IBM Just-In-Time Compiler (JIT) for Java
Best practices and coding guidelines for improving performance

Pramod Ramarao
Joran Siu
Pavan Pamula

Contributors: Vijay Sundaresan, Patrick Doyle, Graham Chapman

IBM Software Group
IBM Systems & Technology Group

November 2008

© Copyright IBM Corporation, 2008.
# Table of contents

Abstract ........................................................................................................................................ 1  
Introduction ................................................................................................................................. 1  
**JIT technology** .......................................................................................................................... 2  
  
  - Compilation happens during run time ............................................................................................ 2  
    - Strategy: Focus optimization effort on important code ................................................................. 2  
  
  - Inlining ........................................................................................................................................ 3  
    - Strategy: Aggressively inline method calls .................................................................................. 3  
  
  - Virtual and interface methods ....................................................................................................... 3  
    - Strategy: Devirtualize as many call sites as possible ................................................................. 3  
  
  - Heap allocations .......................................................................................................................... 4  
    - Strategy: Optimize for object locality ......................................................................................... 4  
  
**Java coding guidelines** .............................................................................................................. 5  
  
  - Object allocations ......................................................................................................................... 5  
    - Guideline: Avoid creating objects inside loops ........................................................................... 5  
    - Guideline: Minimize object allocations, if possible ................................................................... 7  
    - Guideline: Use immutable fields ............................................................................................... 8  
  
  - Methods ...................................................................................................................................... 8  
    - Guideline: Keep methods small .................................................................................................. 8  
    - Guideline: Use exceptions and reflection rarely ......................................................................... 9  
  
  - Loops ....................................................................................................................................... 10  
    - Guideline: Do not modify the loop bounds within the loop body ................................................. 10  
    - Guideline: Increment the loop index by a single value across all paths ...................................... 12  
    - Guideline: Make loops as compact as possible ......................................................................... 13  
    - Guideline: Use locals instead of fields or static variables, where possible .............................. 13  
    - Guideline: System.arraycopy() is well-optimized by the JIT .................................................... 13  
    - Guideline: TRJIT can also optimize special loops such as memset and array translate ................ 14  
  
**Synchronization** ....................................................................................................................... 15  
  
**Tuning** ..................................................................................................................................... 16  
  
  - Tuning application code ............................................................................................................... 16  
  
  - Tuning JVM parameters ............................................................................................................... 16  
  
  - Tuning the garbage collector ...................................................................................................... 16  
  
  - Tuning JVM parameters for application startup ......................................................................... 17  
  
  - Diagnostic techniques and tools available for performance problems ..................................... 18  
  
**Summary** .................................................................................................................................. 18  
**Footnotes** ................................................................................................................................ 19  
**Resources** ................................................................................................................................ 20  
**About the authors** ..................................................................................................................... 21  
**Trademarks and special notices** ................................................................................................. 22
Abstract

This paper describes some guidelines for writing Java applications that use the IBM Java Development Kit 6.0, based on analyzing the performance of various applications. These guidelines are not strict rules, but serve to act as an aid for developers who need help in obtaining good performance from their applications.

Introduction

The Java programming language is designed to facilitate software development by providing features such as exception checks, dynamic class loading, interface and virtual dispatch and a framework for automatic garbage collection. While these features originally imposed a significant performance overhead challenge for Java virtual machines, over time, optimization techniques have been developed to help reduce the cost of using these features significantly.

One of the major features of the IBM® J9 Java™ virtual machine (JVM) is an optimizing just-in-time (JIT) compiler (called TRJIT) [21], which is designed to deliver high performance for different kinds of Java applications. Developing Java software that is amenable to optimization by the JIT compiler is critical to obtaining high performance.

Some of the features of the Java programming language that have just been outlined introduce inherent uncertainties inadvertently. Uncertainties in application code make the optimizations of such code difficult and this, in turn, makes the application run slower. However, complicating the Java code for the sake of the JIT compiler is something to generally avoid; you should only do so when absolutely necessary. Optimizing Java code by making the code simpler is usually a good idea, but there are many reasons not to complicate code to make it faster. Firstly, complicated code usually has more bugs and is harder to maintain. Secondly, optimizing Java code for the current virtual machine, JIT technology and execution environment (which includes the hardware and operating system) might make things worse for future technology and run environments.

This paper outlines some guidelines for writing high-performance Java code that is amenable for optimization by the TRJIT compiler. You should apply these guidelines to the important parts of your application code (to the hotspots), as determined by using profiling tools, to obtain the best performance.

This paper is organized as follows: The first section contains a background of the JIT technology. It also provides a brief overview of the challenges faced and the approaches taken by the TRJIT compiler to optimize Java code. The next section outlines some suggestions for writing Java applications from the perspective of the TRJIT compiler. Finally, this paper briefly touches upon tuning application code.
JIT technology

This section sketches some of the challenges of optimizing Java applications and strategies that the TRJIT compiler employs to optimize such applications.

Compilation happens during run time

Unlike static applications, a JIT compiler dynamically compiles Java methods during the running of the application. Although dynamic compilation carries a runtime overhead, it also allows the JIT compiler to collect runtime information and optimize the application for the specific target platform.

Strategy: Focus optimization effort on important code

Given the runtime overhead, the JIT focuses its optimization efforts on the most frequently invoked parts of the application to maximize the application’s performance. To determine which methods are important, a sampling thread periodically checks which method the virtual machine is currently running on each Java thread. The TRJIT compiler chooses a method for compilation only if it is invoked frequently, or if the sampling thread indicates that the method has been running for a long time.

The TRJIT compiler employs an adaptive-compilation strategy. The first compilation of a method is usually done at a low optimization level, resulting in a cheap compile (in terms of time and computational resources). The newly compiled code runs when the method is next invoked. Some methods that are long-running and that are only invoked one time (for example, the main() method) are optimized by transitioning them from interpreted to compiled code through on-stack replacement [7]. A recompilation of a method is triggered if the sampling thread indicates that the method runs frequently. This strategy guarantees that the JIT compiler focuses on the most important parts of the application code. Recompilations are usually performed at higher optimization levels. Methods that run for more than roughly 10 percent of the total run time compile at the highest optimization level, though this is not guaranteed.

The TRJIT compiler employs aggressive-feedback-directed profiling to assist compilations that are performed at higher optimization levels [1]. When a method is chosen for compilation at higher optimization levels, a temporary instrumented compilation is first done. The instrumented method runs for a short duration to collect profiling information, which is in turn used to optimize the method. The virtual machine also performs profiling during bytecode interpretation; the JIT compiler then uses this information (in addition to the profiling information described earlier) to optimize the method.
**Inlining**

One of the strengths of Java and other object-orientated programming languages is their ability to modularize and reuse code in the form of packages, classes and methods.

**Strategy: Aggressively inline method calls**

Currently, the unit of compilation that the JIT compiler considers is a single method. To make optimizations more effective, the JIT compiler aggregates small methods into larger ones through effective inlining. Inlining is a technique where a method call is replaced with the target method’s bytecodes. This technique increases the scope of compilation beyond a single method. Inlining is also a powerful way to propagate information into target methods, thereby reducing uncertainty in the application code and making optimizations more effective.

**Virtual and interface methods**

Virtual and interface methods in Java provide a useful abstraction for developers. However, this abstraction comes with an associated performance cost, because polymorphic calls might have multiple target implementations during the execution of the application. Furthermore, the uncertainty of the target method at these call sites can limit the optimizations that the JIT compiler performs, such as inlining.

**Strategy: Devirtualize as many call sites as possible**

However, the TRJIT compiler tries to overcome such constraints by using devirtualization techniques, whereby a single-target implementation is chosen and the JIT transforms the virtual call to a direct call. The JIT can use profiling or class hierarchy analysis [5] to detect that the receiver of a virtual call is predominantly of a certain type. The transformation needs to be done with a virtual-guard test in place to ensure that the correct target method is invoked in all cases for functional correctness. Figure 1 shows an example of this transformation.

```java
o = receiver object;
x = receiver class (o);
if (x == expected-class) {  // virtual guard
    x.foo(a, b, c);    // direct call can be inlined
} else {
    o.foo(a, b, c);    // guard failed, virtual call
}
```

*Figure 1: Example of Devirtualization*

Certain guards can be removed if they are proven to be unnecessary, based on the current class hierarchy during the execution of the application. If a new class is loaded during the execution of the application that changes the hierarchy of the class involved in a guard, then the TRJIT reinserts the guard to maintain correct behavior.
Heap allocations

Java allocates objects on the heap. Therefore, references to objects might need to go through multiple levels of the memory hierarchy, which includes the processor cache and main memory. Data locality is important because hardware cache misses can lead to poor application performance. An application that allocates many objects with a short lifespan can have reduced performance because of potential heap fragmentation and poor locality, resulting in more hardware-cache misses. One of the ways in which the TRJIT compiler helps data locality (and in turn, performance) is by converting heap allocations into stack allocations when possible [3].

Strategy: Optimize for object locality

Using the stack for short-lived allocations leads to better locality because the stack is usually available in the cache. This reduces memory latencies that result from cache misses. As an added benefit, using the stack for short-lived allocations reduces the load on the garbage collector because these allocations are no longer present in the heap.

The TRJIT compiler attempts to find allocations that never escape the method in which they are allocated. See the example shown in Figure 2, where the allocation of the Integer object, `intObj`, is local to method `m1` and does not escape the method. The TRJIT compiler can convert this allocation into a stack-allocation.

```java
public class myClass {
    public static void m1() {
        Integer intObj = new Integer(100); //intObj does not escape m1
        System.out.println("Value is: "+ intObj.intValue());
    }
}
```

Figure 2. `intObj` does not escape `m1` and can be stack-allocated

However, in Figure 3, the allocation is stored into a field of another object and is, therefore, not local to the method `m1`.

```java
public class myClass {
    public static void m1(myClass myObj) {
        Integer intObj = new Integer(100);
        myObj._myField = intObj; // intObj escapes m1
        System.out.println("Value is: "+ intObj.intValue());
    }
    public Integer _myField;
}
```

Figure 3. `intObj` is stored into the field of another object, thus escaping, and cannot be stack-allocated
Java coding guidelines

This section presents a set of Java coding guidelines that are based on analyzing the performance of various Java applications. The focus is on four areas with the most impact on application performance.

Object allocations

Object allocations are not cheap in Java because, after memory is allocated for an object, the object needs to be initialized before it is accessible to the program. This involves zero-initializing the object fields as well as invoking the constructor to perform initializations as specified by the user.

Guideline: Avoid creating objects inside loops

Objects that are allocated inside loops have a short lifespan and might not survive subsequent iterations of the loop. For example, you can avoid creating objects inside loops to improve performance of String concatenation operations. Although it is not obvious, the javac compiler implicitly allocates a java/lang/StringBuilder object when you use the plus (+) concatenation operator. It is more efficient to explicitly use java/lang/StringBuilder to build a string than to use the + concatenation operator.

Figure 4, Figure 5 and Figure 6 show an example of this.

In Java, strings are immutable. Therefore, in Figure 4, the character x is not appended to the string that str points to. The concatenation + operator implicitly creates a StringBuilder object and copies the string (which str points to) into this new object. It then appends the character x to the contents of the new object.

```java
public class myClass {
    public static int N = 10000;
    public static void main(String [] argv) {
        String str = "";
        for (int i = 0; i < N; i++)
            str = str + "x";
    }
}
```

Figure 4. Example of avoiding unnecessary allocations: New objects implicitly created inside loop on every iteration

Figure 5 shows the code that javac generates.

```java
public class myClass {
    public static int N = 10000;
    public static void main(String [] argv) {
        String str = "";
        for (int i = 0; i < N; i++)
            StringBuilder sb = new StringBuilder();
            sb.append(str);
            sb.append("x");
            str = sb.toString();
    }
}
```

Figure 5. Example of avoiding unnecessary allocations: Code generated by javac for loop in Figure 3a
However, in Figure 6, the character x is directly appended to the StringBuilder object that str points to, and no objects are created.

```java
public class myClass {
    public static int N = 10000;
    public static void main(String[] argv) {
        StringBuilder str = new StringBuilder();
        for (int i = 0; i < N; i++)
            str.append("x");
    }
}
```

**Figure 6. Example of avoiding unnecessary allocations: More efficient version of Figure 3a, no new objects created**

Figure 7 shows the performance of the examples shown in Figure 4, Figure 5 and Figure 6. As you can see, the performance improvements are significant when allocations are avoided inside loops. All performance measurements that are mentioned in this paper were obtained on an IBM System x™ machine using Intel® Xeon™ Quad Core system running at 3.2 GHz with 8 GB of RAM.

Here are a few guidelines to improve the locality of the Java application:

- Carefully design the application’s class hierarchy. It is better to place less data in the super classes, unless all the subclasses in the hierarchy access the data. This greatly reduces the size of the objects.
- Move rarely used field members into separate classes. Although this can increase the memory footprint of the Java application, this separation has benefits in the form of reduced cache misses.
- Try to avoid the unnecessary creation of objects. Instead, you should design the application so that it reuses objects when possible. If the application is multithreaded, you can use thread-local data for temporary buffers. In fact, use the ThreadLocal class wherever possible.
Guideline: Minimize object allocations, if possible

Minimizing the number of objects that an application allocates also improves the locality of the application. Objects reside more often in the lower levels of the memory hierarchy (cache), which results in better application performance. However, you should avoid the allocation of a large number of short-lived objects, as this tends to result in poor cache locality of the application. If you need to create short-lived objects, then the recommendation is that you use the IBM JDK’s generational garbage collection policy (-Xgcpolicy:gencon) [10].

As explained in the previous section, the TRJIT compiler tries to aggressively convert heap allocations to stack allocations through an optimization called escape analysis [3]. When possible, you can follow a few guidelines when writing Java application code to facilitate the stack allocation of objects. In the following cases, the TRJIT compiler cannot convert a heap allocation into a stack allocation:

- An object escapes if it is stored into a static field or a field of another object.
- If the object allocation is passed as a parameter into a method call that cannot be inlined.
- If the allocation is returned from a method call that cannot be inlined.
At lower optimization levels, the TRJIT compiler does not perform this optimization. Reducing the number of allocations also reduces the load on the J9 JVM garbage collector. There are fewer objects on the heap, which results in fewer garbage collections, in turn reducing the pause times of the application. A last point to be noted here is about memory leaks. To avoid memory leaks in the application, release any references to obsolete objects (by storing a NULL into such references). References from static fields or objects with a long life time are not garbage collected until the application ends.

**Guideline: Use immutable fields**

The TRJIT compiler aggressively optimizes immutable fields. You can follow these guidelines to make fields immutable:

- Declare fields as `final` when possible.
- Declare fields as `private` and write to such fields only in constructors.

When an object is declared as `final static`, the TRJIT compiler can apply optimizations based on an analysis of the contents of the object on the heap. For example, the TRJIT compiler can determine the length of an array, which can help optimize loops that iterate over the array. *(Note: Java has no way to declare multidimensional arrays to be final — only the first dimension of a final array is fixed.)*

**Methods**

As explained in the previous section, the TRJIT compiler attempts to focus optimization efforts on the most frequently run parts of the application. Therefore, a method that is invoked frequently has a much greater chance of being optimized aggressively, as compared to a method that is only invoked a few times. It is also beneficial to structure the Java application so that most frequently run methods *(hot)* invoke other frequently run methods, rather than infrequently run *(cold)* methods. This helps the TRJIT compiler to aggressively inline method calls within hot methods.

**Guideline: Keep methods small**

From a Java development perspective, it is advantageous to keep methods small (about 10 lines of Java code), as this allows the TRJIT compiler to inline callee methods into their callers. Some heuristics are outlined in this paper because they cause a call more likely to be inlined by the TRJIT compiler:

- **Calls in loops**: The inlining of method calls within loops is likely to benefit performance.
- **Specialized calls**: Calls that have constant arguments and arguments with a specific type (rather than a generic type) are more likely to be inlined.

You should declare parameters to method calls to be of a final or leaf type in the class hierarchy of the Java application. This allows the TRJIT compiler to optimize virtual calls on such parameters by devirtualizing, without using a virtual guard. If parameters are declared to be `Object` or an abstract or interface type, then the compiler might not be able to optimize virtual calls as effectively. Because devirtualization is important for optimizations of the Java code, keep the class hierarchy as simple as possible to allow the TRJIT compiler to devirtualize as many call sites as possible.
When the Java application is highly polymorphic (complex class hierarchy), then it is more efficient to use abstract types, rather than interface types, as interface calls are more expensive. For interface calls with few implementations, the TRJIT compiler attempts to use an if-then-else chain (called Polymorphic-Inline-Cache) [8] to compare the receiver types for faster interface dispatch. Figure 8 shows an example.

```
o = receiver object;
x = receiver class (o);
if ( x == expected-class-1 )
    x.foo(a, b, c); // implementation of foo from class-1
else if ( x == expected-class-2)
    x.foo(a, b, c); // implementation of foo from class-2
else if ( x == expected-class-3)
    x.foo(a, b, c); // implementation of foo from class-3
else
    o.foo(a, b, c); // checks failed, interface call
```

Figure 8. Example of Polymorphic Inline Cache

**Guideline: Use exceptions and reflection rarely**

To provide the best possible performance of the application in the common cases, the VM and the JIT optimize the non-exception paths of the application in favor of the exception paths. Throwing an exception is expensive because of the associated runtime overhead. Therefore, exceptions should not be used for normal flow-of-control in a Java application.

The Java reflection API provides a simple interface for you to obtain information about various classes, methods and objects of the application when it runs in the JVM. However, reflection introduces additional levels of abstraction and indirection that affects application performance. You should not use reflection in the core of a performance-critical application. Figure 9 and Figure 10 show an example and the performance cost that is associated with using reflections.
public int invokeDirect() {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        sum = increment(sum);
    }
    return sum;
}

public int invokeReflect() {
    try {
        Class c = Class.forName("reflectTest");
        Method m = c.getDeclaredMethod("increment", new Class[]{int.class});
        // create the args array
        Object[] args = new Object[1];
        // init the sum
        Object sum = new Integer(0);
        for (int i = 0; i < N; i++) {
            args[0] = sum;
            sum = m.invoke(this, args);
        }
        return ((Integer)sum).intValue();
    } catch (Exception e) {
        System.out.println(e);
    }
    return 0;
}

Figure 9: Example of using reflection

Loops

Loops are an integral part of the Java programming language. For many programs, a good majority of the application time is often spent executing loops. As a result, TRJIT employs many optimization techniques that are intended to aggressively improve loop performance in Java. You can assist the TRJIT compiler by using the following guidelines to ensure that their loops are well-behaved.

Guideline: Do not modify the loop bounds within the loop body

When the loop bounds are independent of the loop body, the TRJIT can usually profile the bounds and create specialized, more optimized versions of the loop. Furthermore, the ability to predict and compensate for loop exits is important for many fundamental loop optimization techniques, such as loop versioning and loop unrolling. If the loop bounds were modified within the loop body, such loop-optimization techniques are typically not applicable, which results in poorer performance.
Figure 10: Performance of Reflection example in Figure 9

Figure 11 shows an example of a poorly behaved loop with a loop bound ‘j’ that is modified within the loop.

```java
public void badLoopBounds() {
    int i = 0;
    while (i < j) {
        ...
        j = newLoopBounds();  // Modifying of loop bounds
        i++;
    }
}
```

Figure 11. Example of a poorly behaved loop bound - The loop bounds: j is modified within the loop by using the newLoopBounds() method.
Figure 12 shows a well-behaved version, where the loop bound is invariant. The JIT will be able to optimize the latter loop better.

```java
public void goodLoopBounds() {
    int i = 0;
    while (i < N) { // Loop bounds N is never modified
        ...
        i++;
    }
}
```

**Figure 12. Example of a well-behaved loop bound - The loop bound: N is never modified within the loop**

**Guideline: Increment the loop index by a single value across all paths**

Similar to the previous guideline, many loop-optimization techniques depend on well-behaved loop iterations and the prediction of the number of iterations a loop will run. If two paths in the loop body increment the loop index by different amounts, many TRJIT loop-optimization techniques cannot be applied to improve the loop’s performance.

Figure 13 shows a loop that increments the loop index by 1 or 2, depending on the condition of the nested if-statement. For the JIT compiler to optimize the loop, it must prove certain properties of the condition expression (for example, whether the condition expression is loop invariant), which might not be possible.

```java
public void badLoopIndexIncrements() {
    for (int i = 0; i < 100; i++) { // Loop Index may be incremented by 1 or 2
        if (condition)
            i++;
        else
            i = i + 2;
    }
}
```

**Figure 13. Example of poorly behaved loop-indices increments - The loop index: i will increment by 1 or 2, depending on the condition**

Figure 14 shows a well-behaved loop that only increments the loop index by a single value. The JIT recognizes this property and optimizes the loop accordingly.

```java
public void goodLoopIndexIncrement() {
    for (int i = 0; i < 100; i++) { // Loop index incremented by 1 all the time
        i++
    }
}
```

**Figure 14. Example of well-behaved loop-indices increments - The loop index: i will increment by the same value on each iteration**
Guideline: Make loops as compact as possible

Compact loops tend to run faster and are easier to optimize. If possible, remove any code that is loop-independent, or that rarely runs, from the loop body. Although the TRJIT compiler tries to determine loop-invariant code and extract such code from loop bodies, proving such loop-invariant properties can sometimes be difficult, because of virtual calls and dynamic class loading.

Guideline: Use locals instead of fields or static variables, where possible

If a field or static variable is accessed within a loop body, consider using local variables to hold the contents of the field or static variable for the duration of the loop. A JIT compiler might not optimize memory access to a field or static variable out of the loop, because such accesses can be visible outside the scope of the current method (for example, callee methods and other threads).

Figure 15 shows a loop with an assignment to a static variable, `myStaticX`, inside the loop. The JIT compiler might not be able to optimize the assignment out of the loop.

```
public static int myStaticX;   // Static variable myStaticX;
public void methodBad() {
    for (int i = 0; i < N; i++) {
        // Static assignment
        myStaticX = i;
    }
}
```

*Figure 15. Example of a poorly behaved loop with static assignment - The assignment to myStaticX might not be removed from the loop body*

In contrast, Figure 16 shows a well-behaved loop without any field or static variable accesses. The static assignment to `myStaticX` occurs outside the loop body.

```
public static int myStaticX;   // Static variable myStaticX;
public void methodGood() {
    int temp = 0;
    for (int i = 0; i < N; i++) {
        temp = i;
    }
    myStaticX = temp;    // Static assignment is outside of loop.
}
```

*Figure 16. Example of a well-behaved loop with no static or field assignments - The assignment to myStaticX is outside the loop body*

Guideline: `System.arraycopy()` is well-optimized by the JIT

When copying the elements of one array to another, consider using the `System.arraycopy()` function that is defined in the Java API, instead of writing your own loop or routine. TRJIT recognizes the
System.arraycopy() function and generates highly optimized platform-specific code to perform the array-copy operations.

**Guideline: TRJIT can also optimize special loops such as memset and array translate**

Certain hardware platforms offer vector or other complex instructions that you can use to improve the performance of array operations. An example of this are the translate instructions that are offered on the IBM System z® platform; the translate instructions take bytes from an input array, find the corresponding output bytes in a lookup table, and insert them into an output array. All operations are performed in hardware, yielding significant performance benefits.

TRJIT has an optimization to recognize special loops and to transform them to take advantage of these special hardware instructions [18]. The examples in Figure 17, Figure 18 and Figure 19 show the basic idioms for memset, array translate and search string. If any potential loops are coded to similarly match the idioms, then TRJIT tries to transform such loop candidates to exploit the hardware instructions.

```java
// Initializes an array to a particular value.
public void memset() {
    byte myArray[]; // Byte or Char array
    for (int i = 0; i < myArray.length; i++) {
        myArray[i] = C;
    }
}

Figure 17. Basic Idiom for memset
```

```java
// Translates an input array to an output array via a lookup table.
public void arraytranslate() {
    byte in[]; // Byte or Char array
    byte out[]; // Byte or Char array
    byte map[]; // Lookup table.
    int i = 0; // in array index.
    int j = 0; // out array index.
    for (i = 0; i <N; i++) {
        byte T = Map[in[i]];
        if (T == exitChar) break; //Terminate on certain chars (optional)
        out[j] = T;
        j++;
    }
}

Figure 18. Basic Idiom for arraytranslate
```
// Searches a char array for specific characters.
public void searchString() {
    char myCharArray[];  // Byte or Char array
    for (int i = 0; i < myCharArray.length; i++) {
        char T = myCharArray [i];
        if (T == 0x0A || T == 0x0D || T <= 0x40) // Exit condition.
            break;
        i++;
    }
    return myCharArray [i];
}

Figure 19. Basic Idiom for search string

**Synchronization**

Although synchronization is very important for concurrent programming in Java, a cost is associated with synchronization. This includes the cost of acquiring, as well as releasing, the lock that a thread holds. Another associated cost is starvation – the longer a thread holds onto a resource, the longer other threads might have to wait to acquire access to the resource. Incorrect uses of synchronization can also lead to deadlocks within the Java application.

The performance cost that is associated with `volatile` is almost the same as synchronization. `Long` `volatiles` are particularly expensive on some 32-bit platforms.

Minimizing the cost of synchronization usually means minimizing the contention on a shared resource.

- As much as possible, use the java.util.concurrent API because this offers more scalable nonblocking data structures.
- Avoid synchronization on the same lock when it is possible to use different locks to access different data by threads.
- Use the `synchronized` keyword judiciously. Make synchronized blocks short and move thread-safe code out of synchronized blocks in the application.
- Use thread-local data. Restrict data to a single thread or use `java.lang.ThreadLocal` to maintain data on a per-thread basis.
- Avoid synchronizing static methods.
Tuning

One of the most important steps to achieving optimal performance from Java applications is tuning. Tuning strategies can be applied to either the application code itself or to the underlying JVM parameters. This section outlines some of the techniques.

Tuning application code

It is important to avoid premature optimization when developing Java applications. During the development cycle, you should use profiling tools to determine hotspots in the application code. You can then tweak these sections of the code to deliver the best application performance.

Assumptions about what parts of the application are hot or cold might not be true in practice. After an application has been profiled, you should then separate the hot parts of the code (into their own methods) from the cold parts. Microbenchmarks are generally not representative of performance on actual workloads. It is better to measure the performance of the applications on real workloads, identify the bottlenecks in the application code and then tune these parts of the application.

A variety of tools exist to profile Java applications. For profiling the entire system, including Java applications, IBM has developed an Eclipse-based toolkit called IBM Visual Performance Analyzer (VPA) [16] that you can use for profiling applications to identify hotspots or performance problems. You can use VPA to visualize entire system profiles that are collected by using TPROF [20] or the Linux® operating-system profile tool called oprofile [19]. Profiling tools such as the Eclipse Test and Performance Tools Platform (TPTP) [6] and the IBM Rational® Application Developer for IBM WebSphere® Software [14], allow for profiling Java applications. The TPTP toolkit is an Eclipse platform that allows for profiling Java applications and the IBM rapid application development (RAD) tools help with analysis and visualization of running Java applications.

Tuning JVM parameters

The following sections briefly discuss some knobs available for tuning the garbage collector and options available when reducing application startup time is important. Finally, some techniques and tools available for diagnosing performance problems are discussed.

Tuning the garbage collector

Because the garbage collector (GC) can have a big impact on the performance of Java applications, it is important to tune the garbage collector, based on the characteristics of the application. Choosing the right parameters, garbage-collection policy and optimal heap settings can mean the difference between good and poor performance of the application.

In response to a variety of performance requirements from client applications and benchmarks, IBM SDKs, Java Technology Edition, Version 5.0 and Version 6.0 have implemented a number of high-performance garbage collectors to provide a broad selection of approaches and strategies for garbage collection. The objective of the framework is to give clients the flexibility of selecting a garbage collector
suitable for their applications in a given environment. Garbage collection policies are optimized for different application scenarios and a particular garbage collection policy [10] can be chosen depending on the characteristic of the application.

The size of the Java heap has a direct impact on the performance of Java programs. Specifying too small a Java heap can result in excessive garbage collection activities, which results in poor application performance. The IBM SDK provides command-line options to both shape the Java heap and tune its size for performance. The \texttt{--verbose:gc} and \texttt{--Xverbosegclog} [13] JVM options can be used to request detailed information about garbage collection activities as well as the memory footprint information of the application. An appropriate Java heap size can usually be derived from analyzing the frequency of garbage collection, its duration, and the Java heap occupation information. Then, with Java heap size information derived using these techniques, the \texttt{--Xmx} and \texttt{--Xms} options can be used to specify the maximum and minimum Java heap sizes, respectively. For 64-bit applications that do not need very large heap sizes (25 GB or less), the IBM SDK for Java 6 can make use of compressed references to reduce the size of objects, thereby making efficient use of the Java heap. This feature can be used for 64-bit Java applications where keeping memory footprint low is important. The compressed references feature can be enabled using the \texttt{--Xcompressedrefs} option in 64-bit JVMs [12].

To analyze the garbage-collection behavior of the IBM JDK when running the Java application, you can use the Garbage Collection and Memory Visualizer (GCMV) tool [4] developed by IBM. The tool helps visualize the application’s memory-usage pattern, detects memory leaks during execution, and also helps in tuning the various parameters of the garbage collector to improve application performance. Another tool for discovering possible memory leaks in an application is the HeapAnalyzer [9].

\section*{Tuning JVM parameters for application startup}

If the startup performance of a Java application is important, you can minimize the startup time by using the IBM \texttt{quickstart} and \texttt{class sharing} technologies. The nonstandard \texttt{--Xquickstart} option reduces the initial compilation of methods to a lower compilation level, as compared to the default mode [11]. Although performing quicker compilations for more methods can improve application startup, it can also degrade the performance of long-running applications that contain hot methods. As a result, you should only use this JVM option for applications where initial startup speed is more important than the long-running throughput.

IBM SDK for Java 5 introduced class-sharing technology, which saves class data from the current invocation of an application into a persistent cache [17]. Subsequent invocations of the application can simply reload the data from the persistent cache, thereby reducing the virtual memory footprint and improving startup time. In IBM SDK for Java 6, this technology was extended to also save Ahead-Of-Time (AOT) compilations of methods. Subsequent iterations can reload these AOT compilations directly from the cache, effectively gaining the benefit of compiled code without incurring compilation costs. The JVM option to enable class-sharing technology is \texttt{--Xshareclasses}.

Furthermore, application startup time can also be improved by recompiling the application classes with the Java compiler that is provided with the IBM JDK. In particular, compile-time inlining of JSR bytecodes [2] and the generation of stack maps (in Java 6.0) reduces the time that is taken to load classes. You can also reduce memory usage and startup time by directing the compiler to produce only the required
debugging information. For example, if you deploy an application with no need for Java-debug support, then you can omit the local-variable tables.

**Diagnostic techniques and tools available for performance problems**

Even after tuning the important parts of the application code, the Java application may still suffer from poor performance. This may be due to system characteristics or some uncharacteristic behavior of the underlying JVM runtime environment. Given that the JIT compiler and the garbage collector have the maximum impact on the performance of applications, we discuss some types of diagnostic information that can be gathered to help with the diagnosis of application performance problems.

As mentioned in the previous section, `–verbose:gc` and `–Xverbosegclog` [13] JVM options can be used to request detailed information about GC activities. This information will help determine the GC overhead costs and potential GC tuning opportunities. A heapdump which is a snapshot of the Java heap at any given point during the execution of the Java application can also be obtained as described in the diagnostic guides [12].

The TRJIT compiler provides a command-line option `–Xjit:verbose`. With this option, the TRJIT compiler generates detailed information about which application methods were compiled and determined by the TRJIT compiler to be executed frequently. This information can help in diagnosing potential bottlenecks in the Java application.

At any point during the runtime execution of the Java application, a javacore can be triggered by sending a signal to the JVM. The javacore provides various diagnostic information including operating system details, application threads, locks and memory (including the heap, JIT compiler and JVM itself) usage by the JVM. This information can be used to detect the root cause of hangs, deadlocks or resource contention in the system. The IBM Thread and Monitor Dump Analyzer for Java [15] is a tool that can be used to analyze javacores produced by the IBM SDK.

**Summary**

Using good and simple design principles to write Java applications is better than trying prematurely to make such applications run faster through performance-tweaking methods. However, by following the appropriate guidelines mentioned in this article, and carefully tuning the application code, it is possible to get the best performance out of the application running with the IBM Java Development Kit 6.0.
Footnotes


[12]. IBM Java Diagnostic Guide. ibm.com/developerworks/java/jdk/diagnosis


[14]. IBM Rational Application Developer for WebSphere Software. ibm.com/software/awdtools/developer/application


Resources

These Web sites provide useful references to supplement the information contained in this document:

- IBM System p® Information Center
  http://publib.boulder.ibm.com/infocenter/pseries/index.jsp
- System p on IBM PartnerWorld®
  ibm.com/partnerworld/systems/p
- System z on IBM PartnerWorld®
  ibm.com/partnerworld/systems/z
- IBM AIX® on IBM PartnerWorld®
  ibm.com/partnerworld/aix
- IBM z/OS® on IBM PartnerWorld®
  ibm.com/partnerworld/zos
- Linux on System z on IBM PartnerWorld®
  ibm.com/partnerworld/systemz/linux
- IBM Publications Center
- IBM Redbooks® publications
  ibm.com/redbooks
- IBM developerWorks® site
  ibm.com/developerworks
About the authors

Pramod Ramarao holds Bachelor of Engineering (B.E.) and Master of Science (M.S.) degrees in Computer Engineering. He joined IBM in 2003 and has since been working on the optimizer component of the IBM TR JIT compiler.

Joran Siu obtained his Bachelor of Science degree in Electrical and Computer Engineering from Cornell University. Since joining IBM, he has been contributing to the IBM TR JIT compiler, with a focus on performance on the IBM System z platform.

Pavan Pamula holds Bachelor of Engineering (B.E.) in Electronics and Communications Engineering and Master of Science (M.S.) degrees in Software Engineering. He is currently a technical lead for the IBM software development kit (SDK) for Java enablement team. He leads IBM Systems family and independent software vendor (ISV) initiatives with IBM SDK for Java development and support teams.
Trademarks and special notices

© Copyright IBM Corporation 2008.

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Neither International Business Machines Corporation nor any of its affiliates assume any responsibility or liability in respect of any results obtained by implementing any recommendations contained in this article. Implementation of any such recommendations is entirely at the implementor’s risk.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.