JVM INTERNALS

Loop Unrolling

An elaborate mechanism for reducing loop iterations improves performance but can be thwarted by inadvertent coding.

by Chris Newland and Ben Evans

In previous articles in this series on the inner workings of the JVM, you have seen some of the Java HotSpot VM's just-in-time (JIT) compilation techniques, including escape analysis and lock elision. In this article, we discuss another automatic optimization, known as *loop unrolling*. This technique is used by the JIT compiler to make loops (such as Java's `for` or `while` loops) execute faster.

Because we'll be delving deep inside the JVM here, you will at times encounter C code and even some assembly language for the purpose of illustration, so hold on to your hats!

Let's start by considering the following piece of C code, which allocates space for 1 million longs and fills the space with 1 million long random numbers:

```
int main(int argv, char** argc) {
    int MAX = 1000000;

    long* data = (long*)calloc(MAX, sizeof(long));

    for (int i = 0; i < MAX; i++) {
        data[i] = randomLong();
    }
}
```

C can be thought of as a high-level language, but is that really the case? On an Apple Macintosh, the Clang compiler (with the `-S` switch to dump the assembly language in Intel format) produces the following output for the previous code:

```
_main:                                  ## @main
## BB#0:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $48, %rsp
    movl    $8, %eax
    movl    %eax, %ecx
    movl    $0, -4(%rbp)
    movl    %edi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    movl    $1000000, -20(%rbp)   ## imm = 0xF4240
    movslq  -20(%rbp), %rdi
    movq    %rcx, %rsi
    callq   _calloc
    movq    %rax, -32(%rbp)
    movl    $0, -36(%rbp)
LBB1_1:                                 ## =>This Inner Loop
    movl    -36(%rbp), %eax
    cmpl    -20(%rbp), %eax
    jge LBB1_4
## BB#2:                                ##   in Loop: Header
```

```
    callq   _randomLong
    movslq  -36(%rbp), %rcx
    movq    -32(%rbp), %rdx
    movq    %rax, (%rdx,%rcx,8)
## BB#3:                                ##   in Loop: Header
    movl    -36(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -36(%rbp)
    jmp LBB1_1
LBB1_4:
    movl    -4(%rbp), %eax
    addq    $48, %rsp
    popq    %rbp
    retq
```

Looking at the code, you can see that there is one call to the `calloc()` function at the start and only one call (per loop iteration) to the `randomLong()` function. There are two separate jumps, and the produced machine code is essentially the same as that produced from the following variant C code:

```
int main(int argv, char** argc) {
    int MAX = 1_000_000;

    long* data = (long*)calloc(MAX, sizeof(long));
    int i = 0;
    LOOP: if (i >= MAX)
        goto END;
    data[i] = randomLong();
    ++i;
    goto LOOP;
    END: return 0;
}
```

In the case of Java, the equivalent code would be something like this:

```
public class LoopUnroll {
    public static void main(String[] args) {
        int MAX = 1000000;

        long[] data = new long[MAX];
        java.util.Random random = new java.util.Ran

        for (int i = 0; i < MAX; i++) {
            data[i] = random.nextLong();
        }
    }
}
```

When it is compiled into bytecode, the code becomes:

```
public static void main(java.lang.String[]);
    Code:
       0: ldc           #2    // int 1000000
       2: istore_1
       3: iload_1
       4: newarray       long
       6: astore_2
       7: new           #3    // class java/util/Ra
      10: dup
      11: invokespecial #4    // Method java/util/R
      14: astore_3
      15: iconst_0
      16: istore        4
      18: iload         4
      20: iload_1
      21: if_icmpge     38
      24: aload_2
      25: iload         4
      27: aload_3
      28: invokevirtual #5    // Method java/util/R
      31: lastore
      32: iinc          4, 1
      35: goto          18
      38: return
```

These programs are very similar in the overall shape of the code. They all perform one operation on the `data` array per loop. However, real processors have pipelines of upcoming instructions, so if the program keeps moving forward linearly, the pipeline can be used efficiently because the next instruction to be executed is always immediately at hand.

But, if a jump instruction is encountered, the benefit of the instruction pipeline is typically lost, because the pipeline contents need to be dumped and reloaded from main memory with new opcodes starting from the jumped-to address. The performance penalty in such a case will be similar to a cache miss—an additional fetch from main memory.

Native code disassembly into readable assembly language is performed directly after the JIT thread emits the compiled method. It is an expensive operation that should not be used on production processes.

For a *back branch*—a jump to a previous point—as seen in a `for` loop, the effect on performance depends on the precise form of the branch prediction algorithm provided by the CPU. Section 3.4.1 of the *Intel 64 and IA-32 Architectures Optimization Reference Manual* [PDF] has details about branch prediction optimization for the specific chips it covers.

However, in the case of Java programs, there is more to this story because of HotSpot's JIT compiler. The JIT compiler contains several optimizations that can produce very different compiled code under favorable circumstances.

In particular, there are optimizations for counted loops (for example, `for` loops) that use an `int`, `short`, or `char` variable as the loop counter. The body of the loop is unrolled, and it is replaced by multiple copies of the loop body, arranged one after the other. This reworking of the loop reduces the number of back branches needed. In addition, it can generate a significant performance improvement over the assembly language generated by the compiled C code, because the instruction pipeline cache needs to be discarded less often.

Let's examine some simple methods that execute loops in different ways. You can look at the assembly language to spot when a loop has been unrolled so that several loop body operations can be executed within a single loop iteration.

Before diving into the assembly language, we should note that the previous Java code needs to be slightly modified for JIT compilation to take effect, because the HotSpot VM compiles only whole methods. Not only that, but methods are not compiled until they have been executed in interpreted mode a certain number of times (typically 10,000 times for fully optimized compilation) before the compiler considers them. Using only a single `main()` exactly as shown would mean that JIT compilation would never be invoked and the optimization would not be performed.

A Java method that is essentially equivalent to the earlier example and that you can use for benchmarking is

```
private long intStride1()
{
    long sum = 0;
    for (int i = 0; i < MAX; i += 1)
    {
        sum += data[i];
    }
    return sum;
}
```

The example method performs summation of data fetched sequentially from an array, and then it returns the total. This is similar to earlier examples, but we chose to return the total to ensure that the JIT compiler does not combine loop unrolling with escape analysis to optimize even further, which would obscure the effect of unrolling.

You can spot a key access pattern in the assembly language that helps you understand what's going on. It shows up as a triple consisting of `[base, index, offset]` made up of registers and offsets, where

- `base register` contains the start address of data in the array
- `index register` contains the loop counter (which gets multiplied by the data type `size`)
- `offset` is used for offsetting each unrolled access

The actual assembly language will look something like this:

```
add rbx, QWORD PTR [base register + index register
```

Considering an array of type `long`, let's look at the conditions under which the loop will be unrolled. Note that loop unrolling behavior can vary between HotSpot VM versions and is dependent on the details of the CPU architecture, but the overall concept remains the same.

To get the disassembled native code produced by the JIT compilers, you will need a disassembly library (the standard choice is hsdis, the HotSpot Disassembler), which you need to install in the `jre/lib` directory of your Java installation.

hsdis can be built from the OpenJDK source code, and instructions for doing so can be found in the JITWatch wiki. Alternatively, Oracle's GraalVM project ships hsdis as part of the downloadable binaries—and the file can simply be copied from the GraalVM installation into the main Java installation location.

Once you have installed hsdis, you need to instruct the VM to output the method's assembly language. To achieve this, you need to add some additional VM switches, including `-XX:+PrintAssembly`.

Note that native code disassembly into readable assembly language is performed directly after the JIT thread emits the compiled method. It is an expensive operation that can affect the performance of your program and should not be used on production processes.

To see the disassembly in action, execute the program with the following VM switches, which output the assembly language for just the named method:

```
java -XX:+UnlockDiagnosticVMOptions \
     -XX:-UseCompressedOops         \
     -XX:PrintAssemblyOptions=intel \
     -XX:CompileCommand=print,javamag.lu.LoopUnroll
     javamag.lu.LoopUnrolling
```

This command produces the corresponding assembly language for an `int` loop counter with a constant stride of 1.

Note that we use `-XX:-UseCompressedOops` here only to simplify the assembly language output by switching off the arithmetic for pointer address compression. This saves some memory usage in a 64-bit JVM, but we don't recommend you do this in normal VM use. You can learn all about compressed ordinary object pointers (oops) in the OpenJDK wiki.

The accumulating `long` sum is stored in the 64-bit register `rbx`. Each `add` instruction loads the next value from the `data` array and adds it to `rbx`. The constant offset into the array increases by 8 bytes (which is the size of a Java `long` primitive) with each load.

When the unrolled section branches back to the `main` loop start, the offset register will be incremented by the amount of data processed in this loop iteration:

```
//==============================
// SETUP CODE
//==============================

// MOVE ADDRESS OF data ARRAY INTO rcx
0x00007f475d1109f7: mov rcx,QWORD PTR [rbp+0x18]  ;

// MOVE SIZE OF data ARRAY INTO edx
0x00007f475d1109fb: mov edx,DWORD PTR [rcx+0x10]

// MOVE MAX INTO r8d
0x00007f475d1109fe: mov r8d,DWORD PTR [rbp+0x10]  ;

// LOOP COUNTER IN r13d, COMPARE WITH MAX
0x00007f475d110a02: cmp r13d,r8d

// JUMP TO EXIT IF COUNTER >= MAX
0x00007f475d110a05: jge L0006

0x00007f475d110a0b: mov r11d,r13d
0x00007f475d110a0e: inc r11d
0x00007f475d110a11: xor r9d,r9d
0x00007f475d110a14: cmp r11d,r9d
0x00007f475d110a17: cmovl r11d,r9d
0x00007f475d110a1b: cmp r11d,r8d
0x00007f475d110a1e: cmovg r11d,r8d

//==============================
// PRE-LOOP
//==============================

// ARRAY BOUNDS CHECK
            L0000: cmp r13d,edx
0x00007f475d110a25: jae L0007

// PERFORM A SINGLE ADDITION
0x00007f475d110a2b: add rbx,QWORD PTR [rcx+r13*8+0x

// INCREMENT THE LOOP COUNTER
0x00007f475d110a30: mov r9d,r13d
0x00007f475d110a33: inc r9d  ;*iinc

// JUMP TO MAIN LOOP IF FINISHED PRE-LOOP
0x00007f475d110a36: cmp r9d,r11d
0x00007f475d110a39: jge L0001

// CHECK LOOP COUNTER AND BACK BRANCH IF NOT FINISH
0x00007f475d110a3b: mov r13d,r9d
0x00007f475d110a3e: jmp L0000

//==============================
// MAIN LOOP SETUP
//==============================
            L0001: cmp r8d,edx
0x00007f475d110a43: mov r10d,r8d
0x00007f475d110a46: cmovg r10d,edx
0x00007f475d110a4a: mov esi,r10d
0x00007f475d110a4d: add esi,0xfffffff9
0x00007f475d110a50: mov edi,0x80000000
0x00007f475d110a55: cmp r10d,esi
0x00007f475d110a58: cmovl esi,edi
0x00007f475d110a5b: cmp r9d,esi
0x00007f475d110a5e: jge L000a
0x00007f475d110a64: jmp L0003
0x00007f475d110a66: data16 nop WORD PTR [rax+rax*1+(

//==============================
// MAIN LOOP START (UNROLLED SECTION)
// PERFORMS 8 ADDITIONS PER LOOP ITERATION
//==============================
            L0002: mov r9d,r13d
            L0003: add rbx,QWORD PTR [rcx+r9*8+0x1
0x00007f475d110a78: movsxd r10,r9d
0x00007f475d110a7b: add rbx,QWORD PTR [rcx+r10*8+0x
0x00007f475d110a80: add rbx,QWORD PTR [rcx+r10*8+0x
0x00007f475d110a85: add rbx,QWORD PTR [rcx+r10*8+0x
0x00007f475d110a8a: add rbx,QWORD PTR [rcx+r10*8+0x
0x00007f475d110a8f: add rbx,QWORD PTR [rcx+r10*8+0x
```

```
0x00007f475d110a94: add rbx,QWORD PTR [rcx+r10*8+0x
0x00007f475d110a99: add rbx,QWORD PTR [rcx+r10*8+0x

// INCREMENT LOOP COUNTER BY 8
0x00007f475d110a9e: mov r13d,r9d
0x00007f475d110aa1: add r13d,0x8  ;*iinc

// CHECK LOOP COUNTER AND BACK BRANCH IF NOT FINISHE
0x00007f475d110aa5: cmp r13d,esi
0x00007f475d110aa8: jl L0002
//==============================

0x00007f475d110aaa: add r9d,0x7  ;*iinc

// IF LOOP COUNTER >= MAX JUMP TO EXIT
             L0004: cmp r13d,r8d
0x00007f475d110ab1: jge L0009
0x00007f475d110ab3: nop

//==============================
// POST-LOOP
//==============================

// ARRAY BOUNDS CHECK
             L0005: cmp r13d,edx
0x00007f475d110ab7: jae L0007

// PERFORM A SINGLE ADDITION
0x00007f475d110ab9: add rbx,QWORD PTR [rcx+r13*8+0x

// INCREMENT THE LOOP COUNTER
0x00007f475d110abe: inc r13d  ;*iinc

// CHECK LOOP COUNTER AND BACK BRANCH IF NOT FINISHE
0x00007f475d110ac1: cmp r13d,r8d
0x00007f475d110ac4: jl L0005
//==============================
```

(To make things easier, we've included some comments in the assembly code so that the separate sections are clear. For brevity, we show only one exit block, but usually there will be multiple exit blocks in the assembly language to handle the different ways the method can end. The setup section is included for comparison to other operations later in this article.)

When the loop accesses the array, HotSpot VM eliminates array bounds checks by splitting the loop into three sections:

In Java 10, a more advanced technique called loop strip mining was introduced to further balance the effects of safepoints on throughput and latency.

- Pre-loop: This performs initial iterations with bounds checking.
- Main loop: The loop stride (the amount the loop counter is increased on each iteration) is used to calculate the maximum number of iterations that can be performed without requiring a bounds check.
- Post-loop: This performs the remaining iterations with bounds checking.

You can see the practical effect of this approach by looking at the ratio of add operations to jumps. In the un-optimized C case we examined earlier, this ratio was 1:1, but the Java HotSpot VM's JIT compiler has increased this ratio to 8:1, reducing the number of jumps by 87% for this section. Because the effect of a jump is typically to consume from 2 to 300 cycles waiting for a refill of code from main memory, this improvement is potentially significant. (To learn more about how the HotSpot VM eliminates bounds checks when iterating loop-invariant arrays, see the online documentation.)

The HotSpot VM can also unroll loops with an `int` counter and a regular stride of 2 or 4. For example, with a stride of 4, the body is unrolled 8 times and the address offset increases by 0x20 (32) bytes for each

access. The compiler can also unroll loops with a counter of type `short`, `byte`, or `char`, but not of type `long`, as we explain in the next section.

Safepoints

The Java code for a method with a `long` loop counter seems very similar to the `int` case:

```
private long longStride1()
{
    long sum = 0;
    for (long l = 0; l < MAX; l++)
    {
        sum += data[(int) l];
    }
    return sum;
}
```

However, with the loop counter of type `long`, the assembly language produced is completely different from the setup section in the previous assembly language listing—no loop unrolling occurs even with a constant stride of 1:

```
// ARRAY LENGTH INTO r9d
  0x00007fefb0a4bb7b: mov    r9d,DWORD PTR [r11+0x1

  // JUMP TO END OF LOOP TO CHECK COUNTER AGAINST L
  0x00007fefb0a4bb7f: jmp    0x00007fefb0a4bb90

  // BACK BRANCH TARGET - SUM ACCUMULATES IN r14
  0x00007fefb0a4bb81: add    r14,QWORD PTR [r11+r10

  // INCREMENT LOOP COUNTER IN rbx
  0x00007fefb0a4bb86: add    rbx,0x1

  // SAFEPOINT POLL
  0x00007fefb0a4bb8a: test   DWORD PTR [rip+0x9f394

  // IF LOOP COUNTER >= 1_000_000 THEN JUMP TO EXIT
  0x00007fefb0a4bb90: cmp    rbx,0xf4240
  0x00007fefb0a4bb97: jge    0x00007fefb0a4bbc9

  // MOVE LOW 32 BITS OF LOOP COUNTER INTO r10d
  0x00007fefb0a4bb99: mov    r10d,ebx

  // ARRAY BOUNDS CHECK AND BRANCH BACK TO LOOP STA
  0x00007fefb0a4bb9c: cmp    r10d,r9d
  0x00007fefb0a4bb9f: jb     0x00007fefb0a4bb81
```

There is now only one add instruction per loop body iteration—the ratio of add to jump instructions is back to 1:1, and the benefit of loop unrolling has disappeared. Not only that, but a *safepoint* poll has been added to the loop.

A safepoint is a place in code at which the executing thread knows that it has completed all modifications to internal data structures (such as objects in the heap). It is an ideal time to check and see whether the JVM needs to halt all threads executing Java code. By checking at safepoints and safely suspending execution, application threads provide an opportunity for the JVM to perform operations that might change memory layout and modify internal data structures, such as stop-the-world (STW) garbage collection.

In the case of interpreted code, a very natural location for safepoint checks already exists: after a bytecode has finished executing and just before the next bytecode is executed.

The “in-between bytecodes” safepoint check for interpreted code is very useful, but in the case of JIT-compiled methods, additional checks must be synthesized and inserted into the code emitted by the compiler.

Without these checks, a thread could continue to run while other threads had already stopped at their safepoints. This could lead to a pathological VM state in which almost all application threads are paused but some continue to run for a substantial amount of time.

As a virtual machine, Java HotSpot VM has advanced loop unrolling capabilities to reduce or remove the overhead of back branches.

HotSpot has several heuristics for inserting a safepoint check into compiled code. The two most common are just before a back branch (as in this case), and just after a method has exited and before control returns to the caller.

However, the appearance of the safepoint check in the example of a `long` counter also points out another feature of the `int` counted loops: They do not contain safepoint checks. This means that the entirety of an `int` counted loop (with constant stride) will run without encountering any safepoint checks, which may be a considerable length of time in extreme cases.

However, consider a loop with an `int` counter and a stride that is not constant, for example one where the stride can be different on each method invocation:

```
private long intStrideVariable(int stride)
{
    long sum = 0;
    for (int i = 0; i < MAX; i += stride)
    {
        sum += data[i];
    }
    return sum;
}
```

This code will indeed force the JIT compiler to emit a safepoint check on each back branch.

If you are concerned about latency pauses introduced by long-running counted `int` loops holding other threads at a safepoint until the loop completes, you can use the VM switch `-XX:+UseCountedLoopSafepoints`. This option adds a safepoint check before the back branch of the unrolled loop. So, in the long assembly code listing, the test would occur every eight additions.

As with every performance-related command-line switch, you should not activate it until you have proved in a performance test that it will provide a significant benefit. Very few applications will see any benefit from activating this switch, so it should not be switched on blindly. In Java 10, a more advanced technique called *loop strip mining* was introduced to further balance the effects of safepoints on throughput and latency.

Let's conclude by looking at a JMH benchmark to compare the performance of iterating the same array using either an `int` counter or a `long` counter. As we explained earlier, the body of a loop with a `long` counter will not be unrolled, and the loop will also contain a safepoint poll.

```
package optjava.jmh;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
@State(Scope.Thread)
public class LoopUnrollingCounter
{
    private static final int MAX = 1_000_000;

    private long[] data = new long[MAX];
```

```
    @Setup
    public void createData()
    {
        java.util.Random random = new java.util.Ran

        for (int i = 0; i < MAX; i++)
        {
            data[i] = random.nextLong();
        }
    }

    @Benchmark
    public long intStride1()
    {
        long sum = 0;
        for (int i = 0; i < MAX; i++)
        {
            sum += data[i];
        }
        return sum;
    }

    @Benchmark
    public long longStride1()
    {
        long sum = 0;
        for (long l = 0; l < MAX; l++)
        {
            sum += data[(int) l];
        }
        return sum;
    }
}
```

The output shows the following:

```
Benchmark                            Mode  Cnt     Sc
LoopUnrollingCounter.intStride1     thrpt  200  2423.
LoopUnrollingCounter.longStride1    thrpt  200  1469.
```

This means that the loop with the `int` counter performs nearly 64% more operations per second.

Conclusion

The HotSpot VM can perform more-complex loop unrolling optimizations—for example, on a loop containing multiple exit points. In this case, the loop is unrolled, and each unrolled iteration contains a test for the exit condition.

As a virtual machine, the HotSpot VM has advanced loop unrolling capabilities to reduce or remove the overhead of back branches. However, the majority of Java programmers do not need to know about this capability—it's just one more transparent performance optimization that the runtime provides.

Also in This Issue

Chris Newland

Chris Newland (@chriswhocodes) is a Java Champion. He invented and still leads developers on the JITWatch project, an open source log

analyzer for visualizing and inspecting just-in-time compilation decisions made by the HotSpot JVM.

Ben Evans

Ben Evans (@kittylyst) is a Java Champion, a tech fellow and founder at jClarity, an organizer for the London Java Community (LJC), and a member of the Java SE/EE Executive Committee. He has written four books on programming, including the recent *Optimizing Java* (O'Reilly).

Share this Page