Specialization of Java Generic Types

Scott Lembcke, Sam BeVier, Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
lemb0029, bevi0022, elenam@morris.umn.edu

Abstract

Generic types are a new feature in the Java programming language, and allow the programmer to write a class or method that has a type parameter. Type parameters allow the program to create different instances of the parameterized class or method replacing the type parameter with a specific type such as String or Integer. This feature is similar to C++ templates, but differs in implementation. Generic types in Java are implemented using type erasure, which works by removing type information from variables at compile time and automatically generating code to cast them to the correct type. This contrasts with C++ templates, which create a specialized copy for each instance of a type parameter.

Type erasure creates only one implementation for each generic class or method, minimizing code duplication. However, erasure may not produce the most efficient program. By producing an implementation specific to each instance type, the compiler has an opportunity to optimize the program according to that type. This process is called specialization.

We propose a transformation of program source code that allows the Java compiler to create implementations of generic classes and methods specialized for the types used in the program. Such a transformation is beneficial in some cases, but if applied too aggressively, can decrease runtime efficiency. We explore how this specialization affects the runtime efficiency of various programs. We demonstrate that for a variety of programs that use generic methods and classes specializing generic types results in substantially more efficient programs.

1 Introduction to Generic Types

Generic types are a feature that is present in many modern programming languages. Using generic types, a programmer can write a data structure or a function that can work for different data types. The type in such data structures or functions is specified as a type parameter, i.e. it is denoted by a variable. When using a parameterized entity, the programmer specifies a concrete type for the type parameter. This process is called type instantiation. Type safety is still maintained: a linked list parameterized to contain integers will give a compiler error if a string is added to the list. This makes code more reusable and increases type safety. There are two basic ways that generic types can be implemented. One is a template approach, and is implemented in C++. The other is type erasure which is primarily geared towards user convenience. Java uses the type erasure approach.

1.1 Implementations of Generic Types

In the template approach, the compiler produces a separate copy of the entire data structure or function for each instantiation used in the program. Type erasure works quite differently. There are no duplicate copies of code that are specific to different types. At compilation time, the compiler removes all specific type information, replacing the type with the most general supertype, and inserts appropriate type casts wherever needed.

Erasing the type information at compile time means that further type specific program optimizations are unavailable. This can decrease the time efficiency of programs, especially those with complicated type hierarchies. Templates do not present this problem. However, templates create multiple copies of the code, which is less memory efficient and can be confusing to the programmer.

2 Java Compilation Model

Java is unlike most other compiled languages in that it compiles a program to a platform independent bytecode instead of native code. This part of the compilation is called static compilation and allows Java to offer a compromise between speed, runtime safety, and platform independence. Originally, the JVM (Java virtual machine) interpreted the bytecode, but since this was slow, JIT (just in time) compilers were added to the JVM to compile the bytecode into native code as it is run. However, JIT compilers can create an additional overhead because compiling and optimizing code at run time adds to the running time of the program. To address this problem, Sun developed the Java HotSpot™ VM which decreased this overhead by profiling the program being run to selectively compile only "hot spots", or frequently called methods, into native code. This way time is not needlessly spent performing expensive compilation and optimization on code that does not significantly affect the runtime of a program. Compiling the bytecode to native code as the program is running is called dynamic compilation.

2.1 Common Optimizations

Compiler optimizations relevant to our research include constant propagation and folding, inlining, and devirtualization [3]. Constant propagation is when the compiler replaces any instance of a constant in a program with its value. Constant folding is when the compiler evaluates expressions in the code that can be evaluated at compile time. Inlining is an optimization where a method call is replaced by the code of the method. After inlining a method, the code can be further optimized. This could potentially allow an entire method call to be replaced by a constant value which may open up even more opportunities for constant folding.

Another optimization is called devirtualization. A virtual function call is when a function must be looked up in the class hierarchy before it can be called. The name comes from the *virtual* modifier in C++ which can be thought of as the opposite of the *final* modifier in Java. Non-virtual (in particular, *final*) functions are those defined only in one class along the class hierarchy, and therefore their address is known before the program runs. Devirtualization eliminates an expensive function lookup when the address of a function can be determined at compile time.

The static compiler transforms Java source into bytecode and does relatively few optimizations. Such optimizations include constant propagation and folding [2]. Most of the optimization is performed by the dynamic compiler as it compiles the bytecode into native code. The reasons for that are explained in Section 2.3. Runtime, or dynamic, optimizations would include more complex optimizations such as devirtualization and method inlining, possibly followed by constant propogation and folding, and similar optimizations. As mentioned before, dynamic optimizations are not performed universally, but selectively. The JVM profiles the running program to identify "hot spots" that would benefit the most from optimization.

2.2 JVM Client and Server Modes

There are two modes for running the JVM, a client and server mode. The client mode has a simpler and faster compiler. It is intended to start a program running as quickly as possible without performing heavy optimization. The client mode is also more memory efficient. The server mode compiler focuses more on optimizing the runtime of a program than on faster compilation speed and startup time [4].

2.3 Dynamic Loading

In Java, it is possible to load classes at runtime**.** This is known as dynamic loading, and is the main reason for delaying optimizations until runtime. Inlining cannot be performed by the static compiler because a class may be reloaded at runtime, replacing the code running in the JVM with new code. Any code inlined by the static compiler could become invalid. Similarly, a method call cannot be devirtualized by the static compiler because it may not know all of the types that are used.

3 Java Type System

Like most object-oriented languages, Java supports type hierarchies. Classes can inherit fields and methods from other classes. There are two kinds of types in Java: primitive types, such as integers and booleans (denoted as int and bool, respectively), and object types. Object types include some predefined types, for instance strings and arrays. Additionally a user can define their own object types. A class may contain fields (i.e. variables) and methods. To declare a method, a programmer specifies the method's name, the number and types of its parameters, its return type, and, finally, the method's code, i.e. the actions performed by the method. All of the above, except for the method's code, is called the method's descriptor.

3.1 Class Hierarchy

All object types are a part of Java class hierarchy: classes can inherit from each other. By default, a class inherits from the predefined class *Object*. A class can inherit directly only from one class, but if a class *A* inherits from a class *B*, it also indirectly (transitively) inherits from all classes that *B* directly and indirectly inherits from, including the *Object* class. When a class *A* inherits from a class *B*, it automatically includes all of *B*'s methods (those declared directly in *B* and those that *B* in turn inherits through inheritance). However, *A* can also declare additional methods and overwrite its inherited methods. A method in a superclass is overwritten by a method in its subclass if the two methods have the same name and argument types, i.e. they have the same descriptor. If a method is overwritten in a subclass, a call to this method on an object of this subclass will call the method of the subclass, not the method of the superclass. However, if a subclass does not overwrite a method, the method with this descriptor defined in the closest superclass will be called. The program will compile only if all methods called on an object are defined in the object's class or in one of its superclasses.

3.1 Interfaces

In addition to classes Java type hierarchy includes interfaces. An interface can be seen as a class specification, but not an implementation. An interface lists method descriptors, but not the code, of all methods required by this interface. A class can implement an interface, in which case the class must define all methods declared in the interface. A class can implement several interfaces. A common Java interface that we use in our test programs is *Comparable*. This interface requires that the class provides a method

```
int compareTo(Object o) {...}
```

This method allows the object to be compared to other objects of the same type. The result of the method is negative, positive, or 0, signifying that the given object is, respectively, less than, greater than, or equal to the object passed as a parameter. Generic methods that sort data or otherwise require that data items can be compared to each other are usually defined for Comparable objects.

Arguments to type parameters can be any object type (i.e. one that is part of the class hierarchy). This means that primitive types cannot be used as a type argument to a generic type or method as they are not objects. Therefore Java provides object versions of the primitive types, simple objects with a field that contains the value of the primitive type. For instance, *Integer* is an object that contains a single integer value. Since *Integer* objects can be compared to each other, the class Integer implements the *Comparable* interface.

3.2 Type casting

Often objects are passed to a method or returned from a method as variables of their superclass or an interface they implement, not as the most specific class of the object. For instance, one can pass two Integers to a minimum method that works on *Comparable* objects:

```
Comparable min (Comparable a, Comparable b) {
      if (a.compareTo(b) < 0) return a;
      else return b;
}
```

Note that the minimum is returned as a *Comparable*, not as an *Integer*. In order for it to be used as an *Integer* (for instance, be assigned to a variable of type Integer), it has to have a type cast applied to it. This is done by adding the type name in parentheses before the value:

```
Integer i = (Integer) min(n,m);
```

4 Generic types in Java

Generic types were added to Java in release 1.5 (also known as Java 5). Many programming languages provide the convenience of generic types, and their absence in Java was viewed as a significant deficiency. There has been a variety of proposals for extending Java with generic types. When adding generics to Java, it was highly desirable to make such an extension compatible with the existing JVM and the existing specification of Java bytecode so that the Java legacy code was not affected by the new addition. Type erasure is such an implementation. The static compiler checks the type-safety of the generic code, inserts all necessary type casts, and outputs the bytecode which has no information about specific type parameters used to instantiate generic classes or methods. This means that no instance-specific type information can be used by generic code at run-time.

4.1 Generic Classes

When writing a generic, i.e. parameterized class, the class declaration contains the type parameter in angle brackets. The methods in a parameterized class then can use that parameter as a return type or use it for its parameter types. The methods now view that type as they would any concrete type because it is declared in the class declaration. For instance, consider the following class declaration:

```
public class LinkedList<K> {
      ...
      public K getFront() { ... }
      ...
      public void setFront(K k1) { ... }
      ...
}
```

Where <K> is the type parameter passed to the class upon instantiation. Notice that K is then used as the return type for one method and a parameter type in another.

When writing parameterized code, it is also possible to restrict the range of acceptable types by introducing an upper bound. An upper bound (a class or an interface) can be set so that only subtypes of that bound can be passed as the type parameter. The keyword *extends* is used to set the upper bound.

```
public class LinkedList<K extends Comparable> { ... }
```

Any class that implements the interface *Comparable* will be accepted by this implementation of *LinkedList*.

Whenever an instance of a parameterized object is created, the type parameter must be specified. For instance:

```
LinkedList<String> _list = new LinkedList<String>();
```

The above code creates a *LinkedList* which can only contain *String* objects. The compiler checks that the actual type (in this case, *String*) is a subtype of the bound of the type parameter. If it is not, the code will not compile.

4.2 Generic Methods

Methods can be parameterized even if the class they are in isn't. In these cases, the method descriptors just have to contain the type parameter in angle brackets much like the parameterized classes do.

```
public <K> K someMethod() { ... }
```

For example, consider a generic method *min()* that determines a minimum of two objects. As we mentioned in section 3.1, *Comparable* is an interface requiring a class to have a method *compareTo()* that determines the order of objects of that class. Any time the method *min()* is called, it has to be on a class that implements *Comparable*. Note that, unlike the example in Section 3.1, the method below is written using generic types.

```
public <K extends Comparable> K min(K k1, K k2) {
      if (k1.compareTo(k2) > 0) {
            return k2;
      } else { return k1;}
}
```

4.3 Implementation

Type erasure works by replacing the type parameter by its bound at compile time. If no bound is specified, the compiler assumes that the bound is Object. Specifying Object as the upper bound and not specifying any bound are actually the same thing. Here is a very simple example of generic code:

```
public class GenericMethod{
      public static <T> T aMethod(T anObject){
            return anObject;
      }

      public static void main(String[] args){
            String greeting = "Hi";
            String reply = aMethod(greeting);
      }
}
```

The compiler would transform this into something equivalent to the following code:

```
public class GenericMethod{
      public static Object aMethod(Object anObject){
            return anObject;
      }

      public static void main(String[] args){
            String greeting = "Hi";
            String reply = (String)aMethod(greeting);
      }
}
```

Notice the generic information is gone, casts have been inserted, and the return value of the method now has a type cast to *String*.

There are features of generic types in Java that we are not presenting at this point such as wildcards and bounded wildcards. Optimizations related to these features will be the subject of further research.

5 A Case for Specialization

While type erasure presents a simple implementation of generic types, it has several inefficiency problems. By removing the type information, type erasure removes the potential for some optimizations. A method call cannot be inlined or devirtualized unless the type of the object that it is called on is known exactly. Additionally, type casts may have to be added when calling methods or returning values because the type information cannot be checked by the static compiler.

Replacing the parameter bound with a more specific bound opens more possibilities for optimization. This can be done as a transformation of the source code. For example, consider the following modified version of *GenericMethod*:

```
public class GenericMethod{
      public static <T> T aMethod(T anObject) {
            ((Comparable)anObject).compareTo(anObject);
            anObject.toString();
            return anObject;
      }

      public static void main(String[] args){
            String greeting = "Hi";
            String reply = aMethod(greeting);
      }
}
```

Now we will transform *GenericMethod* into *SpecificMethod* by changing the method declaration for *aMethod()* to:

```
public static <T extends String> T aMethod(T anObject)
```

In *SpecificMethod*, the compiler will erase the return type and argument type of *aMethod* to *String*. At static compilation time it is known that String is comparable, so the static compiler will not actually have to add a cast of *anObject* to *Comparable*. The return type is also *String*, so it will not need a cast to *String* to be assigned to *reply*.

Below we show the bytecode that the static compiler generated for *aMethod()* in both classes and examine the differences.

The *checkcast* instruction checks that an object belongs to a given type. *invokeinterface* dynamically finds the method required by the interface (in this case, the method *compareTo()*) in the object's class, and *invokevirtual* dynamically finds the right method along the object's class hierarchy. The other bytecode instructions are irrelevant to the example.

GenericMethod.aMethod():

```
 0 aload_0
 1 checkcast #2 <java/lang/Comparable>
 4 aload_0
 5 invokeinterface #3 <java/lang/Comparable.compareTo> count 2
10 pop
11 aload_0
12 invokevirtual #4 <java/lang/Object.toString>
15 pop
16 aload_0
```

Specific.aMethod():

```
 0 aload_0
 1 aload_0
 2 invokeinterface #2 <java/lang/Comparable.compareTo> count 2
 7 pop
 8 aload_0
 9 invokevirtual #3 <java/lang/String.toString>
12 pop
13 aload_0
14 areturn
```

GenericMethod.main():

```
 0 ldc #4 <Hi>
 2 astore_1
 3 aload_1
 4 invokestatic #5 <Generic.aMethod>
 7 checkcast #6 <java/lang/String>
10 astore_2
11 return
```

SpecificMethod.main():

```
0 ldc #3 <Hi>
2 astore_1
3 aload_1
4 invokestatic #4 <Specific.aMethod>
7 astore_2
8 return
```

As you can see, there are extra *checkcast* instructions present in the bytecode of *GenericMethod*. Checking casts, like method look ups, can be expensive to perform. Interestingly, despite removing the cast to *Comparable*, the static compiler did not use *invokevirtual* on the *compareTo()* method call. However, it is possible that this call is further optimized (inlined or devirtualized) at run time since the method descriptor for *aMethod()* in *SpecificMethod* specifies that the method parameter as a *String*. However, calling a method on an object directly makes the type more explicit. Calling a *final* method on a *final* object is a perfect candidate for method inlining or devirtualization

6 Results

6.1 Testing Methodology

Our testing consisted of creating a test program that would highlight one aspect of generic programming, and then modify the bounds used for the type parameter. We then recorded the runtime of the program in several ways. We placed a print statement inside the program that would only time a specific section of the code. We also timed the execution of the JVM with the Unix *time* command, which includes the system and user CPU time, total elapsed time, and other statistics. Each program was run in both the client and server mode of the JVM 20 times each. Programs that used random numbers were seeded externally so that different versions of the same program could be run with the same series of seed numbers. All tests were ran on the same machine running Fedora Core with version 1.5.0_04 of both the Java HotSpot VM and the javac compiler.

We found in many of our test examples that the creation of a large number of objects sometimes took more time than the runtime of the actual testing code. To avoid creating too many objects, we loop the code many times on the same data. We refer to the execution time of this loop as the loop time.

6.2 Bubble Sort Example

For our first example, we chose to use bubble sort. We did so because it is a well known and inefficient sorting algorithm. To increase the number of comparison even further, we sort the same array forward and backward several times. Since the worst case of bubble sort is an array sorted in reverse order, this maximizes the number of comparisons. This allowed us to make a large number of comparisons on a relatively small amount of data. We focused on generic methods first by implementing it as a generic method that accepted an array of a generic type.

We created several variations to test different bounds for String and Integer. We chose <T> (*Object*), <T extends Comparable<T>> (*Comparable*), and <T extends Integer> (*Integer*) as bounds for our *Integer* test, and similar bounds for *String*. Also, in our *String* test we created fewer *String* objects and ran the loop fewer times to keep the runtime from getting too long or the memory usage too high. The code for the *Object* bounded version of our *Integer* test is as follows:

```
public class TestBubbleSortObjectBound {
	public static <T> void sortfront(T [] _list) {
		T temp;
		T current;
		T next;
		boolean whilecondition = true;
		boolean repeatcondition = false;
		while (whilecondition) {
			for (int i = 1; i < _list.length; i++) {
				current = _list[i - 1];
				next = _list[i];
				if (((Comparable)current).compareTo(next) > 0) {
					repeatcondition = true;
					temp = next;
					_list[i] = current;
					_list[i - 1] = temp;
				}
			}
			if (!repeatcondition) {
				whilecondition = false;
			} else {
				repeatcondition = false;
			}
		}
	}

	public static <T> void sortback(T [] _list) {
		... same as sortfront, but works in reverse
	}

	public static void main(String[] args) {
		int n = 10000;
		Integer[] array = new Integer[n];
		for (int i = 0; i < n; i++) {
			array[i] = new Integer(n - i);
		}

		long time1 = System.currentTimeMillis();

		for(int i = 0; i < 2; i++){
			sortfront(array);
			sortback(array);
		}

		long time2 = System.currentTimeMillis();
		System.out.printlntime2 - time1);
	}
}
```

Our test results were as follows:

Bubble Sort Integer	Client	Server
<T extends Integer>	7536 - 7611 - 7835	3130 - 3171 - 3452
<Comparable<T>>	10006 - 10053 - 10447	6160 - 7061 - 7504
<T>	10549 - 10645 - 11028	8564 - 8884 - 9050

Loop run times in milliseconds. (minimum – median – maximum)

Bubble Sort Integer	Client	Server
<T extends Integer>	7.56 - 7.65 - 7.86	3.21 - 3.26 - 3.52
<Comparable<T>>	10.03 - 10.08 - 10.47	6.25 - 7.13 - 7.59
<T>	10.56 - 10.67 - 11.06	8.64 - 8.96 - 9.14

Executable runtimes in user CPU seconds. . (minimum – median – maximum)

Bubble Sort Integer	Client	Server
<T extends Integer>	7.60 - 7.71 - 7.94	3.29 - 3.29 - 3.62
<Comparable<T>>	10.06 - 10.12 - 10.51	6.31 - 7.21 - 7.66
<T>	10.62 - 10.74 - 11.13	8.72 - 9.06 - 9.30

Executable runtimes in seconds. . (minimum – median – maximum)

Bubble Sort String	Client	Server
<T extends String>	568 - 572 - 577	396 - 489 - 490
<Comparable<T>>	671 - 676 - 705	570 - 657 - 659
<T>	636 - 641 - 647	549 - 637 - 646

Loop run times in milliseconds. (minimum – median – maximum)

Bubble Sort String	Client	Server
<T extends String>	0.59 - 0.61 - 0.62	0.47 - 0.56 - 0.58
<Comparable<T>>	0.70 - 0.71 - 0.74	0.65 - 0.74 - 0.76
<T>	0.66 - 0.68 - 0.70	0.62 - 0.72 - 0.73

Executable runtimes in user CPU seconds. . (minimum – median – maximum)

Bubble Sort String	Client	Server
<T extends String>	0.66 - 0.66 - 0.66	0.54 - 0.60 - 0.60
<Comparable<T>>	0.77 - 0.77 - 0.77	0.72 - 0.77 - 0.78
<T>	0.71 - 0.71 - 0.71	0.66 - 0.77 - 0.78

Executable runtimes in seconds. . (minimum – median – maximum)

In our tests, we compare the minimum and median values. We do this because in many sets of results, the maximum values are significantly different than the rest of the runtimes. Comparing the lower half ignores these values. The maximum values are listed for clarity.

These results clearly show that the generic bound does affect the runtime of the program. Also of note is that the server mode of the JVM especially benefits from the added type information as the runtimes for Integer were more than cut in half.

6.3 Sorted Linked List

Our next test was to try specializing one of the built in Java collections. *TreeMap* seemed to be the most promising, since it's a data structure that maintains a tree of ordered elements, and thus uses *compareTo()* method extensively. Unfortunately, we found that the type bound of the *TreeMap* class did not automatically propagate to its parameterized inner classes. Changing the bound on the inner classes would have been very difficult due to the way the class hierarchy had been set up, so instead we decided to make a more controlled example. We created a sorted linked list class that used insertion sort and had an inner node class. Because it was not part of a predefined hierarchy already, we could set the bounds as we needed.

Like in our bubble sort example, we created several variations of the *SortedLinkedList* class with different generic bounds. We tested the list by creating an array of random Integer objects to be inserted and removed from the list. The source code is not shown due to its length.

Our results were as follows:

Sorted Linked List	Client	Server
<T extends Integer>	2463 - 2514 - 2532	1137 - 1210 - 1300
<Comparable<T>>	2458 - 2497 - 2520	1216 - 1273 - 1421
<T>	2989 - 3038 - 3067	1647 - 1975 - 2133

Loop run times in milliseconds. (minimum – median – maximum)

Sorted Linked List	Client	Server
<T extends Integer>	2.50 - 2.56 - 2.58	1.23 - 1.31 - 1.40
<Comparable<T>>	2.50 - 2.53 - 2.56	1.29 - 1.38 - 1.52
<T>	3.03 - 3.08 - 3.12	1.74 - 2.07 - 2.23

Executable runtimes in user CPU seconds. . (minimum – median – maximum)

Once again, changing the bound has a significant impact on the runtimes, and especially so in the server mode of the JVM. In the client mode, the variation using the Comparable bound seems to fare slightly better than the Integer bound, but in the server mode the variation using the Integer bound is clearly faster.

We also created two tests that used *String* and *Integer* instantiations of *SortedLinkedList*. One test used the *Object* bounded variation of *SortedLinkedList* and the other used the *String* and *Integer* bounded variations. The point of this test was to see how significant the overhead of using two separate implementations was. Comparing the difference in the minimum, median and maximum values we get:

Dual Sorted Linked List	Client	Server
Specialized	0.09 - 0.12 - 0.13	0.16 - 0.19 - 0.18
Object	0.08 - 0.09 - 0.11	0.17 - 0.14 - 0.17

Difference between loop and elapsed run time in seconds. (minimum – median – maximum)

Here, we can see the overhead of the extra compilation. The *SortedLinkedList* class is a fairly simple class with very little code. With more code and specializations the overhead could grow to be far more significant. In our future research we will study when and if this overhead could outweigh the benefits of specialization.

7 Conclusions and Future Work

Our results show that when a more specific bound of a generic class or a method is given, the code produced by the compiler and further optimized by the JVM is significantly more efficient. This proves that a program transformation that replaces the type bound by the actual type parameter would lead to more efficient programs. This transformation can be implemented as a source-to-source transformation, i.e. as a transformation of a Java program into another Java program in which the type bounds are replaced by the most specific types of instances.

We also observed that such transformations on real-life code may be non-trivial since parametric types may form their own class hierarchy, and the transformation may affect multiple classes in this hierarchy, as in the case of Java *TreeMap* collection. This may produce a large number of extra classes, especially if the same generic class was instantiated with several different type parameters in the same program.

Our future work is to implement the transformation outlined above as a preprocessor that transforms the program before the compilation stage. Initially we will write a preprocessor that handles programs with a simple type hierarchy (classes with no inheritance among generic classes, no static fields or methods, etc.). As the next phase, we plan to investigate the efficiency of this transformation for real-life Java programs using the preprocessor. One of the goals is to study programs with several instantiations of the same generic class with different type parameters and to find out when the addition of extra classes leads to a noticeable overhead because of handling the extra classes in JVM. The algorithm can then be modified accordingly so that it does not create too many instances of the parameterized class. As a longer goal, we plan to extend the preprocessor to handle other features of Java type, such as static methods and fields, parametric inner classes, hierarchy of generic types, generic types instantiated with other generic types, and wildcards.

Acknowledgements

The authors thank Nic McPhee and Matthew Justin of University of Minnesota, Morris for helpful discussion of Java collections and other related material.

References

[1] Gilad Bracha "Generics in the Java Programming Language". Tutorial by Sun Microsystems, Inc., available at http//java.sun.com

[2] Steve Caudill, Elena Machkasova (Advisor) "Empirical Studies of Java Optimizations", MICS 2005

[3] Michael Paleczny, Christopher Vick, and Cliff Click "The Java HotSpot™ Server Compiler", Java™ Virtual Machine Research and Technology Symposium 2001.

[4] White Paper "The Java HotSpot Virtual Machine, v1.4.1", Sun Microsystems, Inc., available at http//java.sun.com