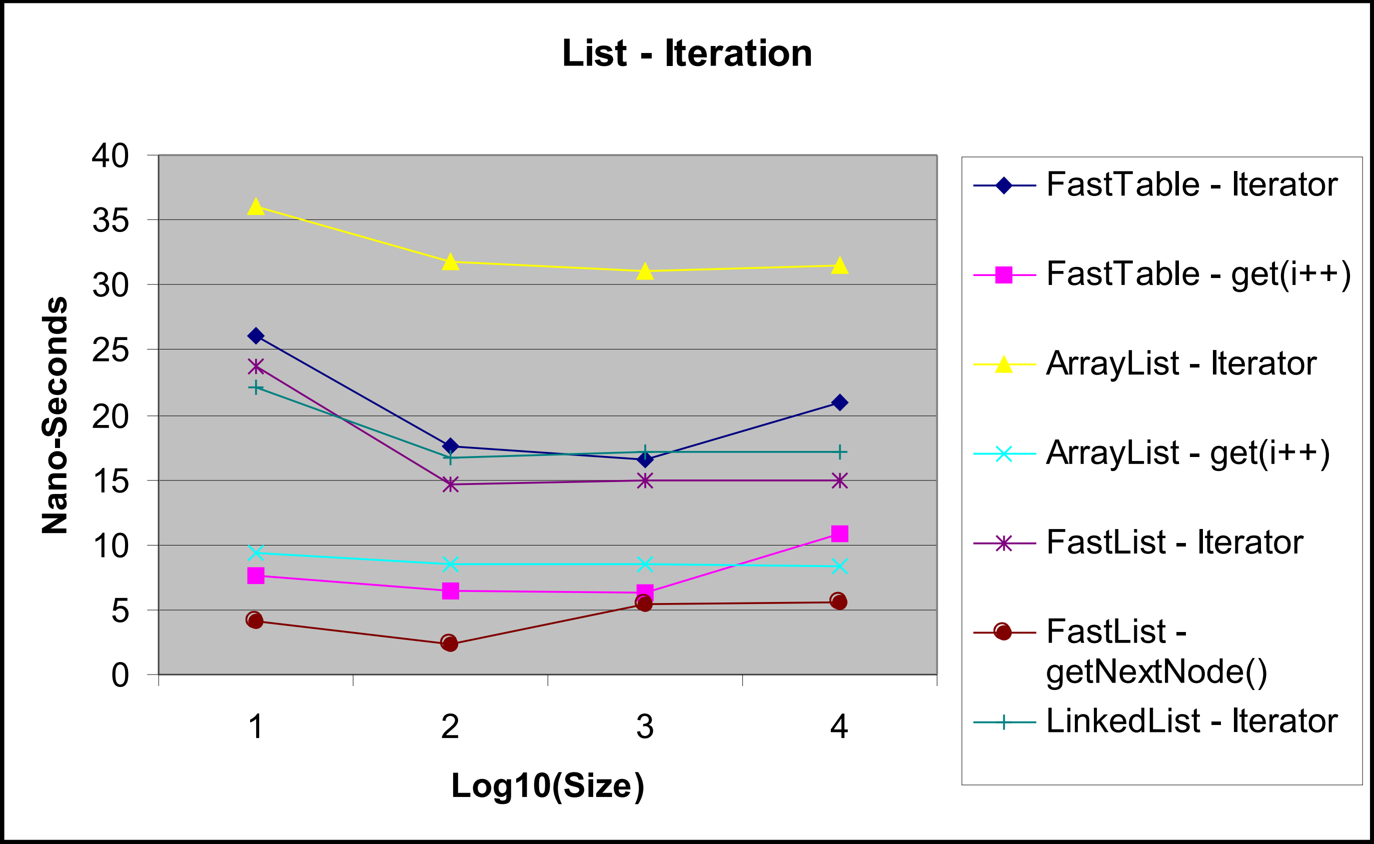
  - Add (new) - The collection is created (using the new keyword), populated, then discarded (throw-away collections).

  - Add (recycled) - The collection is cleared, populated, then reused (static collections or throw-away collections in PoolContext).

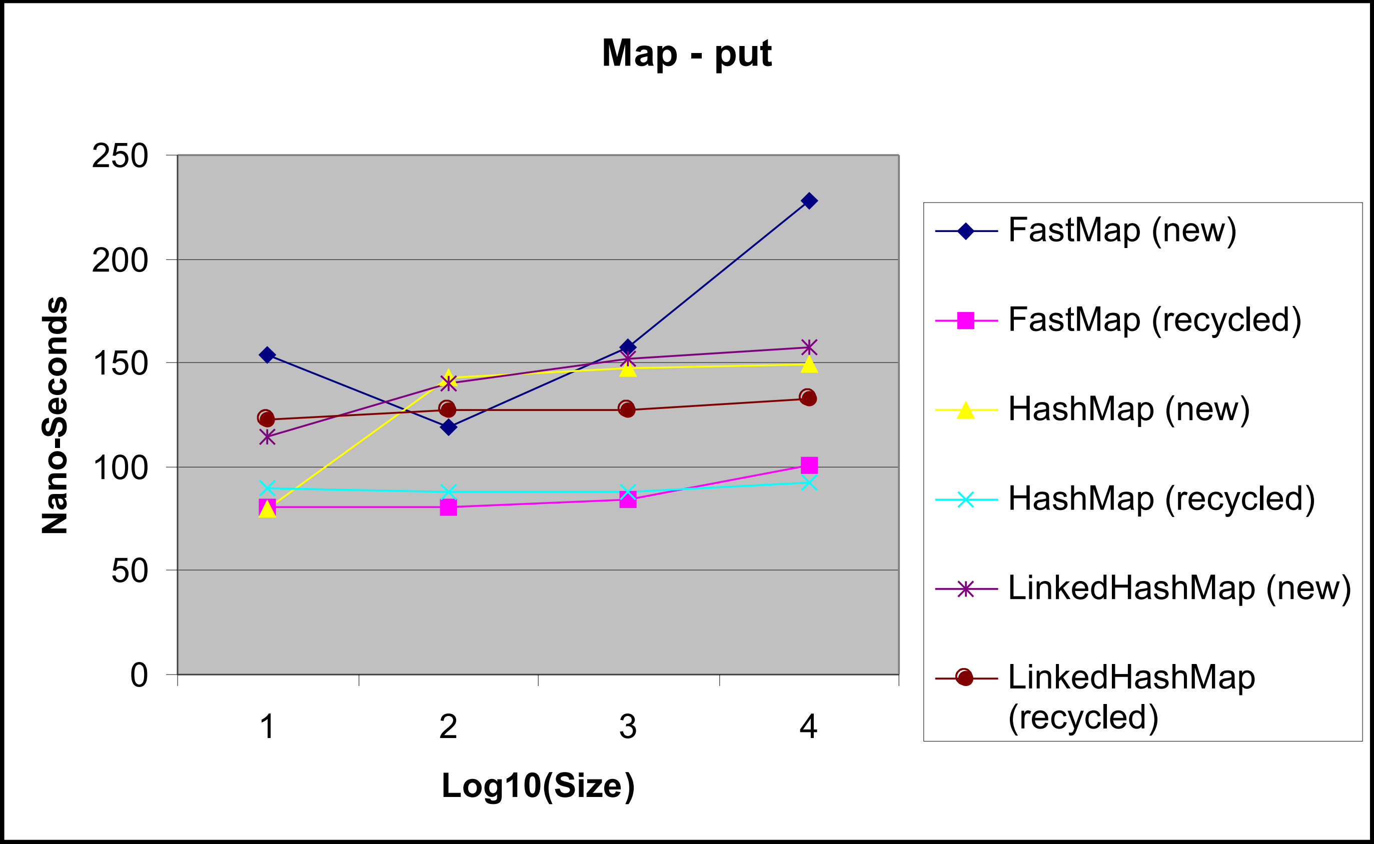
List – Add Performance



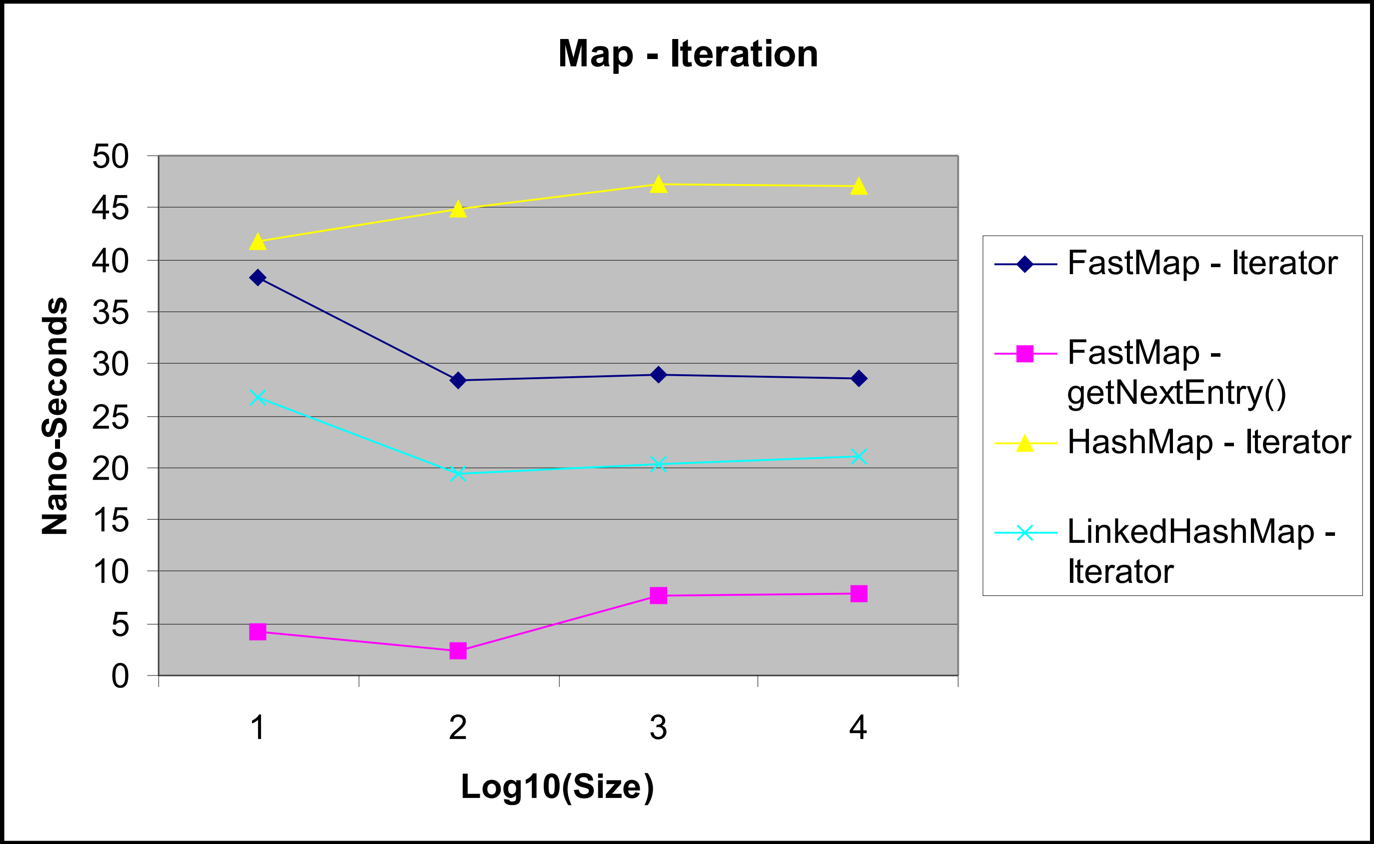
List – Iteration Performance



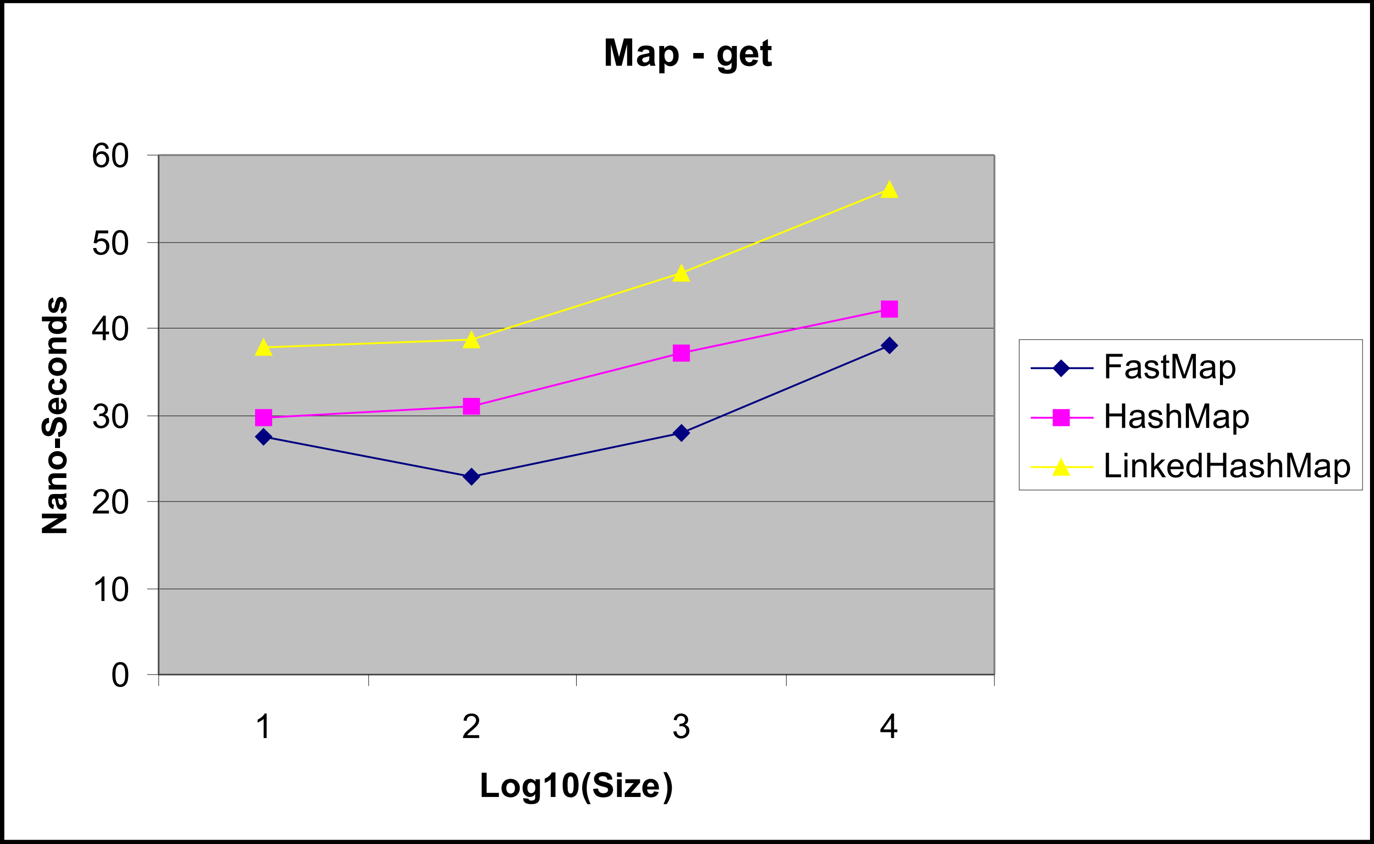
Map - Put Performance



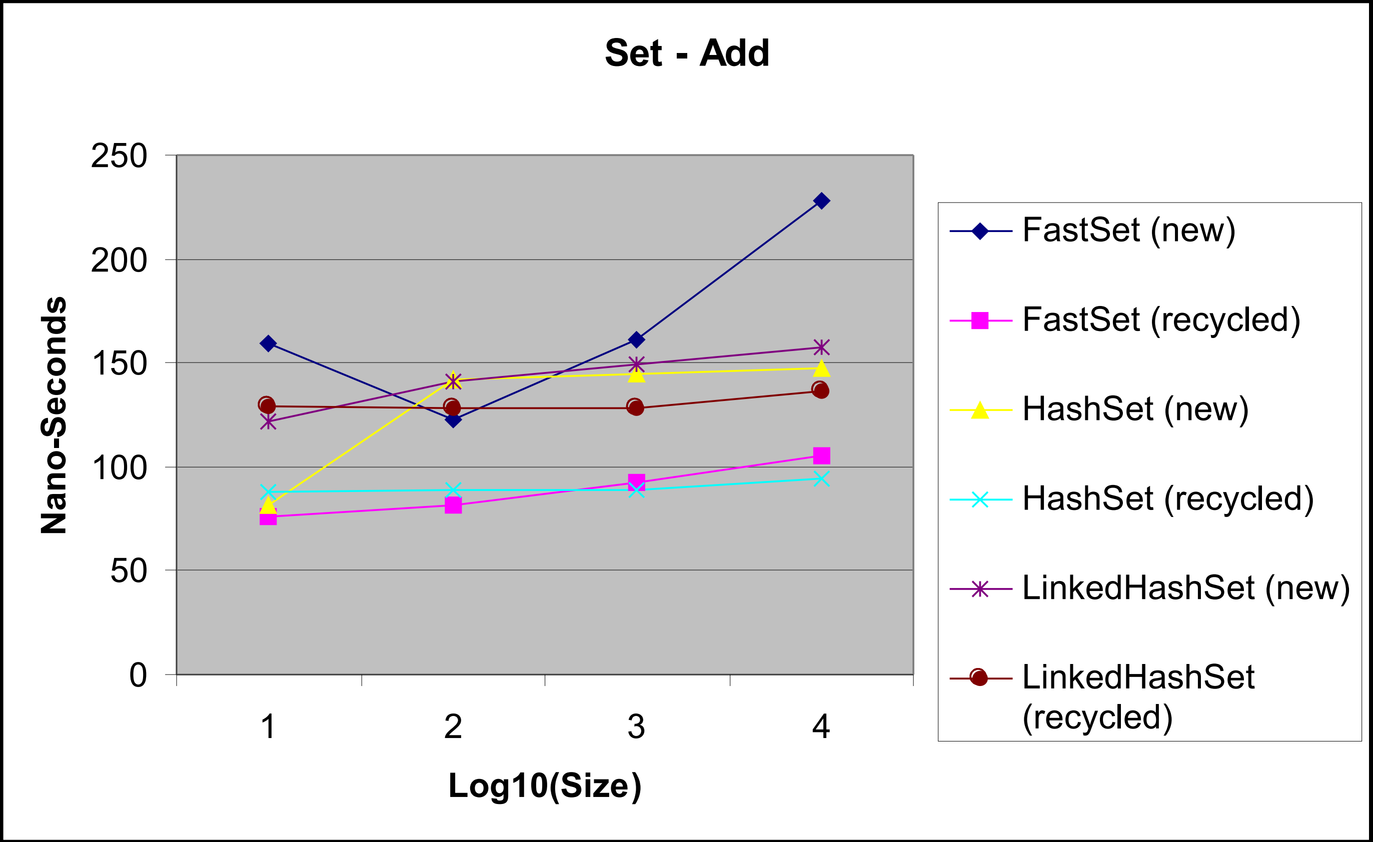
Map – Iteration Performance



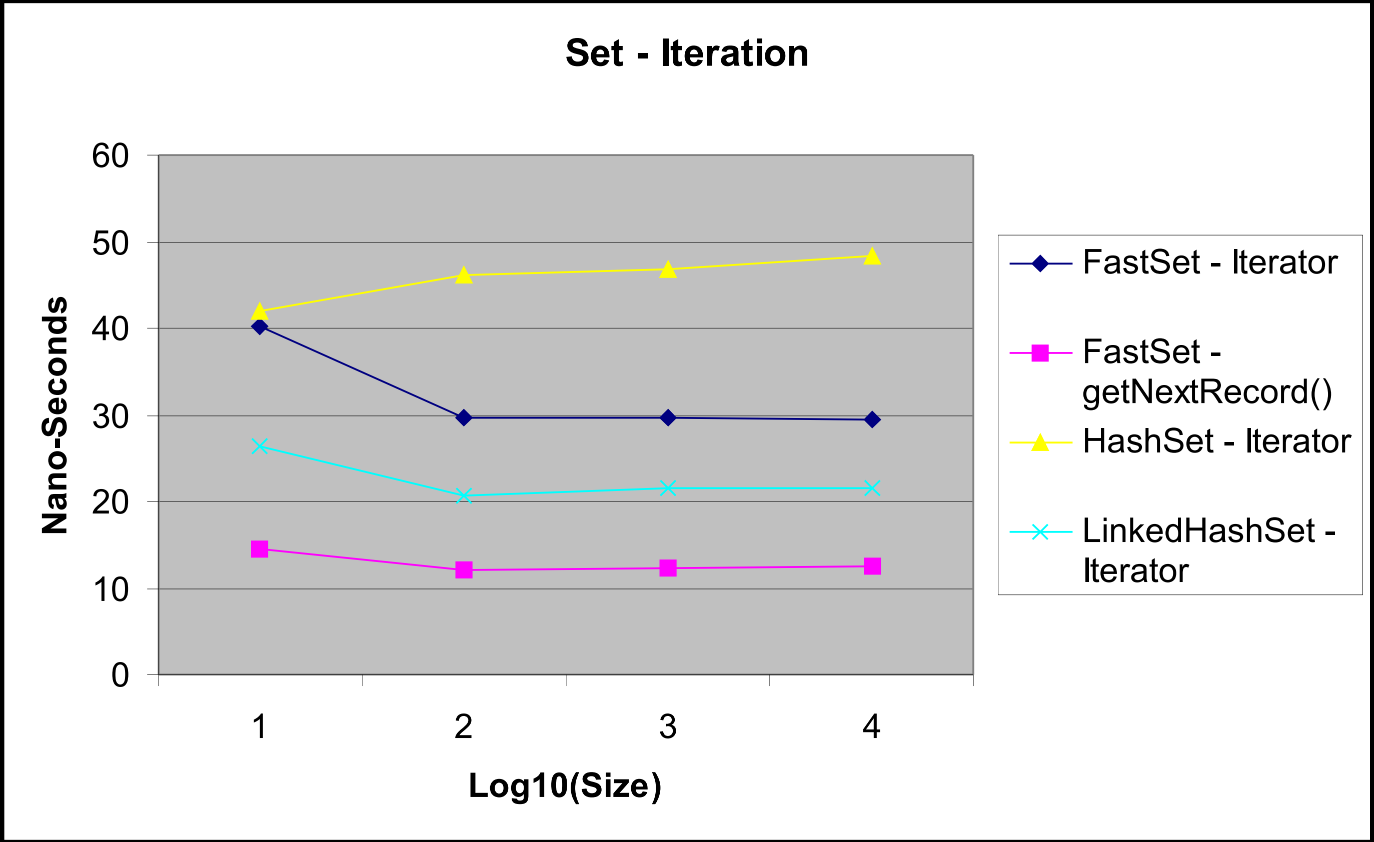
Map – Get Performance



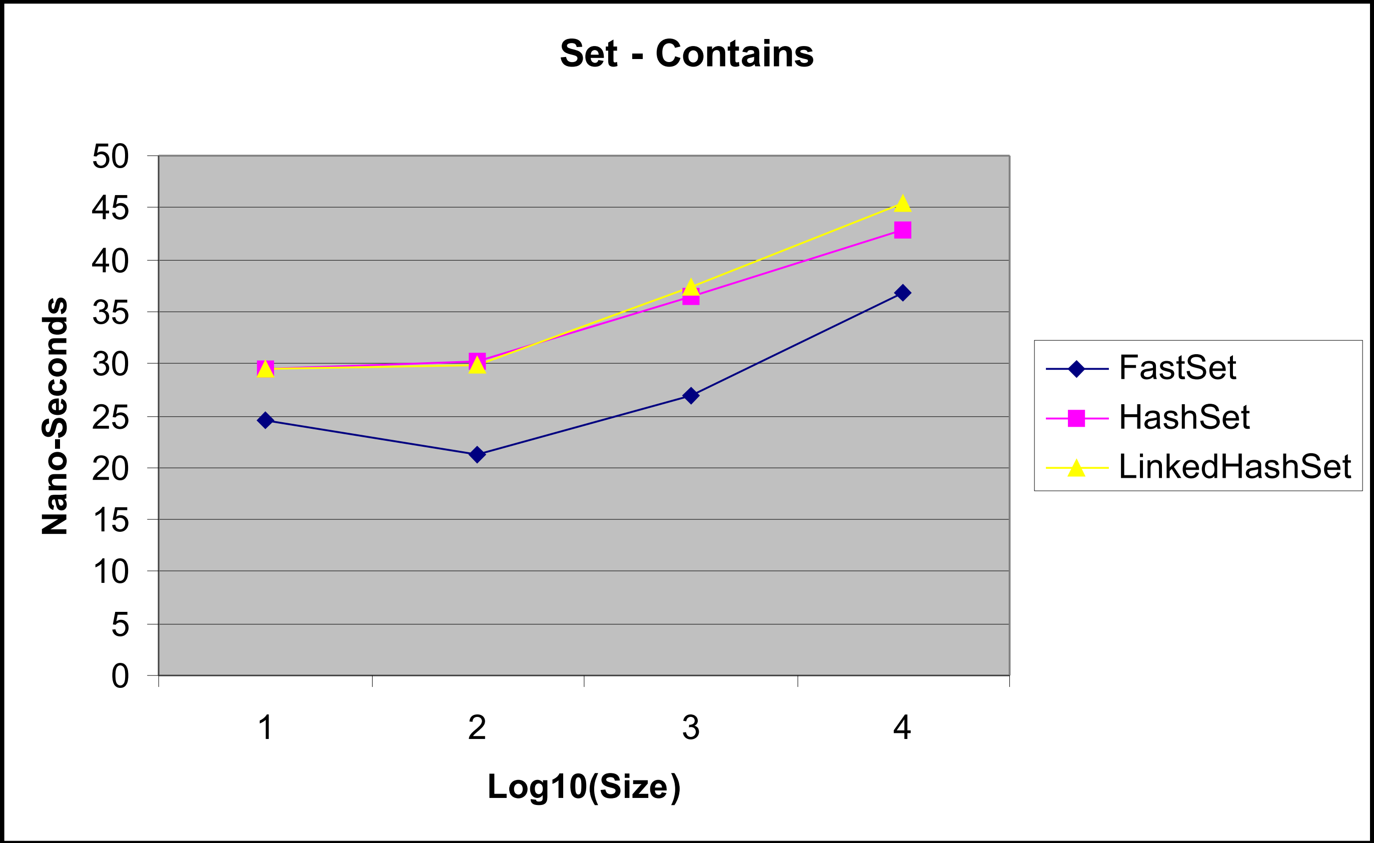
Set – Add Performance



Set – Iteration Performance



Set – Contains Performance


Set - Contains
Nano-Seconds
Log10(Size)
FastSet
HashSet
LinkedHashSet