Garbage Collection and Memory Architecture

By

Mohit Kumar

History of Garbage Collection

- The Java language may be the most widely used programming language to rely on garbage collection, but it is by no means the first. Garbage collection has been an integral part of many programming languages, including Lisp, Smalltalk, Eiffel, Haskell, ML, Scheme, and Modula-3, and has been in use since the early 1960s.
- The benefits of garbage collection are indisputable -- increased reliability, decoupling of memory management from class interface design, and less developer time spent chasing memory management errors.

History of Garbage Collection

- The well-known problems of dangling pointers and memory leaks simply do not occur in Java programs. (Java programs can exhibit a form of memory leak, more accurately called unintentional object retention, but this is a different problem.)
- However, garbage collection is not without its costs -- among them performance impact, pauses, configuration complexity, and nondeterministic finalization.

History of Garbage Collection

- An ideal garbage collection implementation would be totally invisible –
  - there would be no garbage collection pauses,
  - no CPU time would be lost to garbage collection,
  - the garbage collector wouldn't interact negatively with virtual memory or the cache,
  - and the heap wouldn't need to be any larger than the *residency* (heap occupancy) of the application.
  - Of course, there are no perfect garbage collectors, but garbage collectors have improved significantly over the past ten years.

History of Garbage Collection

- The 1.3 JDK includes three different garbage collection strategies; the 1.4.1 JDK includes six, and over a dozen command-line options for configuring and tuning garbage collection. How do they differ? Why do we need so many?
- The various garbage collection implementations use different **strategies** for identification and reclamation of unreachable objects, and they interact differently with the user program and scheduler.
- Different sorts of applications will have different requirements for garbage collection –
  - real-time applications will demand short and bounded-duration collection pauses,
  - whereas enterprise applications may tolerate longer or less predictable pauses in favor of higher throughput.

History of Garbage Collection

- How does garbage collection work?
  - There are several basic strategies for garbage collection:
  - reference counting, mark-sweep, mark-compact, and copying.
  - In addition, some algorithms can do their job *incrementally* (the entire heap need not be collected at once, resulting in shorter collection pauses), and some can run while the user program runs (*concurrent* collectors).
  - Others must perform an entire collection at once while the user program is suspended (so-called *stop-the-world* collectors).
  - Finally, there are hybrid collectors, such as the generational collector employed by the 1.2 and later JDKs, which use different collection algorithms on different areas of the heap.

History of Garbage Collection

- When evaluating a garbage collection algorithm, we might consider any or all of the following criteria:
  - **Pause time.** Does the collector stop the world to perform collection? For how long? Can pauses be bounded in time?
  - **Pause predictability.** Can garbage collection pauses be scheduled at times that are convenient for the user program, rather than for the garbage collector?
  - **CPU usage.** What percentage of the total available CPU time is spent in garbage collection?
  - **Memory footprint.** Many garbage collection algorithms require dividing the heap into separate memory spaces, some of which may be inaccessible to the user program at certain times. This means that the actual size of the heap may be several times bigger than the maximum heap residency of the user program.

History of Garbage Collection

- **Virtual memory interaction.** On systems with limited physical memory, a full garbage collection may fault nonresident pages into memory to examine them during the collection process. Because the cost of a page fault is high, it is desirable that a garbage collector properly manage locality of reference.
- **Cache interaction.** Even on systems where the entire heap can fit into main memory, which is true of virtually all Java applications, garbage collection will often have the effect of flushing data used by the user program out of the cache, imposing a performance cost on the user program.
- **Effects on program locality.** While some believe that the job of the garbage collector is simply to reclaim unreachable memory, others believe that the garbage collector should also attempt to improve the reference locality of the user program. Compacting and copying collectors relocate objects during collection, which has the potential to improve locality

History of Garbage Collection

- **Compiler and runtime impact.** Some garbage collection algorithms require significant cooperation from the compiler or runtime environment, such as updating reference counts whenever a pointer assignment is performed. This creates both work for the compiler, which must generate these bookkeeping instructions, and overhead for the runtime environment, which must execute these additional instructions. What is the performance impact of these requirements? Does it interfere with compile-time optimizations?

- Regardless of the algorithm chosen, trends in hardware and software have made garbage collection far more practical. Empirical studies from the 1970s and 1980s show garbage collection consuming between 25 percent and 40 percent of the runtime in large Lisp programs. While garbage collection may not yet be totally invisible, it sure has come a long way.

History of Garbage Collection

- Reference counting
  - The most straightforward garbage collection strategy is reference counting. Reference counting is simple, but requires significant assistance from the compiler and imposes overhead on the *mutator* (the term for the user program, from the perspective of the garbage collector).
  - Each object has an associated reference count -- the number of active references to that object. If an object's reference count is zero, it is garbage (unreachable from the user program) and can be recycled.
  - Every time a pointer reference is modified, such as through an assignment statement, or when a reference goes out of scope, the compiler must generate code to update the referenced object's reference count. If an object's reference count goes to zero, the runtime can reclaim the block immediately (and decrement the reference counts of any blocks that the reclaimed block references), or place it on a queue for deferred collection.

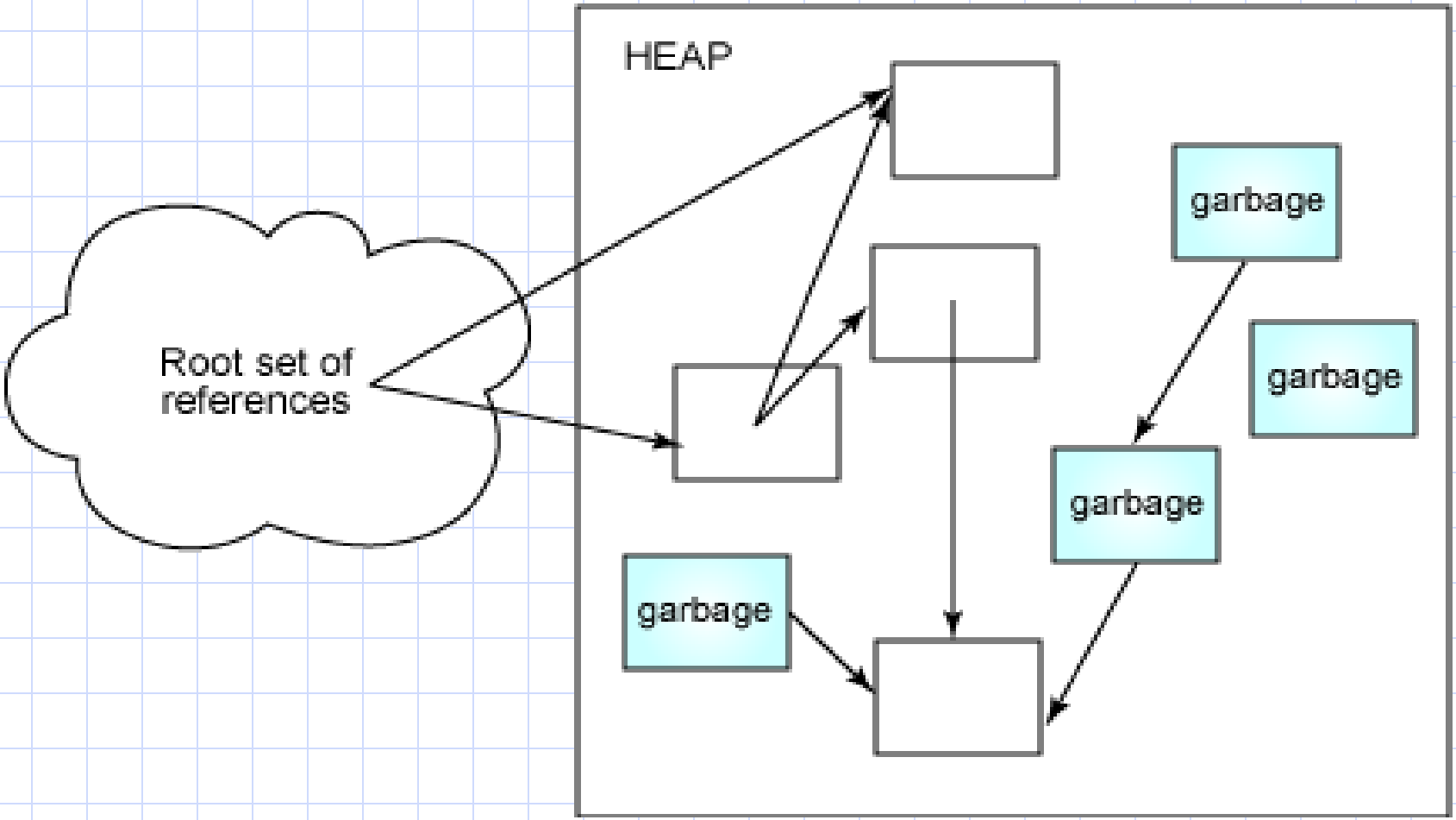
History of Garbage Collection

- Many ANSI C++ library classes, such as string, employ reference counting to provide the appearance of garbage collection. By overloading the assignment operator and exploiting the deterministic finalization provided by C++ scoping, C++ programs can use the string class as if it were garbage collected. Reference counting is simple, lends itself well to incremental collection, and the collection process tends to have good locality of reference, but it is rarely used in production garbage collectors for a number of reasons, such as its inability to reclaim unreachable cyclic structures (objects that reference each other directly or indirectly, like a circularly linked list or a tree that contains back-pointers to the parent node).

History of Garbage Collection

- Tracing collectors
  - None of the standard garbage collectors in the JDK uses reference counting; instead, they all use some form of *tracing collector*. A tracing collector stops the world (although not necessarily for the entire duration of the collection) and starts tracing objects, starting at the root set and following references until all reachable objects have been examined. Roots can be found in program registers, in local (stack-based) variables in each thread's stack, and in static variables.
- Mark-sweep collectors
  - The most basic form of tracing collector, first proposed by Lisp inventor John McCarthy in 1960, is the *mark-sweep* collector, in which the world is stopped and the collector visits each live node, starting from the roots, and marks each node it visits. When there are no more references to follow, collection is complete, and then the heap is swept (that is, every object in the heap is examined), and any object not marked is reclaimed as garbage and returned to the free list. Figure 1 illustrates a heap prior to garbage collection; the shaded blocks are garbage because they are unreachable by the user program:

History of Garbage Collection



History of Garbage Collection

- Mark-sweep is simple to implement, can reclaim cyclic structures easily, and doesn't place any burden on the compiler or mutator like reference counting does. But it has deficiencies -- collection pauses can be long, and the entire heap is visited in the sweep phase, which can have very negative performance consequences on virtual memory systems where the heap may be paged.
- The big problem with mark-sweep is that every active (that is, allocated) object, whether reachable or not, is visited during the sweep phase. Because a significant percentage of objects are likely to be garbage, this means that the collector is spending considerable effort examining and handling garbage. Mark-sweep collectors also tend to leave the heap fragmented, which can cause locality issues and can also cause allocation failures even when sufficient free memory appears to be available.

History of Garbage Collection

- Copying collectors
  - In a *copying collector*, another form of tracing collector, the heap is divided into two equally sized semi-spaces, one of which contains active data and the other is unused. When the active space fills up, the world is stopped and live objects are copied from the active space into the inactive space. The roles of the spaces are then flipped, with the old inactive space becoming the new active space.
  - Copying collection has the advantage of only visiting live objects, which means garbage objects will not be examined, nor will they need to be paged into memory or brought into the cache.
  - The duration of collection cycles in a copying collector is driven by the number of live objects.
  - However, copying collectors have the added cost of copying the data from one space to another, adjusting all references to point to the new copy. In particular, long-lived objects will be copied back and forth on every collection.

History of Garbage Collection

- Heap compaction
  - Copying collectors have another benefit, which is that the set of live objects are compacted into the bottom of the heap. This not only improves locality of reference of the user program and eliminates heap fragmentation, but also greatly reduces the cost of object allocation -- object allocation becomes a simple pointer addition on the top-of-heap pointer.
  - **There is no need to maintain free lists or look-aside lists, or perform best-fit or first-fit algorithms -- allocating N bytes is as simple as adding N to the top-of-heap pointer and returning its previous value, as suggested**

History of Garbage Collection

Inexpensive memory allocation in a copying collector

```
void *malloc(int n) {
        if (heapTop - heapStart < n) doGarbageCollection();
        void *wasStart = heapStart;
        heapStart += n;
        return wasStart;
}
```

History of Garbage Collection

- **This may be one of the reasons for the pervasive belief that object allocation is expensive -- earlier JVM implementations did not use copying collectors, and developers are still implicitly assuming allocation cost is similar to other languages, like C, when in fact it may be significantly cheaper in the Java runtime.**
- **Not only is the cost of allocation smaller, but for objects that become garbage before the next collection cycle, the deallocation cost is zero, as the garbage object will be neither visited nor copied.**

History of Garbage Collection

- Mark-compact collectors
  - The copying algorithm has excellent performance characteristics, but it has the drawback of requiring twice as much memory as a mark-sweep collector.
  - The *mark-compact* algorithm combines mark-sweep and copying in a way that avoids this problem, at the cost of some increased collection complexity. Like mark-sweep, mark-compact is a two-phase process, where each live object is visited and marked in the marking phase.
  - Then, marked objects are copied such that all the live objects are compacted at the bottom of the heap. If a complete compaction is performed at every collection, the resulting heap is similar to the result of a copying collector -- there is a clear demarcation between the active portion of the heap and the free area, so that allocation costs are comparable to a copying collector.
  - Long-lived objects tend to accumulate at the bottom of the heap, so they are not copied repeatedly as they are in a copying collector.

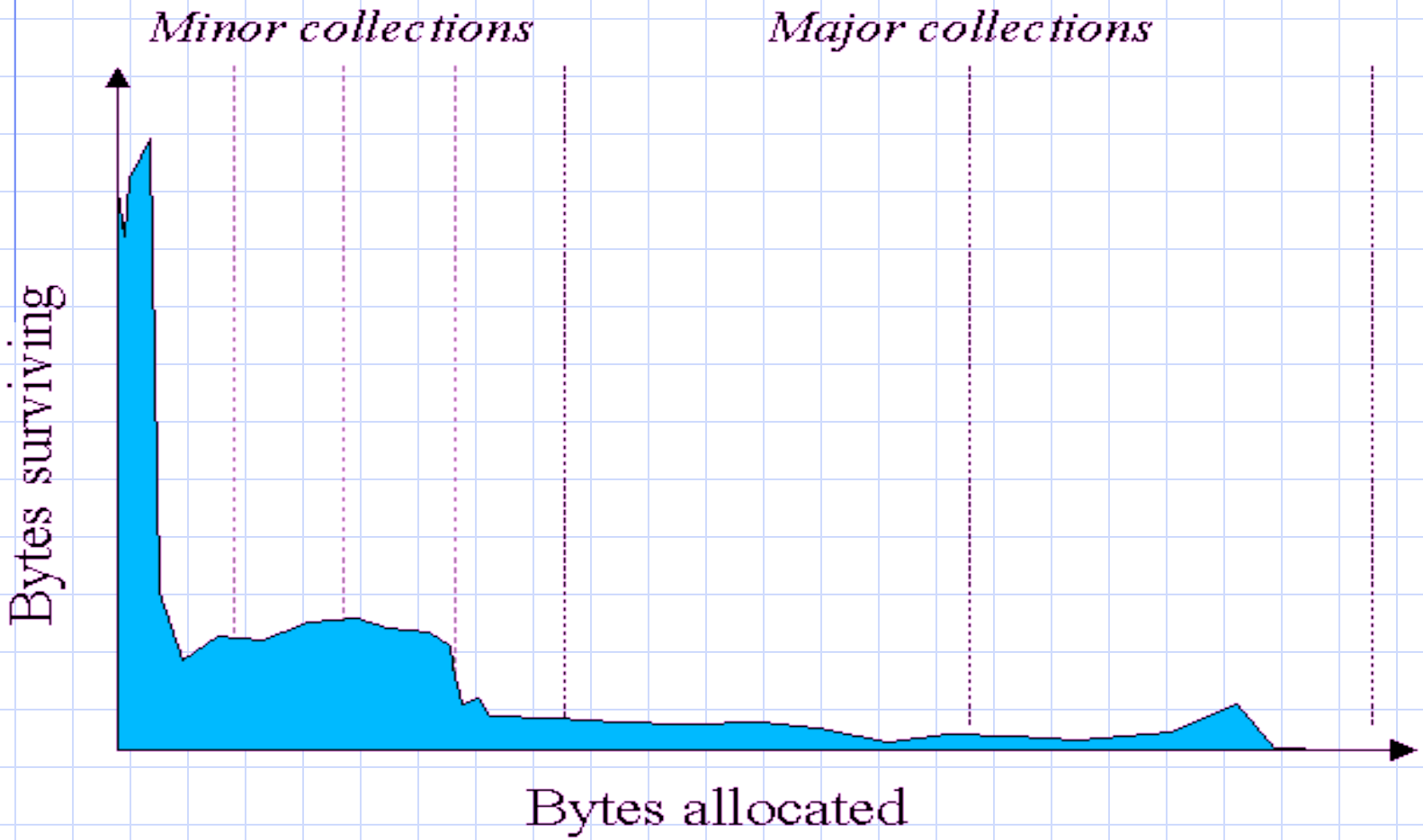
History of Garbage Collection

- Well, which one?
  - OK, so which of these approaches does the JDK take for garbage collection? In some sense, all of them. Early JDKs used a single-threaded mark-sweep or mark-sweep-compact collector. JDKs 1.2 and later employ a hybrid approach, called *generational collection*, where the heap is divided into several sections based on an object's age, and different generations are collected separately using different collection algorithms.
  - Generational garbage collection turns out to be very effective, although it introduces several additional bookkeeping requirements at runtime.

Generational Collectors

- Old objects, young objects
  - In any application heap, some objects become garbage shortly after their creation, some survive for a long time and then become garbage, and others can remain live for the entirety of the program's run.
  - Empirical studies have shown that for most object-oriented languages, the Java language included, the vast majority of objects -- as much as 98 percent, depending on your metric for object youth -- die young. One can measure an object's age in wall-clock seconds, in total bytes allocated by the memory management subsystem since the object was allocated, or the number of garbage collections since the object was allocated.

Generational Collectors



Generational Collectors

- But no matter how you measure, the studies show the same thing -- most objects die young. The fact that most objects die young has significance for the choice of collector.
- **In particular, copying collectors perform quite well when the majority of objects die young, since copying collectors do not visit dead objects at all; they simply copy the live objects to another heap region, then reclaim the whole of the remaining space in one fell swoop.**
- Of the objects that survive past their first collection, a significant portion of those will become long-lived or permanent. The various garbage collection strategies perform very differently depending on the mix of short-lived and long-lived objects.

Generational Collectors

- Copying collectors work very well when most objects die young, because objects that die young never need to be copied at all. However, the copying collector deals poorly with long-lived objects, repeatedly copying them back and forth from one semi-space to another.
- **Conversely, mark-compact collectors do very well with long-lived objects, because long-lived objects tend to accumulate at the bottom of the heap and then do not need to be copied again. Mark-sweep and mark-compact collectors, however, expend considerably more effort examining dead objects, because they must examine every object in the heap during the sweep phase.**

Generational Collectors

- Generational collection
  - A generational collector divides the heap into multiple generations. Objects are created in the young generation, and objects that meet some promotion criteria, such as having survived a certain number of collections, are then promoted to the next older generation. A generational collector is free to use a different collection strategy for different generations and perform garbage collection on the generations separately.
- Minor collections
  - One of the advantages of generational collection is that it can make garbage collection pauses shorter by not collecting all generations at once. When the allocator is unable to fulfill an allocation request, it first triggers a *minor collection*, which only collects the youngest generation.

Generational Collectors

- Since many of the objects in the young generation will already be dead and the copying collector does not need to examine dead objects at all, minor collection pauses can be quite short and can often reclaim significant heap space.
- If the minor collection frees enough heap space, the user program can resume immediately. If it does not free enough heap space, it proceeds to collect higher generations until enough memory has been reclaimed. (In the event the garbage collector cannot reclaim enough memory after a full collection, it will either expand the heap or it will throw an OutOfMemoryError.)

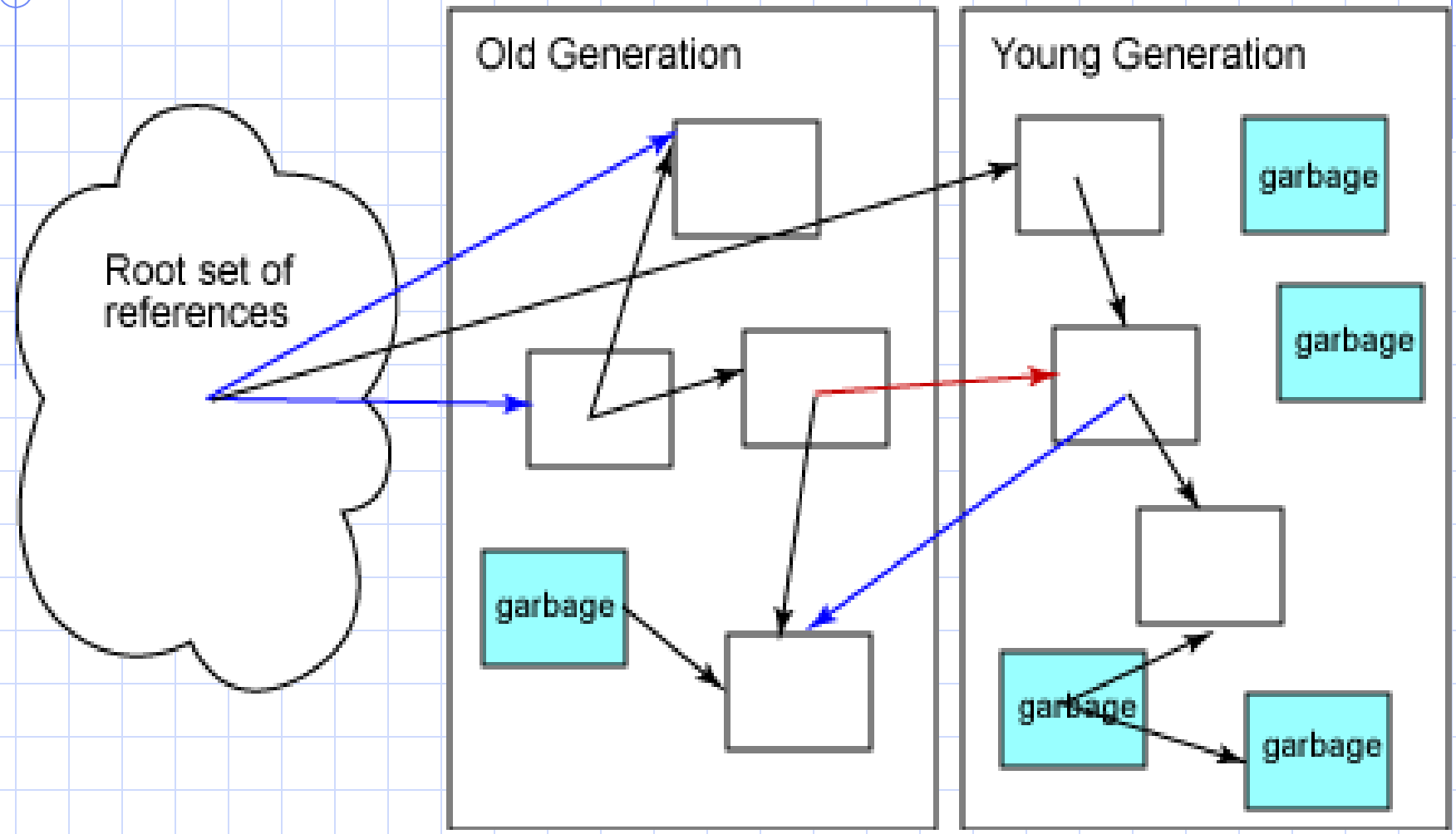
Generational Collectors

- Intergenerational references
  - Tracing garbage collectors, such as copying, mark-sweep, and mark-compact, all start scanning from the root set, traversing references between objects, until all live objects have been visited.
  - A generational tracing collector starts from the root set, but does not traverse references that lead to objects in the older generation, which reduces the size of the object graph to be traced. But this creates a problem -- what if an object in the older generation references a younger object, which is not reachable through any other chain of references from a root?

Generational Collectors

- To address this problem, generational collectors must explicitly track references from older objects to younger objects and add these old-to-young references into the root set of the minor collection.
- **There are two ways to create a reference from an old object to a young object. Either one of the references contained in an old object is modified to refer to a young object, or a young object that refers to other young objects is promoted into the older generation.**

Generational Collectors



Generational Collectors

- Tracking intergenerational references
  - Whether an old-to-young reference is created by promotion or a pointer modification, the garbage collector needs to have a comprehensive set of old-to-young references when it wants to perform a minor collection.
  - One way to do this would be to trace the old generation, but this clearly has significant overhead. Somewhat better would be to linearly scan the old generation looking for references to young objects. This approach is faster than tracing and has better locality, but is still considerable work.
  - The mutator and garbage collector can work together to maintain a comprehensive list of old-to-young references as they are created. When objects are promoted into the older generation, the garbage collector can note any old-to-young references that are created as a result of the promotion, which leaves only the tracking of intergenerational references created by pointer modifications.

Generational Collectors

- The garbage collector can track old-to-young references that arise through modifying references held within existing objects in several ways.
- **It could track them in the same manner as maintaining reference counts in reference-counting collectors (the compiler could generate additional instructions surrounding pointer assignments) or could use virtual memory protection on the old generation heap to trap writes to older objects. Another potentially more efficient virtual memory approach would be to use the page modification dirty bits in the old generation heap to identify blocks to scan for objects containing old-to-young pointers.**
- With a little cleverness, it is possible to avoid the overhead of tracking every pointer modification and inspecting it to see if it crosses generational boundaries.

Generational Collectors

- For example, it is not necessary to track stores to local or static variables, because they will already be part of the root set. It may also be possible to avoid tracking pointer stores within constructors that simply initialize fields of newly created objects (so-called *initializing stores*), as (almost) all objects are allocated in the young generation.
- In any case, the runtime must maintain an explicit set of references from old objects to young ones and add these references to the root set when collecting the young generation.

Generational Collectors

- Card marking
  - The Sun JDKs use an optimized variant of an algorithm called *card marking* to identify modifications to pointers held in fields of old-generation objects. In this approach, the heap is divided into a set of *cards*, each of which is usually smaller than a memory page.
  - The JVM maintains a card map, with one bit (or byte, in some implementations) corresponding to each card in the heap. Each time a pointer field in an object in the heap is modified, the corresponding bit in the card map for that card is set.
  - At garbage collection time, the mark bits associated with cards in the old generation are examined, and dirty cards are scanned for objects containing references into the younger generation. Then the mark bits are cleared.

Generational Collectors

- There are several costs to card marking – additional space for the card map, additional work to be done on each pointer store, and additional work to be done at garbage collection time. Card marking algorithms can add as little as two or three machine instructions per non-initializing heap pointer store, and entails scanning any objects on dirty cards at minor collection time.

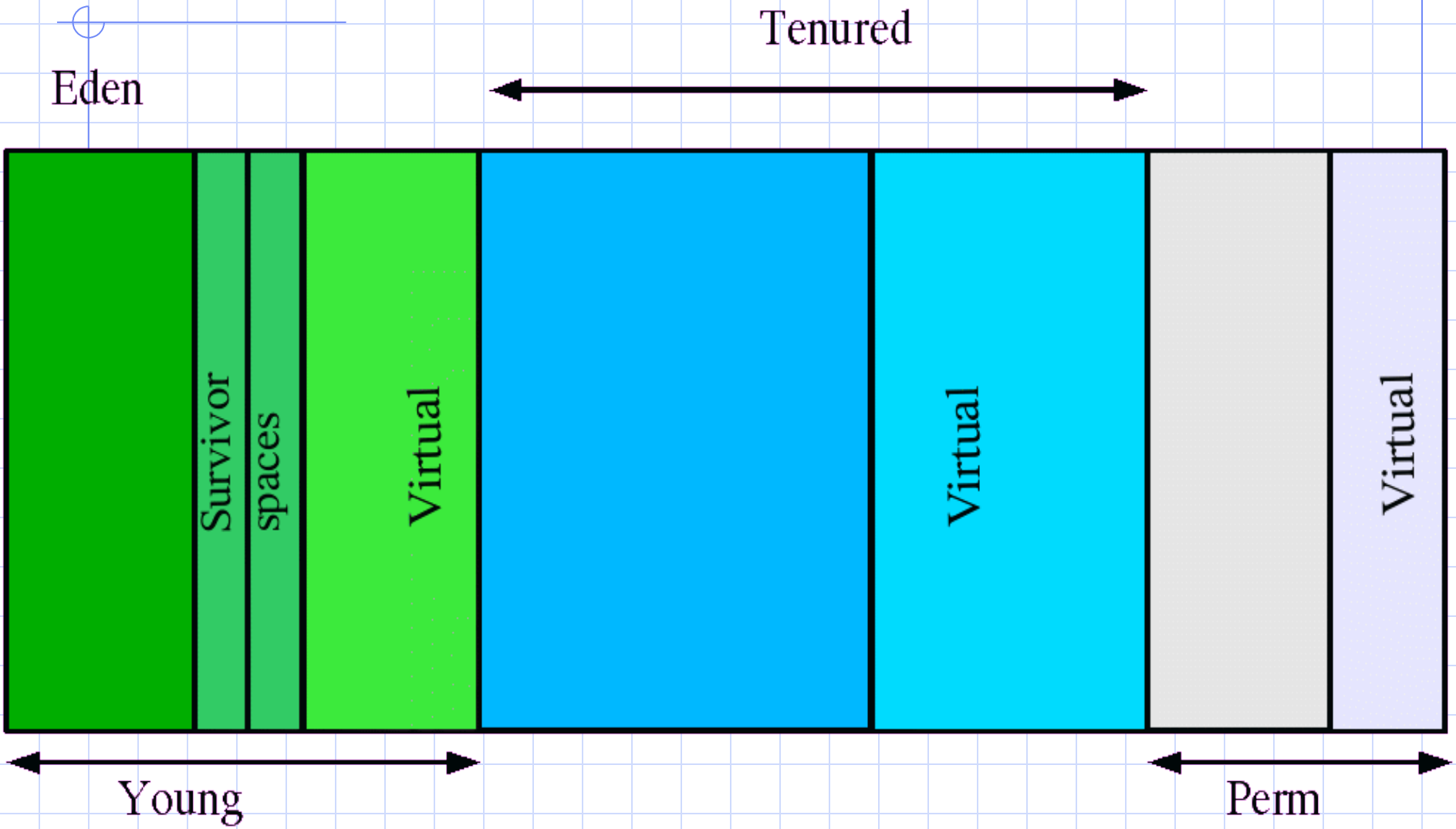
Generational Collectors

- The JDK 1.4.1 default collector
  - By default, the 1.4.1 JDK divides the heap into two sections, a young generation and an old generation. (Actually, there is also a third section, the permanent space, which is used for storing loaded class and method objects.) The young generation is divided into a creation space, often called *Eden*, and two survivor semi-spaces, using a copying collector.
  - The old generation uses a mark-compact collector. Objects are promoted from the young generation to the old generation after they have survived copying a certain number of times.
  - A minor collection will copy live objects from Eden and one of the survivor semi-spaces into the other survivor space, potentially promoting some objects to the older generation. A major collection will collect both the young and old generation.

Generational Collectors

- The System.gc() method always triggers a major collection, which is one of the reasons you should use System.gc() sparingly, if at all, because major collections can take much longer than a minor collection. There is no way to programmatically trigger a minor collection.

Generational Collectors



Generational Collectors

- Other collection options
  - In addition to the copying and mark-compact collectors used by default, the 1.4.1 JDK also contains four other garbage collection algorithms, each of which is suited to a different purpose.
  - JDK 1.4.1 includes an incremental collector (which has been around since JDK 1.2) and three new collectors for more efficient collection on multiprocessor systems -- the parallel copying collector, the parallel scavenging collector, and the concurrent mark-sweep collector.
  - These new collectors address the problem of the garbage collector being a scalability bottleneck on multiprocessor systems.

Generational Collectors

	Low Pause Collectors		Throughput Collectors		Heap Sizes
Generation		2+ CPUs	1 CPU	2+ CPUs	

Young	Copying Collector *(default)*	Parallel Copying Collector
		`-XX:+UseParNewGC`
			Copying Collector *(default)*	Parallel Scavenge Collector
				`-XX:+UseParallelGC`
				`-XX:+UseAdaptiveSizePolicy`
				`-XX:+AgressiveHeap`
					`-XX:NewSize`
					`-XX:MaxNewSize`
					`-XX-SurvivorRatio`

Old	Mark-Compact Collector *(default)*	Concurrent Collector `-XX:UseConcMarkSweepGC`	Mark-Compact Collector *(default)*	Mark-Compact Collector *(default)*	`-Xms, -Xmx`
Permanent	Can be turned off with `-Xnoclassgc` Use with care!				`-XX:PermSize` `-XX:MaxPermSize`

Garbage collection and performance

- **How expensive is allocation?**
  - The 1.0 and 1.1 JDKs used a mark-sweep collector, which did compaction on some -- but not all -- collections, meaning that the heap might be fragmented after a garbage collection. Accordingly, memory allocation costs in the 1.0 and 1.1 JVMs were comparable to that in C or C++, where the allocator uses heuristics such as "first-first" or "best-fit" to manage the free heap space.
  - Deallocation costs were also high, since the mark-sweep collector had to sweep the entire heap at every collection. No wonder we were advised to go easy on the allocator.

Garbage collection and performance

- In HotSpot JVMs (Sun JDK 1.2 and later), things got a *lot* better -- the Sun JDKs moved to a generational collector. Because a copying collector is used for the young generation, the free space in the heap is always contiguous so that allocation of a new object from the heap can be done through a simple pointer addition, as shown below.
- This makes object allocation in Java applications significantly cheaper than it is in C, a possibility that many developers at first have difficulty imagining.
- Similarly, because copying collectors do not visit dead objects, a heap with a large number of temporary objects, which is a common situation in Java applications, costs very little to collect; simply trace and copy the live objects to a survivor space and reclaim the entire heap in one fell swoop. No free lists, no block coalescing, no compacting -- just wipe the heap clean and start over. So both allocation and deallocation costs per object went way down in JDK 1.2.

Garbage collection and performance

Inexpensive memory allocation in a copying collector

```
void *malloc(int n) {
        if (heapTop - heapStart < n) doGarbageCollection();
        void *wasStart = heapStart;
        heapStart += n;
        return wasStart;
}
```

Garbage collection and performance

- **Performance advice often has a short shelf life;**
  - while it was once true that allocation was expensive, it is now no longer the case. In fact, it is downright cheap, and with a few very compute-intensive exceptions, performance considerations are generally no longer a good reason to avoid allocation.
    - Sun estimates allocation costs at approximately *ten machine instructions*. That's pretty much free -- certainly no reason to complicate the structure of your program or incur additional maintenance risks for the sake of eliminating a few object creations.
  - Of course, allocation is only half the story -- most objects that are allocated are eventually garbage collected, which also has costs. But there's good news there, too.
  - The vast majority of objects in most Java applications become garbage before the next collection.

Garbage collection and performance

- The cost of a minor garbage collection is proportional to the number of live objects in the young generation, not the number of objects allocated since the last collection.
- Because so few young generation objects survive to the next collection, the amortized cost of collection per allocation is fairly small (and can be made even smaller by simply increasing the heap size, subject to the availability of enough memory).

Garbage collection and performance

- Wait it gets better.
  - Consider the code below.

```
//Escape analysis can eliminate many temporary allocations entirely

void doSomething() {
    Point p = someObject.getPosition();
    System.out.println("Object is at (" + p.x, + ", " + p.y + ")");
}

... Point getPosition() {
    return new Point(myX, myY);
}
```

Garbage collection and performance

- The JIT compiler can perform additional optimizations that can reduce the cost of object allocation to zero. In the above listing, where the getPosition() method creates a temporary object to hold the coordinates of a point, and the calling method uses the Point object briefly and then discards it.
- The JIT will likely inline the call to getPosition() and, using a technique called *escape analysis*, can recognize that no reference to the Point object leaves the doSomething() method.
- Knowing this, the JIT can then allocate the object on the stack instead of the heap or, even better, optimize the allocation away completely and simply hoist the fields of the Point into registers.

Garbage collection and performance

- While the current Sun JVMs do not yet perform this optimization, future JVMs(5.0 onwards) probably will. The fact that allocation can get even cheaper in the future, with no changes to your code, is just one more reason not to compromise the correctness or maintainability of your program for the sake of avoiding a few extra allocations.

Garbage collection and performance

- **Isn't the allocator a scalability bottleneck?**
  - Above shows that while allocation itself is fast, access to the heap structure must be synchronized across threads. So doesn't that make the allocator a scalability hazard? There are several clever tricks JVMs use to reduce this cost significantly.
  - IBM JVMs use a technique called *thread-local heaps*, by which each thread requests a small block of memory (on the order of 1K) from the allocator, and small object allocations are satisfied out of that block.
  - If the program requests a larger block than can be satisfied using the small thread-local heap, then the global allocator is used to either satisfy the request directly or to allocate a new thread-local heap. By this technique, a large percentage of allocations can be satisfied without contending for the shared heap lock. (Sun JVMs use a similar technique, instead using the term "Local Allocation Blocks.")

Garbage collection and performance

- **Finalizers are not your friend**
  - Objects with finalizers (those that have a non-trivial finalize() method) have significant overhead compared to objects without finalizers, and should be used sparingly. Finalizeable objects are both slower to allocate and slower to collect.
  - At allocation time, the JVM must register any finalizeable objects with the garbage collector, and (at least in the HotSpot JVM implementation) finalizeable objects must follow a slower allocation path than most other objects.
  - Similarly, finalizeable objects are slower to collect, too. It takes at least two garbage collection cycles (in the best case) before a finalizeable object can be reclaimed, and the garbage collector has to do extra work to invoke the finalizer.

Garbage collection and performance

- The result is more time spent allocating and collecting objects and more pressure on the garbage collector, because the memory used by unreachable finalizeable objects is retained longer. Combine that with the fact that finalizers are not guaranteed to run in any predictable timeframe, or even at all, and you can see that there are relatively few situations for which finalization is the right tool to use.

Garbage collection and performance

- Helping the garbage collector . . . not
  - Because allocation and garbage collection at one time imposed significant performance costs on Java programs, many clever tricks were developed to reduce these costs, such as object pooling and nulling. Unfortunately, in many cases these techniques can do more harm than good to your program's performance.
  - Object pooling
    - The theory is that pooling spreads out the allocation costs over many more uses. When the object creation cost is high, such as with database connections or threads, or the pooled object represents a limited and costly resource, such as with database connections, this makes sense. However, the number of situations where these conditions apply is fairly small.

Garbage collection and performance

- In addition, object pooling has some serious downsides. Because the object pool is generally shared across all threads, allocation from the object pool can be a synchronization bottleneck. Pooling also forces you to manage deallocation explicitly, which reintroduces the risks of dangling pointers.
- Also, the pool size must be properly tuned to get the desired performance result. If it is too small, it will not prevent allocation; and if it is too large, resources that could get reclaimed will instead sit idle in the pool.
- By tying up memory that could be reclaimed, the use of object pools places additional pressure on the garbage collector. Writing an effective pool implementation is not simple.

Garbage collection and performance

- **Lastly pooling objects create situation in which gc has to track old to new references.See the example..**

Garbage collection and performance

- **Explicit nulling**
  - Explicit nulling is simply the practice of setting reference objects to null when you are finished with them. The idea behind nulling is that it assists the garbage collector by making objects unreachable earlier. Or at least that's the theory.
  - There is *one* case where the use of explicit nulling is not only helpful, but virtually required, and that is where a reference to an object is scoped more broadly than it is used or considered valid by the program's specification.

Garbage collection and performance

```
public class LinkedList {
    private static class ListElement {
        private ListElement nextElement;
        private Object value;
    }
    private ListElement head;
    ...
    public void finalize() {
        try {   ListElement p = head;
                while (p != null) {
                    p.value = null;
                    ListElement q = p.nextElement;
                    p.nextElement = null; p = q;
                }
                head = null;
        } finally {  super.finalize(); }
    }
}
```

Garbage collection and performance

- Consider the above code, which combines several really bad ideas. The listing is a linked list implementation that uses a finalizer to walk the list and null out all the forward links. We've already discussed why finalizers are bad.
- This case is even worse because now the class is doing extra work, ostensibly to help the garbage collector, but that will not actually help -- and might even hurt.
- Walking the list takes CPU cycles and will have the effect of visiting all those dead objects and pulling them into the cache -- work that the garbage collector might be able to avoid entirely, because copying collectors do not visit dead objects at all. Nulling the references doesn't help a tracing garbage collector anyway; if the head of the list is unreachable, the rest of the list won't be traced anyway.
- **Lastly the list may have been paged out. (It couldn’t have been worse)**

Garbage collection and performance

- **Explicit garbage collection**
  - A third category where developers often mistakenly think they are helping the garbage collector is the use of System.gc(), which triggers a garbage collection (actually, it merely suggests that this might be a good time for a garbage collection).
  - Unfortunately, System.gc() triggers a full collection, which includes tracing all live objects in the heap and sweeping and compacting the old generation. This can be a lot of work. In general, it is better to let the system decide when it needs to collect the heap, and whether or not to do a full collection.
  - If you are concerned that your application might have hidden calls to System.gc() buried in libraries, you can invoke the JVM with the -XX:+DisableExplicitGC option to prevent calls to System.gc() and triggering a garbage collection.

Garbage collection and performance

- **Immutability, again**
  - Making objects immutable eliminates entire classes of programming errors. One of the most common reasons given for not making a class immutable is the belief that doing so would compromise performance.
  - While this is true sometimes, it is often not -- and sometimes the use of immutable objects has significant, and perhaps surprising, performance advantages.
  - In most cases, when a holder object is updated to reference a different object, the new referent is a young object. If we update a MutableHolder by calling setValue(), we have created a situation where an older object references a younger one. On the other hand, by creating a new ImmutableHolder object instead, a younger object is referencing an older one. The latter situation, where most objects point to older objects, is much more gentle on a generational garbage collector.

Garbage collection and performance

- **When good performance advice goes bad**
  - A cover story in the July 2003 *Java Developer's Journal* illustrates how easy it is for good performance advice to become bad performance advice by simply failing to adequately identify the conditions under which the advice should be applied or the problem it was intended to solve.
  - While the article contains some useful analysis, it will likely do more harm than good (and, unfortunately, far too much performance-oriented advice falls into this same trap).

Garbage collection and performance

- The article opens with a set of requirements from a realtime environment, where unpredictable garbage collection pauses are unacceptable and there are strict operational requirements on how long a pause can be tolerated.
- The authors then recommend nulling references, object pooling, and scheduling explicit garbage collection to meet the performance goals. So far, so good -- they had a problem and they figured out what they had to do to solve it (although they appear to have failed to identify what the costs of these practices were or explore some less intrusive alternatives, such as concurrent collection).
- Unfortunately, the article's title ("Avoid Bothersome Garbage Collection Pauses") and presentation suggest that this advice would be useful for a wide range of applications -- perhaps *all* Java applications. This is terrible, dangerous performance advice!

Garbage collection and performance

- For most applications, explicit nulling, object pooling, and explicit garbage collection will harm the throughput of your application, not improve it -- not to mention the intrusiveness of these techniques on your program design.
- In certain situations, it may be acceptable to trade throughput for predictability -- such as real-time or embedded applications. But for many Java applications, including most server-side applications, you probably would rather have the throughput.

Garbage collection and performance

- **The moral of the story is that performance advice is highly situational (and has a short shelf life). Performance advice is by definition reactive -- it is designed to address a particular problem that occurred in a particular set of circumstances.**
- If the underlying circumstances change, or they are simply not applicable to your situation, the advice may not be applicable, either. Before you muck up your program's design to improve its performance, first make sure you have a performance problem and that following the advice will solve that problem.

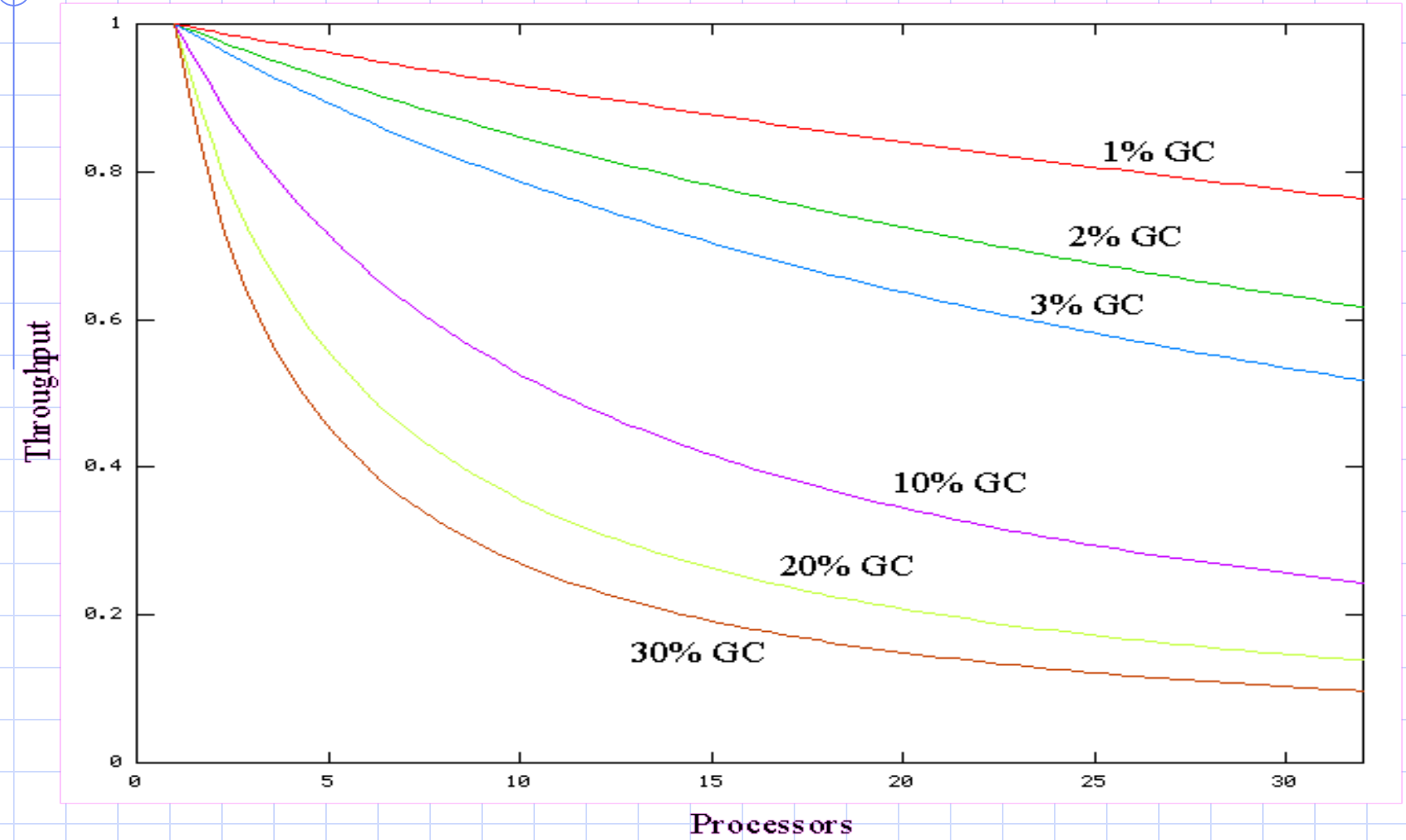
Tuning Garbage Collection

- As Java technology becomes more and more pervasive in the enterprise and telecommunications (telco) industry, understanding the behavior of the garbage collector becomes more important.
  - Typically, telco applications are near-real-time applications. Delays measured in milliseconds are not usually a problem, but delays of hundreds of milliseconds, let alone seconds, can spell trouble for applications of this kind -- applications compromise on throughput to provide near-real-time performance.
  - Enterprise applications are transaction oriented and tolerate delays better. They need to crunch as many transactions as possible in the shortest time i.e., the more compute time and resources available, the better. This means, faster and multiple CPUs, lots of memory, increases performance. A garbage collector that can make use of the extra resources, enhances performance.

Tuning Garbage Collection

- Garbage collection limitations, which affected the performance of telco and enterprise applications is in the process of being eliminated with the introduction of new parallel and concurrent collectors.
  - The collectors can be used to reduce delays to milliseconds for telco applications -- SIP servers, Call Processing applications --, while increasing throughput by providing more compute time to enterprise applications -- J2EE, OSS/BSS, MOM type of applications.

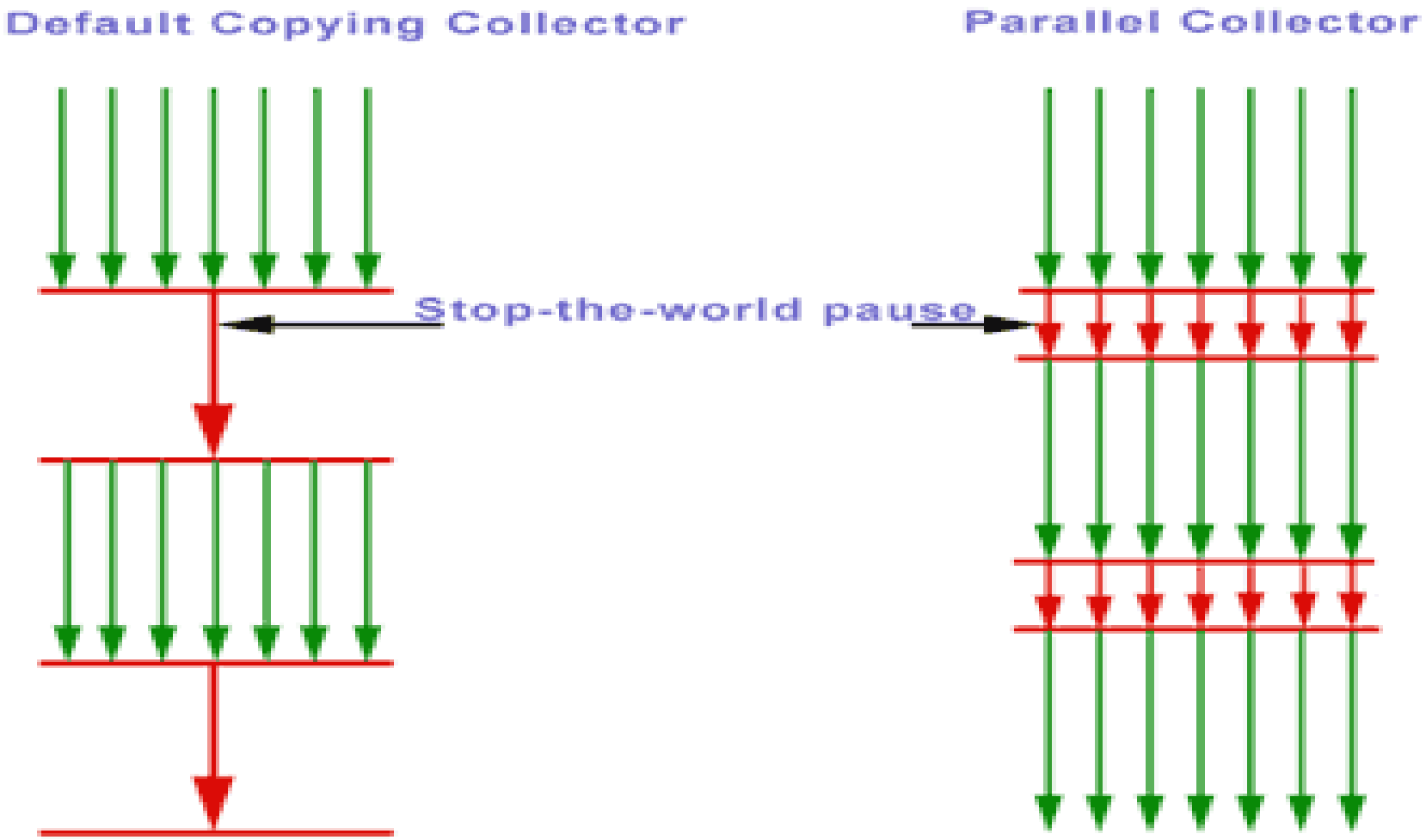
Tuning Garbage Collection



Tuning Garbage Collection

- **Parallel Copying Collector**
  - The Parallel Copying Collector is similar to the Copying Collector , but instead of using one thread to collect young generation garbage, the collector allocates as many threads as the number of CPUs to parallelize the collection. The parallel copying collector works with both the concurrent collector and the default mark-compact collector.
  - The parallel copying collection is still stop-the-world, but the cost of the collection is now dependent on the live data in the young generation heap, divided by the number of CPUs available.
  - So bigger younger generations can be used to eliminate temporary objects while still keeping the pause low. The degree of parallelism i.e., the number of threads collecting can be tuned. This parallel collector works very well from small to big young generations.

Tuning Garbage Collection



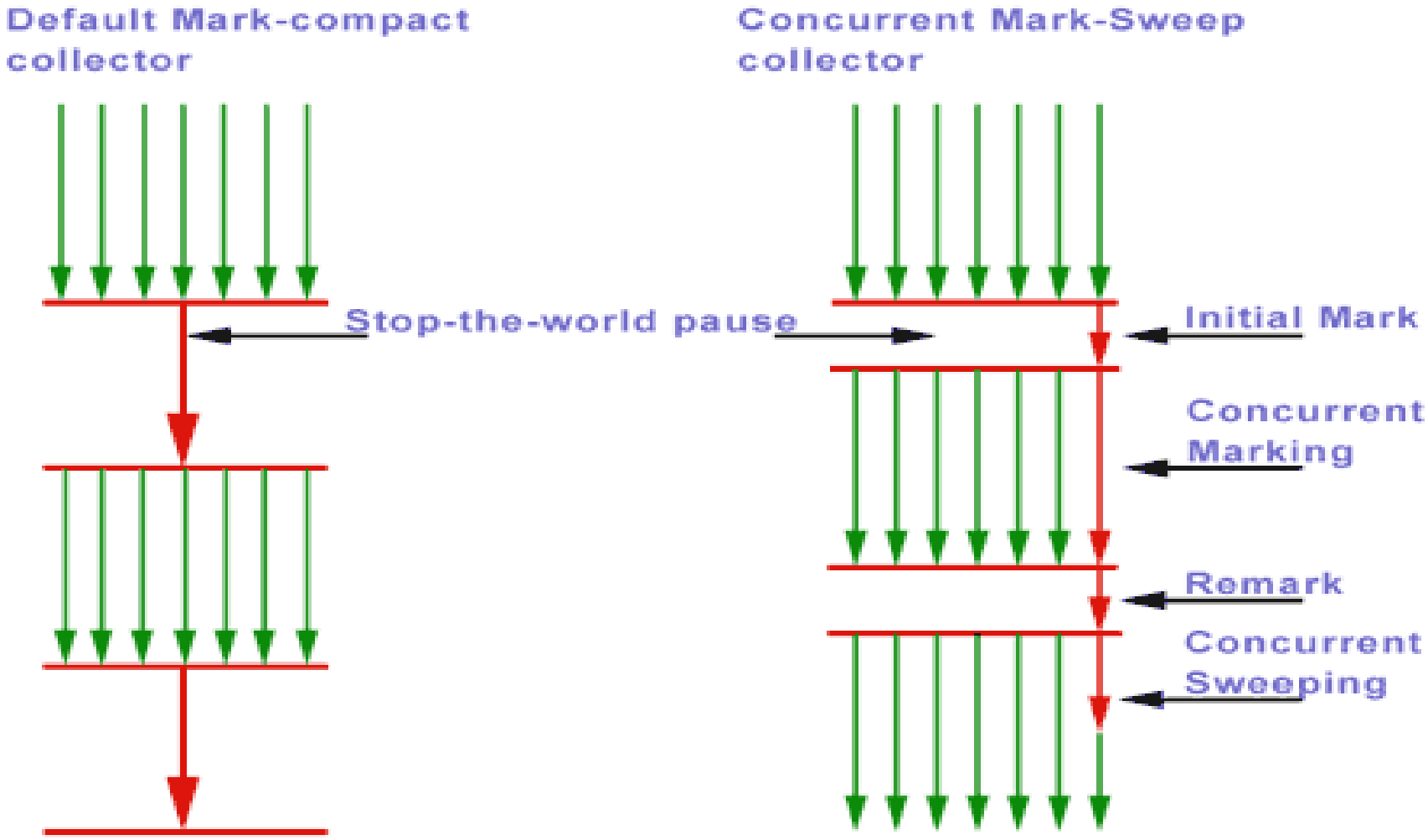
Tuning Garbage Collection

- **Concurrent Collector**
  - The concurrent collector uses a background thread that runs concurrently with the application threads to enable both garbage collection and object allocation/modification to happen at the same time.
  - The collector collects the garbage in phases, two are stop-the-world phases, and four are concurrent and run along with the application threads. The phases in order are, initial-mark phase (stop-the-world), mark-phase (concurrent), pre-cleaning phase (concurrent), remark-phase (stop-the-world), sweep-phase (concurrent) and reset-phase (concurrent).
  - The initial-mark phase takes a snapshot of the old generation heap objects followed by the marking and pre-cleaning of live objects. Once marking is complete, a remark-phase takes a second snapshot of the heap objects to capture changes in live objects.

Tuning Garbage Collection

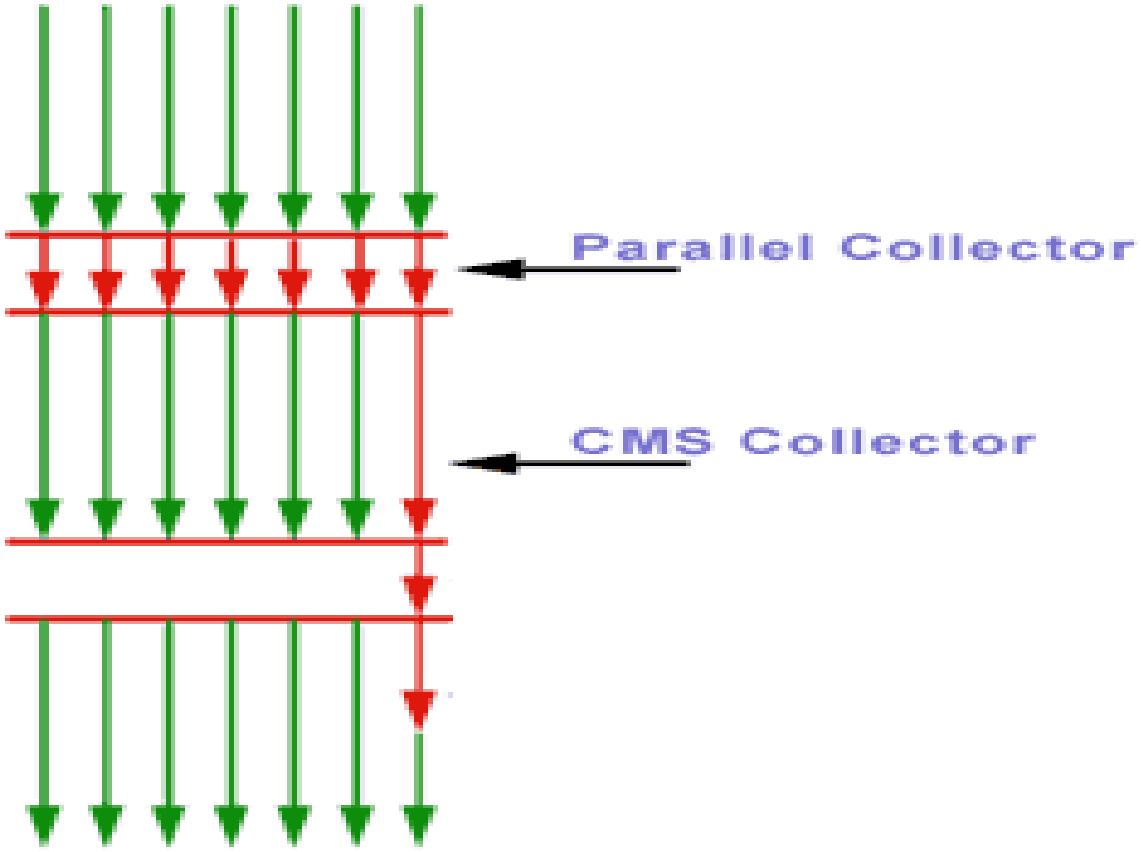
- This is followed by a sweep phase to collect dead objects - coalescing of dead objects space may also happen here. The reset phase clears the collector data structures for the next collection cycle. The collector does most of its work concurrently, suspending application execution only briefly.

Tuning Garbage Collection



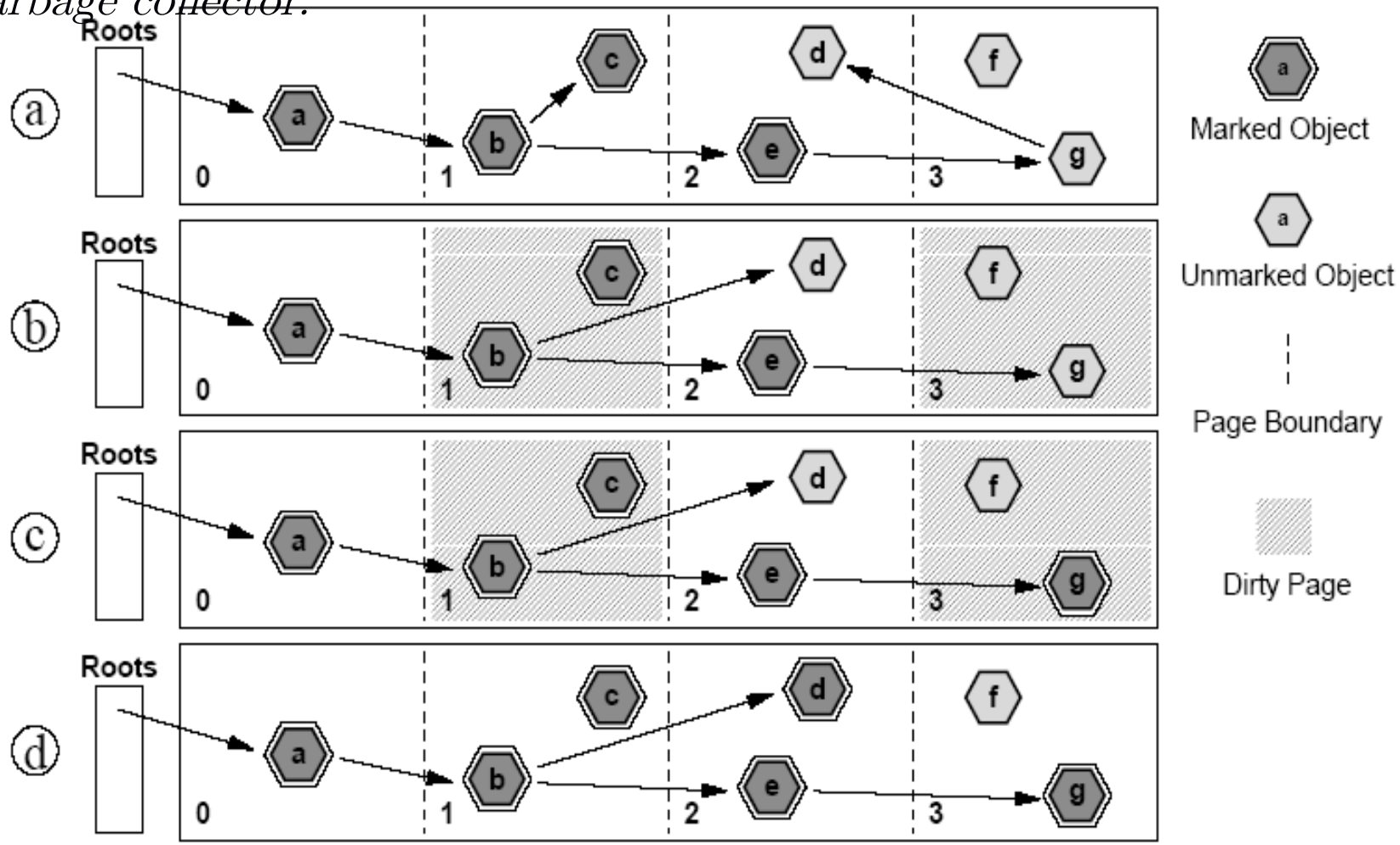
Tuning Garbage Collection

Old generation collectors. Garbage collection with the Parallel Collector and Concurrent mark-sweep collector enabled together. Best Case scenario for high throughput and low latency.



Tuning Garbage Collection

Concrete example of the operation of the original mostlyconcurrent garbage collector.



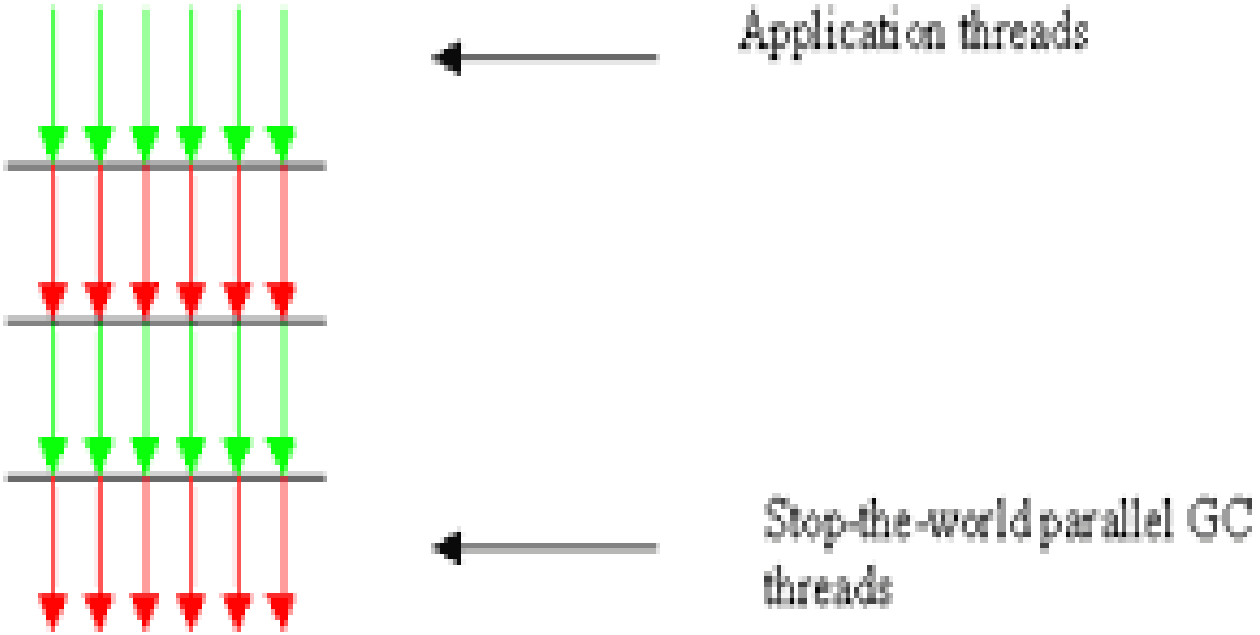
Tuning Garbage Collection

- **Throughput Collectors**
  - **Parallel Scavenge Collector**
    - The parallel scavenge collector is similar to the parallel copying collector, and collects young generation garbage. The collector is targeted towards large young generation heaps and to scale with more CPUs.
    - It works very well with large young generation heap sizes that are in gigabytes, like 12GB to 300GB or more, and scales very well with increase in CPUs, 8 CPUs or more. It is designed to maximize throughput in enterprise environments where plenty of memory and processing power is available.

Tuning Garbage Collection

- The parallel scavenge collector is again stop-the-world, and is designed to keep the pause down. The degree of parallelism can again be controlled. In addition, the collector has an adaptive tuning policy that can be turned on to optimize the collection.
- It balances the heap layout by resizing, Eden, Survivor spaces and old generation sizes to minimize the time spent in the collection. Since the heap layout is different for this collector, with large young generations, and smaller older generations, a new feature called "promotion undo" prevents old generation out-of-memory exceptions by allowing the parallel collector to finish the young generation collection.

Tuning Garbage Collection

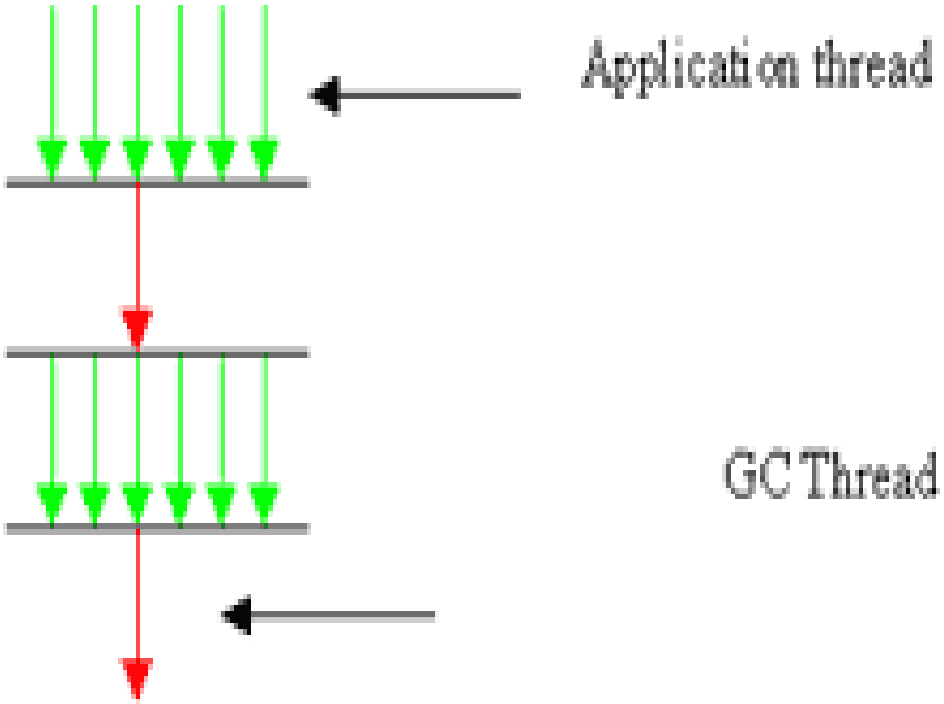


Parallel (multi-threaded) Stop-The-World Young Generation Collection

Tuning Garbage Collection

- **Mark-Compact Collector**
  - The parallel scavenge collector interacts with the mark-sweep-compact collector, the default old generation collector. The mark-compact collector is the traditional mark-compact collector, and is very efficient for enterprise environments where pause is not a big criterion.
  - The throughput collectors are designed to maximize the younger generation heap while keeping the older generation heap to the needed minimum - old generation is intended to very long-term objects only.

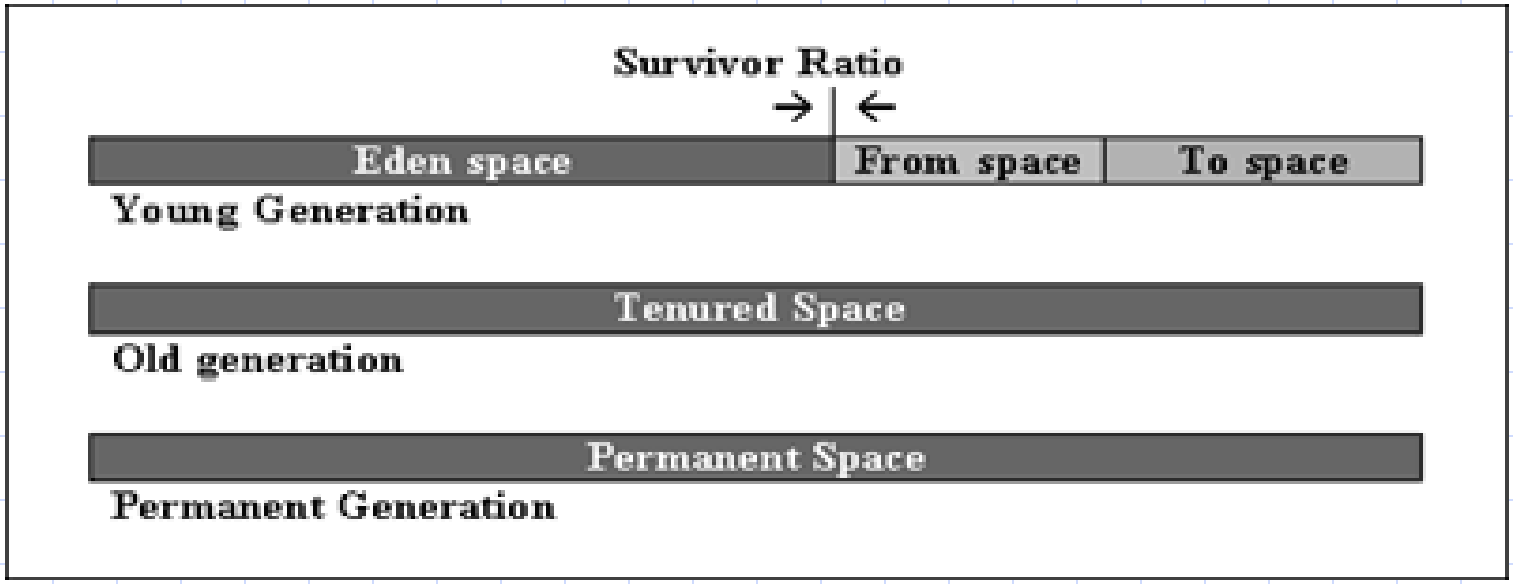
Tuning Garbage Collection



Single Threaded Stop-The-World Old Generation Mark-Compact Collection

Tuning Garbage Collection

- Heap Layout in JDK 1.4.1
  - **Young Generation Heap**
    - In JDK 1.4.1, the heap is divided into 3 generations, young generation, old generation, and permanent generation. Young generation is further divided into an Eden, and Semi-spaces.



Tuning Garbage Collection

- **Young Generation Heap** :
  - The size of the Eden and semi-spaces is controlled by the SurvivorRatio and can be calculated roughly as: .
  - The young generation can be sized using the following options:

    -XX:NewSize
    -XX:MaxNewSize
    -XX:SurvivorRatio
  - For example, to size a 128 MB young generation with an Eden of 64MB, a Semi-Space size of 32MB, the NewSize, MaxNewSize, and SurvivorRatio values can be specified as follows:

    java -Xms512m -Xmx512m **-XX:NewSize=128m -XX :MaxNewSize=128m** \
    -XX:SurvivorRatio=2 application

Tuning Garbage Collection

- **Old Generation Heap**
  - The old generation or the tenured generation is used to hold or age objects promoted from the younger generation. The maximum size of the older generation is controlled by the -Xms parameter.
  - For the previous example to size a 256 MB old generation heap with a young generation of 256 MB the -mx value can be specified as:
  - java **-Xms512m -Xmx512m** -XX:NewSize=256m -XX :MaxNewSize=256m \
    -XX:SurvivorRatio=2 application
  - The young generation takes 256 MB and the old generation 256 MB. -Xms is used to specify the initial size of the heap.

Tuning Garbage Collection

Permanent Generation Heap

- The permanent generation is used to store class objects and related meta data. The default space for this is 4 MB, and can be sized using the **-XX:PermSize**, and **-XX:MaxPermSize** option.
- Sometimes you will see Full GCs in the log file, and this could be due to the permanent generation being expanded. This could be prevented by sizing the permanent generation with a bigger heap using the -XX:PermSize and -XX:MaxPermSize options.
- For example:

  java -Xms512m -Xmx512m -XX:NewSize=256m -XX :MaxNewSize=256m \
  -XX:SurvivorRatio=2 **-XX:PermSize=64m -XX :MaxPermSize=64m** application

Tuning Garbage Collection

- Another way of disabling permanent generation collection is to use the -Xnoclassgc option. This is should be used with care since this disables class objects from being collected. To use this, size the permanent generation bigger so that there is enough space to store class objects, and a garbage collection is not needed to free up space.
- For example:

  java -Xms512m -Xmx512m -XX:NewSize=256m -XX :MaxNewSize=256m \
  -XX:SurvivorRatio=2 **-XX:PermSize=128m -XX :MaxPermSize=128m** \
  -Xnoclassgc application

Tuning Garbage Collection

- **Using the :XX Switches to Enable the New Low Pause or Throughput Collectors**
  - **Using the Low Pause Collectors**
  - The young generation, parallel copying collector can be enabled by using the **-XX:+UseParNewGC** option, while the older generation, concurrent collector can be enabled by using the **-XX :+ UseConcMarkSweepGC** option.
  - For example:

    java -server -Xms512m -Xmx512m -XX:NewSize=64m -XX :MaxNewSize=64m \
    -XX:SurvivorRatio=2 **-XX:+ UseConcMarkSweepGC** \
    -XX:+UseParNewGC application

Tuning Garbage Collection

- Note:
  - If -XX:+UseParNewGC is not specified, the young generation will make use of the default copying collector.
  - If -XX+UseParNewGC is specified on a single processor machine, the default copy collector is used since the number of CPUs is 1. You can force the parallel copy collector to be enabled by increasing the degree of parallelism.

Tuning Garbage Collection

- **Controlling the Degree of Parallelism**[2]
  - By default, the parallel copy collector will start as many threads as CPUs on the machine, but if the degree of parallelism needs to controlled, then it can be specified by the following option: **-XX :ParallelGCThreads=<desired parallelism>**
  - Default value is equal to number of CPUs.
  - For example, to use 4 parallel threads to process young generation collection:

    java -server -Xms512m -Xmx512m -XX:NewSize=64m -XX :MaxNewSize=64m \
    -XX:SurvivorRatio=2 -XX:+UseParNewGC **-XX :ParallelGCThreads=4** \
    -XX:+UseConcMarkSweepGC application

Tuning Garbage Collection

- **Simulating The "promoteall" Modifier In JDK 1.4.1**
  - "promoteall" is a modifier available in JDK 1.2.2 that enables promotion of all live objects at a young generation collection to be promoted to the older generation without any tenuring. There is no "promoteall" modifier in JDK 1.4.1, but similar behavior can be achieved by controlling the tenuring distribution.
  - The number of times an object is aged in the young generation is controlled by the option MaxTenuringThreshold. Setting this option to 0 means objects are not copied, but are promoted directly to the older generation. SurvivorRatio should be increased to 20000 or a high value so that Eden occupies most of the Young Generation Heap space.
  - **-XX:MaxTenuringThreshold=0 -XX:SurvivorRatio=20000**

Tuning Garbage Collection

- For example:

  java -server -Xms512m -Xmx512m -XX:NewSize=64m -XX:MaxNewSize=64m \
  -XX:SurvivorRatio=20000 -XX:MaxTenuringThreshold=0 \
  -XX:+UseParNewGC -XX:+UseConcMarkSweepGC application

Tuning Garbage Collection

- **Controlling the Concurrent collection initiation**
  - The concurrent collector background thread starts running when the percentage of allocated space in the old generation goes above the **-XX:CMSInitiatingOccupancyFraction**, default value is 68%. This value can be changed and the concurrent collector can be started earlier by specifying the following option:
  - -XX:CMSInitiatingOccupancyFraction=<percent>
  - For example:

    java -server -Xms512m -Xmx512m -XX:NewSize=64m -XX :MaxNewSize=64m \
    -XX:SurvivorRatio=20000 -XX:MaxTenuringThreshold=0 \
    -XX:+UseParNewGC -XX:+UseConcMarkSweepGC \
    -XX:CMSInitiatingOccupancyFraction=35 application

Tuning Garbage Collection

Using the Throughput Collectors

- The young generation, parallel scavenge collector, can be enabled by using the -XX:UseParallelGC option. The older generation collector need not be specified since the mark-compact collector is used by default.
- For 32 bit usage:

  java -server -Xms3072m -Xmx3072m -XX:NewSize=2560m \
  -XX:MaxNewSize=2560m XX:SurvivorRatio=2 \
  -XX:+UseParallelGC application

- For 64 bit usage:

  java -server -d64 -Xms8192m -Xmx8192m -XX:NewSize=7168m \
  -XX:MaxNewSize=7168m XX:SurvivorRatio=2 \
  -XX:+UseParallelGC application

Tuning Garbage Collection

- Note:

  -XX:TargetSurvivorRatio is a tenuring threshold that is used to copy the tenured objects in the young generation. With large heaps and a SurvivorRatio of 2, survivor semi-space might be wasted, as the TargetSurvivorRatio by default is 50. This could be increased to maybe 75 or 90, maximizing use of the space.

Tuning Garbage Collection

- **Controlling the Degree of Parallelism**
  - Again, by default, the parallel scavenge collector will start as many threads as CPUs on a machine, but if the degree of parallelism needs to controlled, then it can be specified by the following switch:

    -XX:ParallelGCThreads=<desired parallelism>
  - Default value is equal to number of CPUs.
  - For example, to use 4 parallel threads to process young generation collection:

    java -server -Xms3072m -Xmx3072m -XX:NewSize=2560m \
    -XX:MaxNewSize=2560m XX:SurvivorRatio=2 \
    -XX:+UseParallelGC **-XX:ParallelGCThreads=4** application

Tuning Garbage Collection

- **Adaptive Sizing for Performance**
  - The Parallel scavenge collector performs better when used with the **-XX:+UseAdaptiveSizePolicy**. This automatically sizes the young generation and chooses an optimum survivor ratio to maximize performance. The parallel scavenge collector should always be used with the -XX:UseAdaptiveSizePolicy.
  - For example:

    java -server -Xms3072m -XX:+UseParallelGC \
    -XX:+UseAdaptiveSizePolicy application

Tuning Garbage Collection

- **Sizing the Young Generation**
  - **Low Pause apps**
    - The young generation's copying collector does not have fragmentation problems, but could have locality problems, depending on the size of the generation. The young heap must be sized to maximize the collection of short-term objects, which would reduce the number of long-term objects that are promoted or tenured. Frequency of the collection cycle is also determined by heap size, so the young heap must be sized for optimal collection frequency as well.
    - Basically, finding the optimal size for the young generation is pretty easy. The rule of thumb is to make it about as large as possible, given acceptable collection times. There is a certain amount of fixed overhead with each collection, so their frequency should be minimized.

Tuning Garbage Collection

- **Throughput apps**
  - The young generation is sized the opposite with the throughput collector. Instead of using smaller younger generation to keep the pause down, the young generation is made very big, maybe in gigabytes so that the throughput collector with AdaptiveTenuring can make use of this space to parallelize young generation collection.
  - Since throughput or enterprise applications are not pause sensitive, this gives them more application time and when a collection does occur, the collection is parallelized over many CPUs, usually more than 8.

Tuning Garbage Collection

- **Sizing the Old Generation**
  - **Low Pause apps**
    - The concurrent collector manages the old-generation heap so it needs to be carefully sized, taking into account the call-setup rate and active duration of call setups. An undersized heap will lead to fragmentation, increased collection cycles, and possibly a stop-the-world traditional mark-sweep collection.
    - An oversized heap will lead to increased collection times and smear problems. If the heap does not fit into physical memory, it will magnify these problems.

Tuning Garbage Collection

- **Throughput apps**
  - Similar to the young generation sizing, the throughput apps usually have a bigger younger generation, and a smaller older generation. The idea is to increase the number of temporary objects and minimize the intermediate objects. Only needed long term objects get stored in the older heap.

Tuning Garbage Collection

- **An Undersized Heap Causes Fragmentation**
  - An undersized heap can lead to fragmentation because the old-generation collector is a mark-sweep collector, and hence never compacts the heap. When the collector frees objects, it combines adjacent free spaces into a single larger space, so that in can be optimally allocated for future objects.
  - Over time, an undersized heap may begin to fragment, making it difficult to find space to allocate an object. If space cannot be found, a collection takes place prematurely, and the concurrent collector cannot be used, because that object must be allocated immediately. The traditional mark-sweep collector runs, causing the application to pause during the collection. Again, simply increasing the heap size, without making it too large, could prevent these conditions from arising.

Tuning Garbage Collection

- Note:
  - The **-XX:+UseCMSCompactAtFullCollection** option can be used to enable compaction of the old generation heap. This hurts performance but will eliminate the fragmentation problem.
  - The above option can be used in conjunction with this option to **-XX:CMSFullGCsBeforeCompaction=0** to force compaction.
  - The **-XX:CMSInitiatingOccupancyFraction**=<**percent**> can be set lower so that the concurrent collection can be started earlier to reduce heap fragmentation. Set this to 30 or lower.

Tuning Garbage Collection

- **An Oversized Heap Increases Collection Times**
  - When using the concurrent collector, the cost of each collection is directly proportional to the size of the heap. So an oversized heap is more costly to the application.
  - The pause times associated with the collections will still not be that bad, but the time the collector spends collecting will increase. This does not pause the application, but does take valuable CPU time away from the application.
  - To aid its interaction with the young collector, the old heap is managed using a card table, in which each card represents a subregion of the heap.
  - Increasing the heap size increases the number of cards and the total size of the data structures associated with managing the free lists in the old collector. An increased heap size and a high call rate will add to the pressure on the old collector, forcing it to search more data structures, and thus increasing collection times.

Tuning Garbage Collection

- **Locality of Reference Problems with an Oversized Heap**
  - Another problem with an oversized heap is "locality of reference." This is related to the operating system, which manages memory. If the heap is larger than physical memory, parts of the heap will reside in virtual memory.
  - Because the concurrent collector looks at the heap as one contiguous chunk of memory, the objects allocated and deallocated could reside anywhere in the heap. That some objects will be in virtual memory leads to paging, translation-lookahead-buffer (TLB) misses, and cache-memory misses.
  - Some of these problems are solved by reducing the heap size so that it fits entirely in physical memory, but that still does not eliminate the TLB misses or cache-memory misses, because object references in a large heap may still be very far away from each other.

Tuning Garbage Collection

- This problem with fragmentation and locality of reference is called a "spread," or " smear," problem. Optimizing heap sizes will help avoid this problem.

- **Using ISMs and Variable Page Sizes To Overcome Locality Problems**
  - TLB misses can be minimized or almost eliminated by using Intimate Shared Memory (ISM) on Solaris. ISM makes available 4 MB pages to the JVM. So instead of accessing many 8KB pages, default page size, less number of pages are accessed since the page size is now 4 MB. Striding across the heap will incur less page faults.
  - ISM can be enabled by using **-XX:+UseISM** option. This option is operating system dependent and works only on Solaris and Linux. Changes need to made to /etc/system file to include the following parameters:

Tuning Garbage Collection

- On the Horizon & Research Ideas
  - **Self Tuning JVM**
    - Modeling described above can be used to tune GC and in turn application behavior to improve application perofrmance. But this modeling process is static ie. verbose:gc logs are needed to analyze application behavior and then changes are made to the Java code and Java environment.
    - How about making this dynamic by incorporating modeling into the JVM itself ? The JVM could interact with the garbage collector to construct the model dynamically and based on criteria like throughput, pause, sequential overhead, etc. adjust the environment to meet the cirteria. This idea was presented at USENIX, JVM02-WIPS [2].

Tuning Garbage Collection

- **Ergonomics**
  - Future JVMs might provide modes where the developer might not have to remember the different tuning options, and instead specify a desired functionality like realtime performance, pause or throughput oriented, etc., and the JVM would automatically choose the right collector and the environment suitable for this.
- **Hardware based Garabage collection**
  - Garbage collection is reaching a stage where it is now stable and so can be supplemented in hardware through GC specific instructions or a GC accelerator. This should allow GC issues to become a non-issue, and allow Java application performance to reach or exceed that of C applications.

Tuning Garbage Collection

- **Enhancements to the concurrent collector**
  - The concurrent collector will be enhanced in future releases, and compaction of the older generation heap might happen automatically. Incremental functionality will be added to the collector so that the concurrent processing is done incrementally allowing the application threads to run more.
- **Enhancements to the parallel collector**
  - The parallel collector is now stop-the-world. This behavior could be changed and application threads could run on some CPUs while collection happens on the other CPUs. This again allows more time for the application threads, and and not affect cache-affinity, a very important factor for performance.

Tuning Garbage Collection

- **Behaviour Based Training**
  - Prior to the J2SE platform version 5.0 tuning for garbage collection consisted principally of specifying the size of the the overall heap and possibly the size of the generations in the heap. Other controls for tuning garbage collection include the size of the survivor spaces in the young generation and the threshold for promotion from the young generation to the old generation.
  - Tuning required of a series of experiments with different values of these parameters and the use of specialized tools or just good judgment to decide when garbage collection was performing well.

Tuning Garbage Collection

- In version 5.0 two parameters based on the desired behavior of the application have been introduced. These are
  - Maximum pause time goal
  - Application throughput goal.
- Setting these goals can be used to tune garbage collection. It should be emphasized that the goals cannot always be met. The application requires heap large enough to at least hold all of the live data. That minimum heap size my preclude reaching these desired goals.

Tuning Garbage Collection

- **Maximum pause time goal**
  - The pause time is the duration during which the garbage collector stops the application and recovers space that is no longer in use. The intent of the maximum pause time goal is to limit the longest of these pauses.
  - An average time for pauses and a variance on that average is maintained by the garbage collector. The average is taken from the start of the execution but is weighted so that more recent pauses count more heavily.
  - If the average plus the variance of the pause times is greater than the maximum pause time goal, then the garbage collector considers that the goal is not being met.
  - The maximum pause time goal is specified with the command line flag

    -XX:MaxGCPauseMillis=<nnn>

Tuning Garbage Collection

- This is interpreted as a hint to the garbage collector that pause times of <nnn> milliseconds or less are desired. The garbage collector will adjust the Java heap size and other garbage collection related parameters in an attempt to keep garbage collection pauses shorter than <nnn> milliseconds.
- By default there is no maximum pause time goal. These adjustments may cause the garbage collector to occur more frequently, reducing the overall throughput of the application. In some cases the desired pause time goal cannot be met.
- The throughput collector is a generational collector so there are separate collections for the young generation and the old generation. Averages and variances are kept separately for each generation.
- The maximum pause time goal is applied to the average plus the variance of the collections of each generation separately. Each generation may separately fail to meet the pause time goal.

Tuning Garbage Collection

- **Throughput goal**
  - The throughput goal is measured in terms of the time spent collecting garbage and the time spent outside of garbage collection (referred to as application time). The goal is specified by the command line flag

    -XX:GCTimeRatio=<nnn>
  - The ratio of garbage collection time to application time is

    1 / (1 + <nnn>)
  - For example -XX:GCTimeRatio=19 sets a goal of 1/20th or 5% of the total time for garbage collection.
  - The time spent in garbage collection is the total time for both the young generation and old generation collections combined. If the throughput goal is not being met, the sizes of the generations are increased in an effort to increase the time the application can run between collections.

Tuning Garbage Collection

- **Footprint goal**
  - If the throughput and maximum pause time goals have been met, the garbage collector reduces the size of the heap until one of the goals (invariably the throughput goal) cannot be met. The goal that is not being met is then addressed.

Tuning Garbage Collection

- **Other Changes**
  - **Thread-local allocation buffers**
    - The resizing of thread-local allocation buffers has been re-implemented. When a thread-local allocation buffer fills, the size of the new buffer depends on the allocation pattern of the thread. Threads that do allocate more memory will get larger buffers.

Tuning Garbage Collection

- **Other Options**
  - **-XX:+AggressiveOpts**
    Turns on point performance optimizations that are expected to be on by default in upcoming releases. The changes grouped by this flag are minor changes to JVM runtime compiled code and not distinct performance features (such as BiasedLocking and ParallelOldGC). This is a good flag to try the JVM engineering team's latest performance tweaks for upcoming releases. Note: this option is *experimental*! The specific optimizations enabled by this option can change from release to release and even build to build. You should reevaluate the effects of this option with prior to deploying a new release of Java.

Tuning Garbage Collection

- **-XX:+UseBiasedLocking**
  Enables a technique for improving the performance of uncontended synchronization. An object is "biased" toward the thread which first acquires its monitor via a monitorenter bytecode or synchronized method invocation; subsequent monitor-related operations performed by that thread are relatively much faster on multiprocessor machines. Some applications with significant amounts of uncontended synchronization may attain significant speedups with this flag enabled; some applications with certain patterns of locking may see slowdowns, though attempts have been made to minimize the negative impact.

Tuning Garbage Collection

- **-XX:LargePageSizeInBytes=256m**
  Causes the Java heap, including the permanent generation, and the compiled code cache to use as a minimum size one 256 MB page (for those platforms which support it).
- **-XX:TargetSurvivorRatio=90**
  Allows 90% of the survivor spaces to be occupied instead of the default 50%, allowing better utilization of the survivor space memory