ConcurrentHashMap is hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates. I recently came across this code during testing, and one part really got my attention. To generate the hash, ConcurrentHashMap uses an algorithm based on bitshifting and bitwise operations.

========================================

Variant of single-word Wang/Jenkins hash

========================================

private static int hash(int h) {

// Spread bits to regularize both segment and index locations,

// using variant of single-word Wang/Jenkins hash.

h += (h << 15) ^ 0xffffcd7d;

h ^= (h >>> 10);

h += (h << 3);

h ^= (h >>> 6);

h += (h << 2) + (h << 14);

return h ^ (h >>> 16);

}

According to the comment in the code, this method applies a supplemental hash function to a given hashCode, which defends against poor quality hash functions.

Good hash functions are important as a hash table effectively turns from a map to a linked list, in the worst case, all keys in the same bucket. There are also other considerations that come into play such as the performance of hash calculation and the number of buckets. Dr. Heinz M. Kabutz explains the power of [“power-of-two number of buckets”](http://www.javaspecialists.co.za/archive/Issue054.html) which gives us some good starting point to understand what is really going on here.

Let’s look at the code above and see how things change, line-by-line. To make things simple, I use **int 1** to perform all the operations.

In Java, the **int** data type is a 32-bit signed two’s complement integer. To represent int 1 in binary code, we have the following:

h=1 > 0000-0000-0000-0000-0000-0000-0001

Now, let’s dissect the following line:

h += (h << 15) ^ 0xffffcd7d

h1 = h << 15 = 0000-0000-0000-0000-1000-0000-0000-0000

hex = 0xffffcd7d = 1111-1111-1111-1111-1100-1101-0111-1101

h2 = h1 ^ hex = 1111-1111-1111-1111-0100-1101-0111-1101

h2 + h = 1111-1111-1111-1111-0100-1101-0111-1110

Using the same thought processing and applying it to each line, we end-up with

h += (h << 15) ^ 0xffffcd7d = 1111-1111-1111-1111-0100-1101-0111-1110

h ^= (h >>> 10) = 1111-1111-1100-0000-1011-0010-1010-1101

h += (h << 3) = 1111-1101-1100-0110-0100-1000-0001-0101

h ^= (h >>> 6) = 1111-1110-0011-0001-0101-0001-0011-0101

h += (h << 2) + (h << 14) = 0100-1011-0100-0011-1101-0110-0000-1001

h ^= (h >>> 16) = 0100-1011-0100-0011-1001-1101-0100-1010

Result:  
Bin = 0100-1011-0100-0011-1001-1101-0100-1010  
Decimal = 1,262,722,378

# HashMap requires a better hashCode() - JDK 1.4 Part II

by Dr. Heinz M. Kabutz

http://www.javaspecialists.co.za/skin/images2/pixle_gr.jpg

Welcome to the 54th edition of **The Java(tm) Specialists' Newsletter** sent to 4289 Java Specialists in [85 countries](http://www.javaspecialists.co.za/countries.jsp). I seem to have to explain with each newsletter why you have not heard from me for soooo long. Last night I was speaking to a Java guru at Dimension Data in Cape Town, and he mentioned that somehow he had gotten unsubscribed from my newsletter. "No", said I, "you are still subscribed. It has just been rather quiet lately."

We have now finally completed moving into our new [home](http://www.javaspecialists.co.za/pics/housefromlawn.jpg), which includes an office with a view across False Bay all the way to Cape Point (when the weather is nice, which is most of the time :-), so perhaps, the newsletters will appear more frequently. Last week I had to go on an emergency mission to Europe to try help solve some Java deadlock in a very large piece of code. If you get stuck with a deadlock in Java, or even worse, a livelock, you are welcome to [contact me](http://www.javaspecialists.co.za/contact.jsp) and I will see if I can be of help.

**Calling all Mac fans & haters!** - I am considering buying a Mac notebook for Java development and to develop my courseware. If you have any horror stories (or success stories) regarding Java and Mac OS X, please let me know.

**Our New Java Design Patterns Self-Study Course** [is Now Available](http://www.javaspecialists.eu/courses/dpc/dpc-self-paced.jsp)

### HashMap requires a better hashCode() - JDK 1.4 Part II

My explorations into the JDK 1.4 source code did not come unrewarded. Within a day of starting my expedition, I stumbled across something that ended up baffling several Java experts, including yours truly: **java.util.HashMap now always has a power-of-two number of buckets, irrespective of what size and load factor you specify!!!**

Conventional wisdom says that the buckets in a hash table should be a prime number size. That's what Craig Larman and Rhett Guthrie argued in their excellent (albeit a bit dated) book on Java Performance. The reason why the number of buckets should be prime is so that the hash values are well distributed, even when the hash function is not very well distributed. I'm sure you will find the theory behind it in some dusty CompSci 101 textbook somewhere. The worst possible number of buckets is a power-of-two! So, why on earth did Sun Microsystems deliberately change java.util.HashMap to force it to have a power-of-two number of buckets? (No, "Sun wants to sell more hardware" is not the correct answer ;-)

I posed this question to [Bruce Eckel](http://www.bruceeckel.com), who also did not know the reason for this. However, he knew Joshua Bloch, the author of java.util.HashMap, so we asked him. Before reading the answer, spend a few minutes thinking about why this could be.

But before I expound the reason, and the problems that came with the change, I want to mention a problem that someone on JavaDesk (an *advanced* yahoogroups Java User Group with excellent content) mentioned: Since JDK 1.3, integer division and remainder calculation has become really slow. I've written a basic microbenching framework based on some code originally posted in the JavaDesk group, and also shown in the[bug report](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4631373) where the slowdown was described. One has to be careful with Microbenchmarking, as you can read up in[this article](http://servlet.java.sun.com/javaone/sf2002/conf/sessions/display-1816.en.jsp) from the San Francisco conference.

We start with a simple interface

*/\*\**

*\* Benchmark contains some calculation that is executed inside*

*\* the doCalculation() method a certain number of times. The*

*\* number of iterations is returned from the method.*

*\*/*

**public interface** Benchmark {

**int** doCalculation();

}

Add to it a little performance testing routine:

**public class** PerformanceTest {

**private static final int** REPEATS = **10**;

**public static void** main(String[] args) {

**if** (args.length != **1**) {

usage();

}

System.out.println(

**"JVM version:"** + System.getProperty(**"java.version"**));

**try** {

evaluate((Benchmark)Class.forName(args[**0**]).newInstance());

System.out.println();

} **catch** (Exception ex) {

ex.printStackTrace();

usage();

}

}

**private static void** usage() {

System.err.println(

**"usage: java PerformanceTest BenchmarkClass"**);

System.err.println(

**"\tBenchmarkClass is a class implementing Benchmark"**);

System.exit(**1**);

}

**private static void** evaluate(Benchmark benchmark) {

*// do the function once to "warm up" HotSpot compiler*

benchmark.doCalculation();

**long** average = **0**;

**for** (**int** i = **0**; i < REPEATS; i++) {

**long** time = -System.currentTimeMillis();

**int** iterations = benchmark.doCalculation();

time += System.currentTimeMillis();

System.out.print(iterations / time);

System.out.print(**" "**);

System.out.flush();

average += iterations / time;

}

System.out.println();

System.out.println(

**"Average "**

+ (average / REPEATS)

+ **" iterations per millisecond"**);

}

}

The most basic test that I want to show is a simple empty loop. This "benchmark" is really quite useless - any half-decent compiler should optimize it away anyway. In addition, if you unroll the loop by 16 steps, you end up doing nothing 16 times and slice the loop into a 16th of the length. The reason for testing this is simply to have some indication of whether my other tests are actually going to measure anything besides the **for** loop. Please also note that *in general* each version of JDK makes programs run *faster*, but that you will always find some benchmark which seems to demonstrate that the old JDK was faster.

**public class** EmptyLoopBenchmark **implements** Benchmark {

**private static final int** ITERATIONS = **10** \* **1000** \* **1000**;

**public int** doCalculation() {

**for** (**int** i = **0**; i < ITERATIONS; i++) {

; // NOP

}

**return** ITERATIONS;

}

}

The results are interesting, although they don't really tell us very much:

JVM version:1.2

333333 333333 333333 200000 333333 ...

Average 311056 iterations per millisecond

JVM version:1.3.1\_03

250000 200000 250000 166666 250000 ...

Average 217941 iterations per millisecond

JVM version:1.4.0

142857 166666 166666 125000 166666 ...

Average 151785 iterations per millisecond

JVM version:1.4.1-beta

82644 76923 76923 76923 83333 ...

Average 78926 iterations per millisecond

Is the JDK 1.4.1 then slower than the JDK 1.2 ?!? Nope, not in general, but as I said, with benchmarks you can find strange things. However, let's look at a benchmark on *addition*. We have to use the val variable after the loop, otherwise a clever compiler could optimize the whole loop away.

**public class** AdditionBenchmark **implements** Benchmark {

**private static final int** ITERATIONS = **10** \* **1000** \* **1000**;

**private int** memory;

**public int** doCalculation() {

**int** val = **0**;

**for** (**int** i = **0**; i < ITERATIONS; i++) {

val = val + i;

}

memory = val;

return ITERATIONS;

}

}

The results on my little Asus Notebook are as follows:

JVM version:1.2

333333 333333 200000 333333 500000 ...

Average 344999 iterations per millisecond

JVM version:1.3.1\_03

200000 200000 200000 243902 200000 ...

Average 209390 iterations per millisecond

JVM version:1.4.0

125000 125000 125000 123456 125000 ...

Average 123853 iterations per millisecond

JVM version:1.4.1-beta

83333 76923 76335 90909 76923 ...

Average 79486 iterations per millisecond

These results at best demonstrate that addition is blindingly fast, as we would expect. Note that we have to compare the addition benchmark to the empty loop benchmark to realise that the actual addition operation is negligible. However, what about remainder, the % operator?

**public class** RemainderBenchmark **implements** Benchmark {

**private static final int** ITERATIONS = **10** \* **1000** \* **1000**;

**private int** memory;

**public int** doCalculation() {

**int** val = **0**;

**for** (**int** i = **0**; i < ITERATIONS; i++) {

val = i % **11**;

}

memory = val;

return ITERATIONS;

}

}

Here we see that performance has dropped significantly since JDK 1.2, as described in the bug report mentioned earlier:

JVM version:1.2

62500 62500 62111 58823 66666 ...

Average 62520 iterations per millisecond

JVM version:1.3.1\_03

17513 17513 17513 17211 17513 ...

Average 17457 iterations per millisecond

JVM version:1.4.0

14903 16920 16638 15360 16366 ...

Average 16051 iterations per millisecond

JVM version:1.4.1-beta

17211 17513 17211 16920 16920 ...

Average 17217 iterations per millisecond

These results is somewhat disturbing! Isn't that one of the important functions that gets executed in the traditional java.util.HashMap class? The engineers at Sun (Joshua Bloch and Doug Lea) decided to rather use a bit masking approach instead of remainder, because that is much faster:

**public class** MaskingBenchmark **implements** Benchmark {

**private static final int** ITERATIONS = **10** \* **1000** \* **1000**;

**private int** memory;

**public int** doCalculation() {

**int** val = **0**;

**for** (**int** i = **0**; i < ITERATIONS; i++) {

val = i & **0x000000ff**;

}

memory = val;

return ITERATIONS;

}

}

You can see that the performance values are far more encouraging:

JVM version:1.2

166666 142857 166666 166666 125000 ...

Average 158416 iterations per millisecond

JVM version:1.3.1\_03

142857 142857 100000 140845 111111 ...

Average 131624 iterations per millisecond

JVM version:1.4.0

111111 142857 125000 125000 123456 ...

Average 128813 iterations per millisecond

JVM version:1.4.1-beta

76923 76923 83333 76335 83333 ...

Average 80849 iterations per millisecond

In order to protect Sun against claims by the [intellectual proletariat](http://www.humournet.com) ;-), who have up to now managed to get away with writing bad hash functions, Josh wrote a super-fast rehash() function inside java.util.HashMapthat attempted to redistribute the bits a little bit. Good hash code writers were rewarded with a nice improvement in performance for java.util.HashMap.get() in**JDK 1.4.0** thanks to the new masking trick:

**import** java.util.\*;

*/\*\**

*\* The lower-order bits are used as part of this hashcode*

*\* which is good if you are using JDK 1.4.0*

*\*/*

**public class** GoodHashcodeBenchmark **implements** Benchmark {

**private static final int** ITERATIONS = **1000** \* **1000**;

**private** Object memory;

**private** HashMap map = new HashMap(**203**);

**private** Integer[] values;

**public** GoodHashcodeBenchmark() {

**for** (**int** i=**0**; i < **1000**; i++) {

map.put(**new** Integer(i), **"Value"**);

}

TreeSet keys = **new** TreeSet(map.keySet());

values = (Integer[])keys.toArray(**new** Integer[**1000**]);

}

**public int** doCalculation() {

**for** (**int** i = **0**; i < ITERATIONS; i++) {

memory = map.get(values[i%**1000**]);

}

return ITERATIONS;

}

}

When we run this benchmark, we get the following values:

JVM version:1.2

5555 5882 5882 5524 5555 5555 5555 5847 5555 6250

Average 5716 iterations per millisecond

JVM version:1.3.1\_03

3846 3690 3846 3436 3846 3571 3831 3703 3690 3571

Average 3703 iterations per millisecond

JVM version:1.4.0

6250 6250 5847 6250 5882 5882 6622 5882 5882 5882

**Average 6062 iterations per millisecond**

JVM version:1.4.1-beta

4149 4347 4347 4329 4000 4329 4347 4545 4149 4347

Average 4288 iterations per millisecond

Despite the slowdown of JDK 1.[34].x with regards to remainder calculations, Joshua Bloch and Doug Lea still managed to actually make HashMap.get() function in the JDK 1.4.0 run faster than in the JDK 1.2! The observant reader would have noticed that JDK 1.4.1 runs slower than JDK 1.4.0 but faster than JDK 1.3.1. Before I explain why, let's have a look at what happens when you write a bad hash function:

**import** java.util.\*;

*/\*\**

*\* The lower-order bits are NOT used as part of this hashcode*

*\* which is bad if you are using JDK 1.4.0*

*\*/*

**public class** BadHashcodeBenchmark **implements** Benchmark {

**private static final int** ITERATIONS = **1000** \* **1000**;

**private** Object memory;

**private** HashMap map = new HashMap(**203**);

**private** Integer[] values;

**public** BadHashcodeBenchmark() {

**for** (**int** i=**0**; i < **1000**; i++) {

map.put(**new** Integer(i \* **1024**), **"Value"**);

}

TreeSet keys = **new** TreeSet(map.keySet());

values = (Integer[])keys.toArray(**new** Integer[**1000**]);

}

**public int** doCalculation() {

**for** (**int** i = **0**; i < ITERATIONS; i++) {

memory = map.get(values[i%**1000**]);

}

return ITERATIONS;

}

}

The result is obvious. Since the new JDK 1.4.0 HashMap is ultra-sensitive to lower-order bits being the same, due to it using masking instead of remainder, the HashMap basically becomes a linked list and the lookup for a single entry becomes O(n) instead of O(1). Oops:

JVM version:1.2

5524 5555 5882 5882 5524 5882 5555 5555 5524 5555

Average 5643 iterations per millisecond

JVM version:1.3.1\_03

3571 3690 3571 3703 3558 3703 3558 3703 3571 3690

Average 3631 iterations per millisecond

JVM version:1.4.0

173 172 154 171 173 171 168 167 172 172

**Average 169 iterations per millisecond**

JVM version:1.4.1-beta

4149 4347 4347 4149 4166 4347 4329 4347 4347 4329

Average 4285 iterations per millisecond

Bug reports started surfacing on the[bug parade](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4669519) saying that the HashMap in JDK 1.4.0 had a bug that caused it to be very slow. The following is an excerpt from a three-way conversation between Joshua Bloch, author of java.util.HashMap, Bruce Eckel and myself (quoted with permission):

**Joshua Bloch:** *The downside of using a power-of-two is that the resulting hash table is very sensitive to the quality of the hash function (hashCode). It is imperative that any change in the input must affect the low order bits of the hash value. (Ideally, it should affect all bits of the hash value with equal likelihood.) Because we have no assurance that this is true, we put in a secondary (or "defensive") hash function when we switched to the power-of-two hash table. This hash function is applied to the results of hashCode before masking off the low order bits. Its job is to scatter the information over all the bits, and in particular, into the low order bits. Of course it has to run \*very\* fast, or you lose the benefit of switching to the power-of-two-sized table. The original secondary hash function in 1.4 turned out to be insufficient. We knew that this was a theoretical possibility, but we thought that it didn't affect any practical data sets. We were wrong. The replacement secondary hash function (which I developed with the aid of a computer) has strong statistical properties that pretty much guarantee good bucket distribution.*

The rehash() method in JDK 1.4.1 is quite interesting, but let's hope that it does a good enough job. From my tests, it seems to distribute the bits beautifully, but you will have to test (I don't know how!) that your hash function is compatible with the rehash method of HashMap. Obviously, having a more complicated rehash() method causes a drop in performance (when the hash function was good anyway), but it gives us a better average performance than JDK 1.4.0 and JDK 1.3.1.

#### Moral of the story - write your own!

If you need to squeeze the last ounce of performance out of the JDK and you need to do that with hash tables, you should probably write your own stripped-down version of HashMap that works specifically for your key-domain.

#### Conclusion

I now understand the reason for using power-of-two for HashMap buckets, but, I am also disappointed. The old mechanism was so much more predictable in terms of distribution of keys into buckets. How am I going to explain this new mechanism to Java beginners?

On the other hand, this new knowledge allows me (and you) to amble up to people who claim to know Java and say: "Did you know that HashMap in JDK 1.4 forces the number of buckets to be a power-of-two?" If the innocent victim responds: "Yes", you can follow-up your question with "Why?" Try it out - and let me know the response :-)))

Heinz