Java theory and practice: Urban performance legends, revisited

Allocation is faster than you think, and getting faster

The Java™ language is the target of a lot of abuse for performance. And while some of it may well be deserved, a tour of message board and newsgroup postings on the subject shows that there is a great deal of misunderstanding about how a Java Virtual Machine (JVM) actually works. In this month's *Java theory and practice*, Brian Goetz pokes some holes in the oft-repeated performance myth of slow allocation in JVMs.

Brian Goetz has been a professional software developer for over 18 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California, and he serves on several JCP Expert Groups. See Brian's published and upcoming articles in popular industry publications.

27 September 2005
Also available in Russian Japanese

Pop quiz: Which language boasts faster raw allocation performance, the Java language, or C/C++? The answer may surprise you -- allocation in modern JVMs is far faster than the best performing `malloc` implementations. The common code path for `new Object()` in HotSpot 1.4.2 and later is approximately 10 machine instructions (data provided by Sun; see Resources), whereas the best performing `malloc` implementations in C require on average between 60 and 100 instructions per call (Detlefs, et. al.; see Resources). And allocation performance is not a trivial component of overall performance -- benchmarks show that many real-world C and C++ programs, such as Perl and Ghostscript, spend 20 to 30 percent of their total execution time in `malloc` and `free` -- far more than the allocation and garbage collection overhead of a healthy Java application (Zorn; see Resources).

Go ahead, make a mess

You don't have to search through too many blogs or Slashdot postings to find confidently worded statements like "Garbage collection will never be as efficient as direct memory management." And, in a way, those statements are right -- dynamic memory management is not as fast -- it's often considerably faster. The `malloc/free` approach deals with blocks of memory one at a time, whereas the garbage collection approach tends to deal with memory management in large batches, yielding more opportunities for optimization (at the cost of some loss in predictability).

This "plausibility argument" -- that it is easier to clean up a mess in one big batch than to pick up individual pieces of dust throughout the day -- is borne out by the data. One study (Zorn; see Resources) also measured the effects of replacing `malloc` with the conservative Boehm-Demers-Weiser (BDW) garbage collector for a number of common C++ applications, and the result was that many of these programs exhibited speedups when running with a garbage collector instead of a traditional allocator. (And BDW is a conservative, nonmoving garbage collector, which greatly constrains its ability to optimize allocation and reclamation and improve memory locality; exact relocating collectors such as those used in JVMs can do far better.)

Allocation in JVMs was not always so fast -- early JVMs indeed had poor allocation and garbage collection performance, which is almost certainly where this myth got started. In the very early days, we saw a lot of "allocation is slow" advice -- because it was, along with everything else in early JVMs -- and performance gurus advocated various tricks to avoid allocation, such as object pooling. (Public service announcement: Object pooling is now a serious performance loss for all but the most heavyweight of objects, and even then it is tricky to get right without introducing concurrency bottlenecks.) However, a lot has happened since the JDK 1.0 days; the introduction of generational collectors in JDK 1.2 has enabled a much simpler approach to allocation, greatly improving performance.

Generational garbage collection

A generational garbage collector divides the heap into multiple generations; most JVMs use two generations, a "young" and an "old" generation. Objects are allocated in the young generation; if they survive past a certain number of garbage collections, they are considered "long lived" and get promoted into the old generation.

While HotSpot offers a choice of three young-generation collectors (serial copying, parallel copying, and parallel scavenge), they are all forms of "copying" collectors and have several important characteristics in common. A copying collector divides the memory space in half and only uses one half at a time. Initially, the half that is in use forms one big block of available memory; the allocator satisfies allocation requests by returning the first N bytes of the portion that is not in use, moving a pointer that separates the "used" part from the "free" part, as shown by the pseudocode in Listing 1. When the half in use fills up, the garbage collector copies all the live objects (those that are not garbage) to the bottom of the other half (compacting the heap), and starts allocating from the other half instead.

Listing 1. Behavior of allocator in the presence of a copying collector

```
void *malloc(int n) {
    if (heapTop - heapStart < n)
        doGarbageCollection();

    void *wasStart = heapStart;
    heapStart += n;
    return wasStart;
}
```

From this pseudocode, you can see why copying collectors enables such fast allocation -- allocating a new object is nothing more than checking to see if there's enough space left in the heap and bumping a pointer if there is. No searching free lists, best-fit, first-fit, lookaside lists -- just grab the first N bytes out of the heap and you're done.

What about deallocation?

But allocation is only half of memory management -- deallocation is the other half. It turns out that for most objects, the direct garbage collection cost is -- zero. This is because a copying collector does not need to visit or copy dead objects, only live ones. So objects that become garbage shortly after allocation contribute no workload to the collection cycle.

It turns out that the vast majority of objects in typical object-oriented programs (between 92 and 98 percent according to various studies) "die young," which means they become garbage shortly after they are allocated, often before the next garbage collection. (This property is called the *generational hypothesis* and has been empirically tested and found to be true for many object-oriented languages.) Therefore, not only is allocation fast, but for most objects, deallocation is free.

Thread-local allocation

If the allocator were truly implemented as shown in Listing 1, the shared `heapStart` field would quickly become a significant concurrency bottleneck, as every allocation would involve acquiring the lock that guards this field. To avoid this problem, most JVMs use *thread-local allocation blocks*, where each thread

allocates a larger chunk of memory from the heap and services small allocation requests sequentially out of that thread-local block. As a result, the number of times a thread has to acquire the shared heap lock is greatly reduced, improving concurrency. (It is more difficult and costly to address this problem in the context of a traditional `malloc` implementation; building both thread support and garbage collection into the platform facilitates synergies such as this one.)

Stack allocation

C++ offers programmers a choice of allocating objects on the heap or on the stack. Stack-based allocation is more efficient: allocation is cheaper, deallocation costs are truly zero, and the language provides assistance in demarcating object lifecycles, reducing the risk of forgetting to free the object. On the other hand, in C++, you need to be very careful when publishing or sharing references to stack-based objects because stack-based objects are automatically freed when the stack frame is unwound, leading to dangling pointers.

Another advantage of stack-based allocation is that it is far more cache-friendly. On modern processors, the cost of a cache miss is significant, so if the language and runtime can help your program achieve better data locality, performance will be improved. The top of the stack is almost always "hot" in the cache, whereas the top of the heap is almost always "cold" (because it has likely been a long time since that memory was used). As a result, allocating an object on the heap will likely entail more cache misses than allocating that object on the stack.

Worse, a cache miss when allocating an object on the heap has a particularly nasty memory interaction. When allocating memory from the heap, the contents of that memory are garbage -- whatever bits happen to be left over from the last time that memory was used. If you allocate a block of memory on the heap that is not already in the cache, execution will stall while the contents of that memory are brought into the cache. Then, you will immediately overwrite those values that you paid to bring into the cache with zeros or other initial values, resulting in a lot of wasted memory activity. (Some processors, such as Azul's Vega, include hardware support for accelerating heap allocation.)

Escape analysis

The Java language does not offer any way to explicitly allocate an object on the stack, but this fact doesn't prevent JVMs from still using stack allocation where appropriate. JVMs can use a technique called *escape analysis*, by which they can tell that certain objects remain confined to a single thread for their entire lifetime, and that lifetime is bounded by the lifetime of a given stack frame. Such objects can be safely allocated on the stack instead of the heap. Even better, for small objects, the JVM can optimize away the allocation entirely and simply hoist the object's fields into registers.

Listing 2 shows an example where escape analysis can be used to optimize away heap allocation. The `Component.getLocation()` method makes a defensive copy of the component location, so that a caller cannot accidentally change the actual location of the component. Calling `getDistanceFrom()` first gets the location of the other component, which involves an object allocation, and then uses the `x` and `y` fields of the `Point` returned by `getLocation()` to compute the distance between two components.

Listing 2. Typical defensive copying approach to returning a compound value

```
public class Point {
  private int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  public Point(Point p) { this(p.x, p.y); }
  public int getX() { return x; }
  public int getY() { return y; }
}

public class Component {
  private Point location;
  public Point getLocation() { return new Point(location); }

  public double getDistanceFrom(Component other) {
```

```
    Point otherLocation = other.getLocation();
    int deltaX = otherLocation.getX() - location.getX();
    int deltaY = otherLocation.getY() - location.getY();
    return Math.sqrt(deltaX*deltaX + deltaY*deltaY);
  }
}
```

The `getLocation()` method does not know what its caller is going to do with the `Point` it returns; it might retain a reference to it, such as putting it in a collection, so `getLocation()` is coded defensively. However in this example, `getDistanceFrom()` is not going to do this; it is just going to use the `Point` for a short time and then discard it, which seems like a waste of a perfectly good object.

A smart JVM can see what is going on and optimize away the allocation of the defensive copy. First, the call to `getLocation()` will be inlined, as will the calls to `getX()` and `getY()`, resulting in `getDistanceFrom()` effectively behaving like Listing 3.

Listing 3. Pseudocode describing the result of applying inlining optimizations to getDistanceFrom()

```
  public double getDistanceFrom(Component other) {
    Point otherLocation = new Point(other.x, other.y);
    int deltaX = otherLocation.x - location.x;
    int deltaY = otherLocation.y - location.y;
    return Math.sqrt(deltaX*deltaX + deltaY*deltaY);
  }
```

At this point, escape analysis can show that the object allocated in the first line never escapes from its basic block and that `getDistanceFrom()` never modifies the state of the `other` component. (By *escape*, we mean that a reference to it is not stored into the heap or passed to unknown code that might retain a copy.) Given that the `Point` is truly thread-local and its lifetime is known to be bounded by the basic block in which it is allocated, it can be either stack-allocated or optimized away entirely, as shown in Listing 4.

Listing 4. Pseudocode describing the result of optimizing away allocation in getDistanceFrom()

```
  public double getDistanceFrom(Component other) {
    int tempX = other.x, tempY = other.y;
    int deltaX = tempX - location.x;
    int deltaY = tempY - location.y;
    return Math.sqrt(deltaX*deltaX + deltaY*deltaY);
  }
```

The result is that we get exactly the same performance as we would if all the fields were public while retaining the safety that encapsulation and defensive copying (among other safe coding techniques) give us.

Escape analysis in Mustang

Escape analysis is an optimization that has been talked about for a long time, and it is finally here -- the current builds of Mustang (Java SE 6) can do escape analysis and convert heap allocation to stack allocation (or no allocation) where appropriate. The use of escape analysis to eliminate some allocations results in even faster average allocation times, reduced memory footprint, and fewer cache misses. Further, optimizing away some allocations reduces pressure on the garbage collector and allows collection to run less often.

Escape analysis can find opportunities for stack allocation even where it might not be practical to do so in the source code, even if the language provided the option, because whether a particular allocation gets optimized away is determined based on how the result of an object-returning method is actually used in a particular code path. The `Point` returned from `getLocation()` may not be suitable for stack allocation in all cases, but once the JVM inlines `getLocation()`, it is free to optimize each invocation separately, offering us the best of both worlds: optimal performance with less time spent making low-level, performance-tuning decisions.

Conclusion

JVMs are surprisingly good at figuring out things that we used to assume only the developer could know.

By letting the JVM choose between stack allocation and heap allocation on a case-by-case basis, we can get the performance benefits of stack allocation without making the programmer agonize over whether to allocate on the stack or on the heap.

Resources

Learn

Java theory and practice: Urban performance legends: Explores some performance myths and how they came about.

Escape analysis for Java: A paper from IBM research that was presented at OOPSLA '99 on implementing escape analysis.

A brief history of garbage collection: Comparison of various garbage collection approaches, including the copying approach used by HotSpot for the young generation.

Garbage collection in the Java Virtual Machine: A presentation from JavaOne 2003 which provides the data on the cost of allocation in HotSpot.

Memory allocation costs in large C and C++ programs (Detlefs, Dosser, and Zorn): Examines the cost of allocation in numerous C and C++ applications.

The measured cost of conservative garbage collection (Benjamin Zorn): Compares the performance of several allocators for C and C++, including the BDW collector.

The Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

The Boehm-Demers-Weiser conservative garbage collector for C: It is possible to use garbage collection in C; the BDW collector is a replacement for `malloc()` and `free()` that uses a garbage-collected heap.

Discuss

developerWorks blogs: Get involved in the developerWorks community.

Dig deeper into Java technology on developerWorks

Overview

New to Java programming

Technical library (tutorials and more)

Forums

Blogs

Communities

Downloads and products

Open source projects

Standards

Events