THE UNIVERSITY
of MANCHESTER

February 15, 2002

Computer Science
University of Manchester

Elimination of Java Array Bounds Checks in the Presence of Indirection

Mikel Luján, John R. Gurd, T. L. Freeman
and José Miguel

Department of Computer Science
University of Manchester
Preprint Series
CSPP-13

Elimination of Java Array Bounds Checks in the Presence of Indirection

Mikel Luján,[*] John R. Gurd,[†] T. L. Freeman[†] and José Miguel[‡]
[†]Centre for Novel Computing, University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
[‡]Dep. de Arquitectura y Tecnología de Computadores, UPV-EHU
P.O. Box 649, 20080 Donostia-San Sebastian, Spain
{mlujan, jgurd, lfreeman}@cs.man.ac.uk miguel@si.ehu.es

February 15, 2002

Abstract

The Java language specification states that every access to an array needs to be within the bounds of that array; i.e. between 0 and length - 1. Different techniques for different programming languages have been proposed to eliminate explicit bounds checks. Some of these techniques are implemented in off-the-self Java Virtual Machines (JVMs). The underlying principle of these techniques is that bounds checks can be removed when a JVM/compiler has enough information to guarantee that a sequence of accesses (e.g. inside a for-loop) is safe (within the bounds).

Most of the techniques for the elimination of array bounds checks have been developed for programming languages that do not support multi-threading and/or enable dynamic class loading. These two characteristics make most of these techniques unsuitable for Java. Those techniques developed specifically for Java have not addressed the elimination of array bounds checks when the index is stored in another array (*indirection array*) – in the presence of indirection.

With the objective of optimising applications with array indirection, this paper proposes and evaluates three implementation strategies, each implemented as a Java class. The classes provide the functionality of Java arrays of type `int` so that objects of the classes can be used instead of indirection arrays. Each strategy enables JVMs, when examining only one of these classes at a time, to obtain enough information to remove array bounds checks.

Recent preprints issued by the Department of Computer Science, Manchester University, are available on WWW via URL `http://www.cs.man.ac.uk/preprints/index.html` or by ftp from `ftp.cs.man.ac.uk` in the directory `pub/preprints`.

[*]acknowledges the support of a research scholarship from the Department of Education, Universities and Research of the Basque Government.

[†]JM acknowledges: this research has been funded by the Spanish MCYT under grant TIC2001-0591-C02-02.

1 Introduction

The Java language [JSGB00] was specified to avoid the most common errors made by software developers. Arguably, these are memory leaks, array indices out-of-bounds and type inconsistencies. These features are the basic pillars that make Java an attractive language for software development. On these basic pillars Java builds a rich API for distributed (network) and graphical programming, and has built-in threads. All these features are even more important when considering the significant effort devoted to make Java programs portable.

The cost of these enhanced language features is/was performance. The first generation of Java Virtual Machines (JVMs) [LY99] were bytecode interpreters that concentrated on providing the functionality and not performance. Much of the reputation of Java as a slow languages comes from early performance benchmarks with these immature JVMs.

Nowadays, JVMs are in their third generation and are characterised by:

- mixed execution of interpreted bytecodes and machine code generated just-in-time,
- profiling of application execution, and
- selective application of compiler transformations to time-consuming sections of the application.

The performance of modern JVMs is increasing steadily and getting closer to that of C and C++ compilers (see for example the Scimark benchmark [PM] and the Java Grande Forum Benchmark Suite [BSPF01, EPC]); especially on commodity processors/operating systems. Despite the remarkable improvement of JVMs, some standard compiler transformations (e.g. loop reordering transformations) are still not included, and some overheads intrinsic to the language specification remain.

The Java Grande Forum, a group of researchers in High-Performance Computing, was constituted to explore the potential benefits that Java can bring their research areas. From this perspective the forum has observed its limitations and noted solutions [Jav98, Jav99, PBG+00, BMPP01, Thi02].

The overhead intrinsic in the language specification that this paper addresses is the array bounds checks. The Java language specification requires that:

- all array accesses are checked at run time, and
- any access outside the bounds (less than zero or greater than or equal to the length) of an array throws an `ArrayIndexOutOfBoundsException`.

The specific case under consideration is the elimination of array bounds checks in the presence of indirection. For example consider the array accesses `foo[ indices[6] ]`, where `indices` and `foo` are both Java one-dimensional arrays of type `int`. The array `foo` is accessed trough the *indirection array* `indices`. Two checks are performed at run time. The first checks the access to `indices` and it is out of the scope of this paper. Readers interested in techniques implemented for Java in JVMs/compilers for eliminating checks of this kind of array accesses can consult [PTH+97, MS99, SOT+00, PVC01, FKR+00, CLS00, AAB+00, BGS00, MMS98]. The second checks the access to `foo` and it is this kind of array bounds check that is the subject of this paper.

When an array bounds check is eliminated from a Java program, two things are accomplished from a performance point of view. The first, direct, reward is the elimination of the check itself

(at least two integer comparisons). The second, indirect, reward is the possibility for other compiler transformations. Due to the strict exception model specified by Java, instructions capable of causing exceptions are inhibitors of compiler transformations. When an exception arises in a Java program the user-visible state of the program has to look like as if every preceding instruction has been executed and no succeeding instructions have been executed. This exception model prevents, in general, instruction reordering.

The following section motivates the work and provides a description of the problem to be solved. Section 3 describes related work. Section 4 presents the three implementation strategies that enable JVMs to examine only one class at a time to decide whether the array bounds checks can be removed. The solutions are described as classes to be included in the Multiarray package (JSR-083 [JSR]). The strategies were generated in the process of understanding the performance delivered by our Java library OoLaLa [Luj99, LFG00, LFG02a, LFG02b], independently of the JSR. Section 5 evaluates the performance gained by eliminating array bounds checks in kernels with array indirection; it also determines the overhead generated by introducing classes instead of indirection arrays. Section 6 summarises the advantages and disadvantages of the strategies. Conclusions and future work are presented in Section 7.

2 Motivation and Description of the Problem

Array indirection is ubiquitous in implementations of *sparse matrix* operations. A matrix is considered sparse when most elements of the matrix are zero; less than 10% of the elements are nonzero elements. This kind of matrix arises frequently in Computational Science and Engineering (CS&E) applications where physical phenomena are modeled by differential equations. The combination of differential equations and state-of-the-art solution techniques produces sparse matrices.

Efficient storage formats (data structures) for sparse matrices rely on storing only the nonzero elements in arrays. For example, the coordinate (COO) storage format is a data structure for a sparse matrix that consists of three arrays of the same size; two of type `int` (`indx` and `jndx`) and the other (`value`) of any floating point data type. Given an index k, the elements in the k-th position in `indx` and `jndx` represent the row and column indices, respectively. The element at the k-th position in the array `value` is the corresponding matrix element. Those elements that are not stored have value zero. Figure 1 presents the kernel of a sparse matrix-vector multiplication where the matrix is stored in COO. This figure illustrates an example of array indirection and the kernel described is commonly used in the iterative solution of systems of linear equations. "It has been estimated that about 75% of all CS&E applications require the solution of a system of linear equations at one stage or another [Joh82]."

```
public class ExampleSparseBLAS {
    // y = A*x + y
    public static void mvmCOO ( int indx[], int jndx[],
          double value[], double vectorY[], double vectorX[] ) {
        for (int k = 0; k < value.length; k++)
            vectorY[ indx[k] ] += value[k] * vectorX[ jndx[k] ];
    }
}
```

Figure 1: Sparse matrix-vector multiplication using coordinate storage format.

Array indirection can also occur in the implementation of irregular sections of multi-dimensional

arrays [MMG99] and in the solution of non-sparse systems of linear equations with pivoting [GvL96].

Consider the set of statements presented in Figure 2. The statement with comment `Access A` can be executed without problems. On the other hand the statement with comment `Access B` would throw an `ArrayIndexOutOfBoundsException` exception because it tries to access the position -4 in the array `foo`.

```
int indx[] = {1, -4};
double foo[] = {3.14159, 2.71828, 9.8};
... foo[ indx[0] ] ... // Access A
... foo[ indx[1] ] ... // Access B
```

Figure 2: Java example of array indirection

The difficulty of Java, compared with other main stream programming languages, is that several threads can be running in parallel and more than one can access the array `indx`. Thus, it is possible for the elements of `indx` to be modified before the statements with the comments are executed . Even if a JVM could check all the classes loaded to make sure that no other thread could access `indx`, new classes could be loaded and invalidate such analysis.

3 Related Work

Related work can be divided into (a) techniques to eliminate array bounds checks, (b) escape analysis, (c) field analysis and related field analysis, and (d) the precursor of the present work (IBM's Ninja compiler and multi-dimensional array package). A survey of other techniques to improve the performance of Java applications can be found in [KCSL00].

Elimination of Array Bounds Checks – To the best of our knowledge, none of the published techniques, per se, and existing compilers-JVMs can (i) optimise array bounds checks in presence of indirection, (ii) are suitable for adaptive just-in-time compilation, or (iii) support multi-threading and (iv) dynamic class loading. Techniques based on theorem probers [SI77, NL98, XMR00] are too heavy weight. Algorithms based on data-flow-style have been published extensively [MCM82, Gup90, Gup93, KW95, Asu92, CG96], but for languages that do not provide multi-threading. Another technique is based on type analysis and has its application in functional languages [XP98]. Linear program solvers have also been proposed [RR00].

Bodik, Gupta and Sarkar developed the ABCD (Array Bounds Check on Demand) algorithm [BGS00] for Java. It is designed to fit the time constraints of analysing the program and applying the transformation at run-time. ABCD targets business-kind applications but its sparse representation of the program does not include information about multi-threading. Thus, the algorithm, although the most interesting for our purposes, cannot handle indirection since the origin of the problem is multi-threading. The strategies proposed in Section 4 can provide cheaply that extra information to enable ABCD to eliminate checks in the presence of indirection

Escape Analysis – In 1999, four papers were published in the same conference describing escape analysis algorithms and the possibilities to optimise Java programs by the optimal allocation of objects (heap vs. stack) and the removal of synchronization [CGS+99, BH99, Bla99, WR99]. Escape analysis tries to determine whether a object that has been created by a method, a thread, or another object escapes the scope of its creator. Escape means that another object can get a reference to the object and, thus, make it live (not garbage collected) beyond the

execution of the method, the thread, or the object that created it. The three strategies presented in Section 4 require a simple escape analysis. The strategies can only provide information (class invariants) if every instance variable does not escape the object. Otherwise, a different thread can get access to instance variables and update them, possibly breaking the desired class invariant.

Field Analysis and Related Field Analysis – Both field analysis [GRS00] and related field analysis [AR01] are techniques that look at one class at the time and try to extract as much information as possible. This information can be used, for example, for the resolution of method calls or object inlining. Related field analysis looks for relations among the instance variables of objects. Aggarwal and Randall [AR01] demonstrate how related field analysis can be used to eliminate array bounds checks for a class following the iterator pattern [GHJV95]. The strategies presented in the following section make use of the concept of field analysis. The objective is to provide a meaningful class invariant and this is represented in the instance variables (fields). However, the actual algorithms to test the relations have not been used in previous work on field analysis. The demonstration of eliminating array bounds checks given in [AR01], cannot be applied in the presence of indirection.

IBM Ninja project and multi-dimensional array package – IBM's Ninja group [MMS98, MMG99, MMG00a, MMG$^+$00b, MMG$^+$01] has focused on the Numerical Intensive Applications based on arrays in Java. Midkiff, Moreira and Snir [MMS98] developed a loop versioning algorithm so that iteration spaces of nested loops would be partitioned into regions for which it can be proved that no exceptions (null checks and array bound checks) and no synchronisations can occur. Having found these exception-free and syncronisation-free regions, traditional loop reordering transformation can be applied without violating the strict exception model.

The Ninja group design and implement a multi-dimensional array package [MMG99] to replace Java arrays so that the discovery of safe regions becomes easier. To eliminate the overhead introduced by using classes they develop *semantic expansion* [WMMG98]. Semantic expansion is a compilation strategy by which selected classes are treated as language primitives by the compiler. In their prototype static compiler, known as Ninja, they successfully implement the elimination of array bounds checks, together with semantic expansion for their multi-dimensional array package and other loop reordering transformations.

The Ninja compiler is not compatible with the specification of Java since it doesn't support dynamic class loading. The semantic expansion technique only ensures that computations that use directly the multi-dimensional array package do not suffer overheard. Although the compiler is not compatible with Java, this does not mean that the techniques that they developed could not be incorporated into a JVM. These techniques would be especially attractive for quasi-static compilers [SBMG00].

This paper extends the Ninja group's work by tackling the problem of the elimination of array bounds checks in the presence of indirection. The strategies, described in Section 4, generate classes that are incorporated into a multi-dimensional array package proposed in the Java Specification Request (JSR) 083 [JSR]. If accepted, this JSR will define the standard Java multi-dimensional array package and it is a direct consequence of the Ninja group's work.

4 Strategies

The strategies that are described in this section have a common goal: to produce a class for which JVMs can discover their invariants simply by examining the class and, thereby, derive information to eliminate array bounds checks. Each class provides the functionality of Java arrays of type `int` and, thus, can substitute them. Objects of these classes would be used instead of indirection arrays. The three strategies generate three different classes that naturally fit into a multi-dimensional array library such the one described in the JSR-083 [JSR]. Figure 3 describes the public methods that the three classes implement; the three classes extend the abstract class `IntIndirectionMultiarray1D`.

```
package multiarray;

public class RuntimeMultiarrayException extends
            RuntimeException {...}
public class UncheckedMultiarrayException extends
            RuntimeMultiarrayException {...}
public class MultiarrayIndexOutOfBoundsException
            extends RuntimeMultiarrayException {...}

public abstract class Multiarray {...}

public abstract class Multiarray1D extends Multiarray {...}
public abstract class Multiarray2D extends Multiarray {...}
public abstract class Multiarray<rank>D extends Multiarray {...}
public final class BooleanMultiarray<rank>D extends
            Multiarray<rank>D {...}
public final class <type>Multiarray<rank>D extends
            Multiarray<rank> {...}

public abstract class IntMultiarray1D extends Multiarray1D{
    public abstract int get ( int i );
    public abstract void set ( int i, int value );
    public abstract int length ();
}

public abstract class IntIndirectionMultiarray1D
            extends IntMultiarray1D {
    public abstract int getMin ();
    public abstract int getMax ();
}
```

Figure 3: Public interface of classes that substitute Java arrays of `int` and a multi-dimensional array package, `multiarray`.

Part of the class invariant that JVMs should discover is common to the three strategies. This is that the values returned by the methods `getMin` and `getMax` are always lower bounds and upper bounds, respectively, of the elements stored. This common part of the invariant can be computed, for example, using the algorithm for constraint propagation proposed in the ABCD algorithm, suitable for just-in-time dynamic compilation [BGS00].

The reflection mechanism provided by Java can interfere with the three strategies. For example, a program using reflection can access instance variables and methods without knowing the names. Even `private` instance variables and methods can be accessed! In this way a program can read from or write to instance variables and, thereby, violate the visibility rules. The circumstances under which this can happen depend on the security policy in-place for each JVM. In order to avoid this interference, hereafter the security policy is assumed to:

1. have a security manager in-place,

2. not allow programs to create a new security manager or replace the existing one (i.e. permissions `setSecurityManager` and `createSecurityManager` are not granted; see `java.lang.RuntimePermission`),

3. not allow programs to change the current security policy (i.e. permissions `getPolicy`, `setPolicy`, `insertProvider`, `removeProvider`, write to security policy files are not granted; see `java.security.SecurityPermission` and `java.io.FilePermission`), and

4. not allow programs to gain access to private instance variables and methods (i.e. permission `suppressAccessChecks` are not granted; see `java.lang.reflect.ReflectPermission`)

JVMs can test these assumptions in linear time by invoking specific methods in the `java.lang.SecurityManager` and `java.security.AccessController` objects at start-up. These assumptions do not imply any loss of generality since, to the authors' knowledge, CS&E applications do not require reflection for accessing `private` instance variables or methods. In addition, the described security policy assumptions represent good security management practice for general purpose Java programs. For a more authoritative and detailed description of Java security see [Gon98].

The first strategy is the simplest. Given that the problem arises from the parallel execution of multiple threads, a trivial situation occurs when no thread can write in the indirection array. In other words, part of the problem disappears when the indirection array is immutable: `ImmutableIntMultiarray1D` class.

The second strategy uses the synchronization mechanisms defined in Java. The objects of this class, namely `MutableImmutableStateIntMultiarray1D`, are in either of two states. The default state is the mutable state and it allows writes and reads. The other state is the immutable state and it allows only reads. This second strategy can be thought of as a way to simplify the general case into the trivial case proposed in the first strategy.

The third and final strategy takes a different approach and does not seek immutability. Only an index that is outside the bounds of an array con generate an `ArrayIndexOutOfBoundsException`: i.e. JVMs need to include explicit bounds checks. The number of threads accessing (writing/reading) simultaneously an indirection array is irrelevant as long as every element in the indirection array is within the bounds of the arrays accessed through indirection. The third class, `ValueBoundedIntMultiarray1D`, enforces that every element stored in an object of this class is within the range of zero to a given parameter. The parameter must be greater than or equal to zero, cannot be modified and is passed in with a constructor.

4.1 ImmutableIntMultiarray1D

The methods of a given class can be divided into constructors, queries and commands [Mey97]. Constructors are those methods of a class that once executed (without anomalies) create a new object of that class. In addition in Java a constructor must have the same name as its class. Queries are those methods that return information about the state (instance variables) of an object. These methods do not modify the state of any object, and can depend on an expression of several instance variables. Commands are those methods that change the state of an object (modify the instance variables).

The class `ImmutableIntMultiarray1D` follows the simple idea of making its objects immutable. Consider the abstract class `IntIndirectionMultiarray1D` (see Figure 3), the methods `get`, `length`, `getMin` and `getMax` are query methods. The method `set` is a command method and, because the class is abstract, it does not declare constructors.

Figure 4 presents a simplified implementation of the class `ImmutableIntMultiarray1D`. In order to make `ImmutableIntMultiarray1D` objects immutable, the command methods are implemented simply by throwing an `UncheckedMultiarrayException`.[1] Obviously, the query methods do not modify any instance variable. The instance variables (`data`,[2] `length`, `min` and `max`) are declared as `final` instance variables and every instance variable is initialised by each constructor.

```
public final class ImmutableIntMultiarray1D extends
            IntIndirectionMultiarray1D {
    private final int data[];
    private final int length;
    private final int min;
    private final int max;

    public ImmutableIntMultiarray1D ( int values[] ) {
        int temp, auxMin, auxMax;
        auxMin = 0;  auxMax = 0;
        length = values.length;  data = new int [length];
        for (int i = 0; i < length; i++){
            temp = values[i];  data[i] = temp;
            if ( auxMin > temp ) auxMin = temp;
            if ( auxMax < temp ) auxMax = temp;
        }
        max = auxMax;  min = auxMin;
    }
    public int get ( int i ) {
        if ( i >= 0 && i < length )
            return data[i];
        else throw new MultiarrayIndexOutOfBoundsException();
    }
    public void set ( int i , int value ) {
        throw new UncheckedMultiarrayException();
    }
    public int length () { return length; }
    public int getMin () { return min; }
    public int getMax () { return max; }
}
```

Figure 4: Simplified implementation of class `ImmutableIntMultiarray1D`.

Note that the only statements (bytecodes in terms of JVMs) that write to the instance variables occur in constructors and that the instance variables are `private` and `final`. These two conditions are almost enough to derive that every object is immutable.

Another condition is that those instance variables whose type is not primitive do not escape the scope of the class: escape analysis. In other words, these instance variables are created by any method of the class, none of the methods returns a reference to any instance variable and they are declared as `private`. The life span of these instance variables does not exceed that of

[1] `UncheckedMultiarrayException` inherits from `RuntimeException` – is an unchecked exception – and, thus, methods need to neither include it in their signature nor provide try-catch clauses.

[2] The declaration of an instance variable of type array as `final` indicates that once an array has been assigned to the instance variable then no other assignment can be applied to that instance variable. However, the elements of the assigned array can be modified without restriction.

its creator object.

For example, the instance variable `data` escapes the scope of the class if a constructor is implemented as shown in Figure 5. It also escapes the scope of the class if the non-`private` method `getArray` (see Figure 5) is included in the class. In both cases, any number of threads can get a reference to the instance variable `array` and modify its content (see Figure 6).

```
public final class ImmutableIntMultiarray1D extends
            IntIndirectionMultiarray1D {
...
    public ImmutableIntMultiarray1D ( int values[] ) {
        int temp, auxMin, auxMax;
        length = values.length;  data = values;
        for (int i = 0; i < length; i++){
            temp = values[i];
            if ( auxMin > temp ) auxMin = temp;
            if ( auxMax < temp ) auxMax = temp;
        }
        max = auxMax;  min = auxMin;
    }
...
    public Object getArray () { return data; }
...
}
```

Figure 5: Methods that enable the instance variable `array` to escape the scope of the class `ImmutableIntMultiarray1D`.

```
public class ExampleBreakImmutability {
    public static void main ( String args[] ) {
        int fib[] = {1, 1, 2, 3, 5, 8};
        ImmutableIntMultiarray1D indx = new
                ImmutableIntMultiarray1D( fib );
        fib[0] = -1;
        System.out.println( indx.get( 0 ) ); // Output: -1
        int aux[] = (int{}) indx.getArray();
        aux[0] = -8;
        System.out.println( indx.get( 0 ) ); // Output: -8
    }
}
```

Figure 6: An example program modifies the content of the instance variable `data`, member of every object of class `ImmutableIntMultiarray1D`, using the method and the constructor implemented in Figure 5.

Consider an algorithm that checks:

- that only bytecodes in constructors write to any instance variable,
- that every instance variable is `private` and `final`, and
- that any instance variable whose type is not primitive does not escape the class.

Such an algorithm can determine whether any class produces immutable objects and it is of $O(\#b \times \#iv)$ complexity, where $\#b$ is the number of bytecodes and $\#iv$ is the number of instance variables. Hence, the algorithm is suitable for just-in-time compilers. Further with the

JSR-083 as the Java standard multidimensional array package, JVMs can recognise the class and produce the invariant without any checking.

The constructor provided in the class `ImmutableIntMultiarray1D` is inefficient in terms of memory requirements. This constructor implies that at some point during execution a program would consume double the necessary memory space to hold the elements of an indirection array. This constructor is included mainly for the sake of clarity. A more complete implementation of this class provides other constructors that read the elements from files or input streams.

The implementation of the method `set` includes a specific test for the parameter `i` before accessing the instance variable `data`. The test ensures that accesses to `data` are not out the bounds. Tests of this kind are implemented in every method `set` and `get` in the subsequent classes. Hereafter, and for clarity, the tests are omitted in the implementations of subsequent classes.

4.2 MutableImmutableStateIntMultiarray1D

Figure 7 presents a simplified and non-optimised implementation of the second strategy. The idea behind this strategy is to ensure that objects of class `MutableImmutableStateIntMultiarray-1D` can be in only two states:

Mutable state – Default state. The elements stored in an object of the class can be modified and read (at the user's own risk).

Immutable state – The elements stored in an object of the class can be read but not modified.

The strategy relies on the synchronization mechanisms provided by Java to implement the class. Every object in Java has associated a lock. The execution of a *synchronized method*[3] or *synchronized block*[4] is a critical section. Given an object with a synchronized method, before any thread can start the execution of the method it must first acquire the lock of that object. Upon completion the thread releases the lock. The same applies to synchronized blocks. At any point in time at most one thread can be executing a synchronized method or a synchronized block for the given object.

The Java syntax and the standard Java API do not provide the concept of acquiring and releasing an object's lock. Thus, a Java application does not contain special keywords or invokes a method of the standard Java API to access the lock of an object. These concepts are part of the specification for the execution of Java applications. Further details about multi-threading in Java can be found in [JSGB00, Lea99].

Consider an object `indx` of class `MutableImmutableStateIntMultiarray1D`. The implementation of this class (see Figure 7) enforces that `indx` starts in the mutable state. The state is stored in the `boolean` instance variable `isMutable` and its value is kept equivalent to the boolean expression `(reader == null)`. For the mutable state, the implementations method `get` and `set` are as expected.

The object `indx` can only change its state when a thread invokes its synchronized method `passToImmutable`. When the state is mutable `indx` changes its instance variable `isMutable` to `false` and stores the thread that executed the method in the instance variable `reader`. When

[3] A synchronized method is a method whose declaration contains the keyword `synchronized`.

[4] A synchronized block is a set of consecutive statements in the body of a method not declared as synchronized which are surrounded by the clause `synchronized (object) {...}` .

```
public final class MutableImmutableStateIntMultiarray1D
            extends IntIndirectionMultiarray1D {
    private Thread reader;
    private final int data[];
    private final int length;
    private int min;
    private int max;
    private boolean isMutable;

    public MutableImmutableStateIntMultiarray1D ( int length ) {
        this.length = length;  data = new int [length];
        reader = null;  isMutable = true;
        min = 0;  max = 0;
    }
    public int get ( int i ) { return data[i]; }
    public synchronized void set ( int i , int value ) {
        while ( !isMutable ){
            try{ wait(); }
            catch (InterruptedException e) {
                throw new UncheckedMultiarrayException();
            }
        }
        array[i] = value;
        if ( min > value ) min = value;
        if ( max < value ) max = value;
    }
    public int length () { return length; }
    public synchronized int getMin () { return min; }
    public synchronized int getMax () { return max; }
    public synchronized void passToImmutable () {
        while ( !isMutable ) {
            try { wait(); }
            catch (InterruptedException e) {
                throw new UncheckedMultiarrayException();
            }
        }
        isMutable = false;  reader = Thread.currentThread();
    }
    public synchronized void returnToMutable () {
        if ( reader == Thread.currentThread() ) {
            reader = null;  isMutable = true;  notify();
        }
        else throw new UncheckedMultiarrayException();
    }
}
```

Figure 7: Simplified implementation of class `MutableImmutableStateIntMultiarray1D`.

the state is immutable, the thread executing the method stops[5] until the state becomes mutable

[5] Explanation of the *wait/notify* thread communication mechanism to clarify what *stops* means in this sentence. The `wait` and `notify` methods are part of the standard Java API and both are members of the class `java.lang.Object`. In Java every class is a subclass (directly or indirectly) of `java.lang.Object` and, thus, every object inherits inherits the methods `wait` and `notify`. Both methods are part of a communication mechanism for Java threads.

For example, the thread executing the method `passToImmutable` being `indx` in immutable state. The thread starts executing the synchronized method after acquiring the lock of `indx`. After checking the state, it needs to *wait* until the state of `indx` is mutable. The thread, itself, cannot force the state transition. It needs to *wait* for another thread to provoke that transition. The first thread stops execution by invoking the method `wait` in `indx`. This method makes the first thread release the lock of `indx`, wait until a second thread invokes the method `notify` in `indx` and then reacquire the lock prior to return from `wait`. Several threads can be waiting in `indx` (i.e. have invoked `wait`), but only one thread is notified the second threads invokes `notify` in `indx`. Further

and then proceeds as explained for the mutable state.

Once `indx` is in the immutable state, the `get` method is implemented as expected while the `set` method cannot modify the elements of `data` until `indx` returns to the mutable state.

The object `indx` returns to the mutable state when the same thread that successfully provoked the transition mutable-to-immutable invokes in `indx` the synchronized method `returnToMutable`. When the transition is completed, this thread notifies other threads waiting in `indx` of the state transition.

Given the complexity of matching the locking-wait-notify logic with the statements (bytecodes) that write in the instance variables, the authors consider that JVMs will not incorporate tests for this kind of class invariant. Thus, the possibility is that JVMs recognise the class as being part of the standard Java API and automatically provide the class invariant.

Note that a requirement for proving the class invariant again is that the instance variables of the class `MutableImmutableStateIntMultiarray1D` do not escape the scope of the class.

4.3 ValueBoundedIntMultiarray1D

Figure 8 presents a simplified implementation of the third strategy. The implementation of this class, `ValueBoundedIntMultiarray1D`, ensures that its objects can only store elements greater or equal than zero and less than or equal to a parameter. This parameter, `upperBound`, is passed to every object by the constructor and cannot be modified.

```
public final class ValueBoundedIntMultiarray1D extends
            IntIndirectionMultiarray1D {
    private final int data[];
    private final int length;
    private final int upperBound;
    private final int lowerBound = 0;

    public ValueBoundedIntMultiarray1D ( int length,
                int upperBound ) {
        this.length = length;
        this.upperBound = upperBound;
        data = new int [length];
    }
    public int get ( int i ) { return data[i]; }
    public void set ( int i , int value ) {
        if ( value >= lowerBound && value <= upperBound )
            data[i] = value;
        else throw new UncheckedMultiarrayException();
    }
    public int length () { return length; }
    public int getMin () { return lowerBound; }
    public int getMax () { return upperBound; }
}
```

Figure 8: Simplified implementation of class `ValueBoundedIntMultiarray1D`.

The implementation of the method `get` is the same as in the previous strategies. The implementation of the method `set` includes a test that ensures that only elements in the range [0..`upperBound`] are stored. The methods `getMin` and `getMax`, in contrast with previous classes, do not return the actual minimum and maximum stored elements, but lower (zero) and upper bounds.

information about threads in Java and the wait/notify mechanism can be found in [JSGB00, Lea99].

The tests that JVMs need to perform to extract the class invariant include the escape analysis for the instance variables of the class (described in Section 4.1) and, as mentioned in the introduction of Section 4 with respect to the common part of the invariant, the construction of constraints using data flow analysis for the modification of the instance variables. As with previous classes, JVMs can also recognise the class and produce the invariant without any checking.

4.4 Usage of the Classes

This section revisits the matrix-vector multiplication kernel where the matrix is sparse and stored in COO format (see Figure 1) in order to illustrate how the three different classes can be used. Figure 10 presents the same kernel but, follows the recommendations of the Ninja group [MMG00a, MMG+00b, MMG+01] to facilitate loop versioning (the statements inside the for-loop do not produce `NullPointerExceptions` nor `ArrayIndexOutOfBoundsExceptions` and do not use synchronization mechanisms). This new implementation checks that the parameters are not `null` and that the accesses to `indx` and `jndx` are within the bounds. These checks are made prior to the execution of the for-loop. If the checks fail then none of the statements in the for-loop is executed and the method terminates by throwing an exception. The implementation, for the sake of clarity, omits the checks to generate loops free of aliases. For example, the variables `fib` and `aux` are aliases of the same array according to their declaration in Figure 9.

```
int fib[] = {1, 1, 2, 3, 5, 8};  int aux[] = fib;
aux[0] = -8;
System.out.println("Are they aliases?: " + fib[0] == -8 );
// Output: Are they aliases? true
```

Figure 9: An example of array aliases.

Note that checks to ensure that accesses to `vectorY` and `vectorX` are within bounds will require traversing completely local copies of the arrays `indx` and `jndx`. Local copies of the arrays are necessary, since both `indx` and `jndx` escape the scope of this method. This makes possible, for example, that another thread can modify their contents after the checks but before the execution of the for-loop. The creation of local copies is a memory inefficient approach and the overhead of copying and checking element by element for the maximum and minimum is similar to (or greater than) to the overhead of the explicit checks.

Figure 11 presents the implementations of `mvmCOO` using the three classes described in previous sections. The checks for nullity are replaced with a line comment indicating where the checks would be placed. Only the implementation of `mvmCOO` for `ImmutableIntMultiarray1D` is complete. The others include comments where the statements of the complete implementation would appear. Due to the class invariants, the implementations include checks for accesses to `vectorY` and `vectorX`. When the checks are passed, JVMs find a loop free of `ArrayIndexOutOfBoundsException` and `NullPointerException`.

The implementations of `mvmCOO` for `ImmutableIntMultiarray1D` and `ValueBoundedIntMultiarray1D` are identical; the same statements but different method signature. The implementation for `MutableImmutableStateIntMultiarray1D` builds on the previous implementations, but it first needs to ensure that the objects `indx` and `jndx` are in immutable state. The implementation also requires that these objects are returned to the state mutable upon completion or abnormal interruption.

```
public class ExampleSparseBLASWithNinja {
   // y = A*x + y
   public static void mvmCOO ( int indx[], int jndx[],
               double value[], double vectorY[],
               double vectorX[] ) throws SparseBLASException {
       if ( indx != null && jndx != null && value != null &&
            vectorY != null && vectorX != null &&
            indx.length >= value.length &&
            jndx.length >= value.length ) {

           for (int k = 0; k < value.length; k++)
             vectorY[ indx[k] ] += value[k] * vectorX[ jndx[k] ];
       }
       else throw new SparseBLASException();
   }
}
```

Figure 10: Sparse matrix-vector multiplication using coordinate storage format and Ninja's group recommendations.

5 Experiments

This section reports two sets of experiments. The first set determines the overhead of array bounds checks for a CS&E application with array indirection. The second set, also for a CS&E application, determines the overhead of using the classes proposed in Section 4 instead of accessing directly Java arrays. In other words, the experiments seek an experimental lower bound for the performance improvement that can be achieved when array bounds checks are eliminated and upper bound of the overhead due to using the classes introduced in Section 4. A lower bound because array bounds checks together with the strict exception model specified by Java are inhibitors of other optimising transformations (see Ninja project [MMG00a, MMG+00b, MMG+01]) that can improve further the performance. An upper bound because techniques, such as semantic expansion [WMMG98] or those proposed by the authors for optimising OoLaLa [LFG02b, LFG02a], can remove the source of overhead of using a class instead of Java arrays.

The considered example is matrix-vector multiplication where the matrix is in COO format (mvmCOO). This matrix operation is the kernel of iterative methods [GvL96] for solving systems of linear equations or determining the eigenvalues of a matrix. The iterative nature of these methods implies that the operation is executed repeatedly until sufficient accuracy is achieved or an upper bound on the number of iterations is reached. The experiments consider an iterative method that executes mvmCOO 100 times.

Results are reported for four different implementations (see Figures 10 and 11) of mvmCOO. The four implementations of mvmCOO are derived using only Java arrays (JA implementation), or objects of class `ImmutableIntMultiarray1D` (I-MA implementation), class `MutableImmutableStateIntMultiarray1D` (MI-MA implementation), and class `ValueBoundedIntMultiarray1D` (VB-MA implementation).

The experiments consider three different square sparse matrices from the Matrix Market collection [BPR+97]. The three matrices are: utm5940 (size 5940 and 83842 nonzero elements), s3rmt3m3 (size 5357, 106526 entries and symmetric), and s3dkt3m2 (size 90449, 1921955 entries and symmetric). The implementations do not take advantage of symmetry and, thus, s2rmt3m3 and s3dkt3m2 are stored and operated on using all their nonzero elements. The vectors `vectorX` and `vectorY` (see implementations in Figures 10 and 11) are initialised with random numbers

```
public class KernelSparseBLAS {
    // y = A*x + y
    public static void mvmCOO ( ImmutableIntMultiarray1D indx,
                ImmutableIntMultiarray1D jndx, double value[],
                double vectorY[], double vectorX[] )
                throws SparseBLASException {
        if ( // nullChecks &&
             indx.length() >= value.length &&
             jndx.length() >= value.length &&
             indx.getMin() >= 0 &&
             jndx.getMin() >= 0 &&
             indx.getMax() < vectorY.length &&
             jndx.getMax() < vectorX.length ) {

            for (int k = 0; k < value.length; k++)
                vectorY[ indx.get( k ) ] += value[k] *
                            vectorX[ jndx.get( k ) ];
        }
        else throw new SparseBLASException();
    }
    // y = A*x + y
    public static void mvmCOO (
                MutableImmutableStateIntMultiarray1D indx,
                MutableImmutableStateIntMultiarray1D jndx,
                double value[], double vectorY[],
                double vectorX[] ) throws SparseBLASException {
        if ( indx != null && jndx != null ) {
            indx.passToImmutable();
            if ( indx != jndx ) jndx.passToImmutable();
  // idem implementation of mvmCOO for ImmutableIntMultiarray1D
  // but before the throw statement include returnToMutable for
  // indx and/or jndx
            indx.returnToMutable();
            if ( indx != jndx ) jndx.returnToMutable();
        }
        else throw new SparseBLASException();
    }
    // y = A*x + y
    public static void mvmCOO (
                ValueBoundedIntMultiarray1D indx,
                ValueBoundedIntMultiarray1D jndx,
                double value[], double vectorY[],
                double vectorX[] ) throws SparseBLASException {
        // idem mvmCOO for ImmutableIntMultiarray1D
    }
}
```

Figure 11: Sparse matrix-vector multiplication using coordinate storage format, Ninja's group recommendations and the classes described in Figures 4, 7 and 8.

according to a uniform distribution with values ranging between 0 and 5, and with zeros, respectively.

The experiments are performed on a 1 GHz Pentium III with 256 Mb running Windows 2000 service pack 2, Java 2 SDK 1.3.1_02 Standard Edition and Jove (an static Java compiler [Jov]) version 2.0 associated with the Java 2 Runtime Environment Standard Edition 1.3.0. The programs are compiled with the flag `-O` and (the Hotspot JVM is) executed with the flags `-server -Xms128Mb -Xmx256Mb`. Jove is used with the following two configurations. The first configuration, hereafter refer to as Jove, creates an executable Windows program using default optimisation parameters and optimization level 1; the lowest level. The second configuration, namely Jove-nochecks, is the same configuration plus a flag that eliminates every array bounds

checks.

Each experiment is run 20 times and the results shown are the average execution time in seconds. The timers are accurate to the millisecond. For each run, the total execution time of mvmCOO is recorded. Recall that the execution of one experiment implies a program with a loop that executes 100 times mvmCOO.

Table 1 presents the execution times for the JA implementation compiled with the two configurations described for the Jove compiler. The fourth column gives the overhead induced by array bounds checks. This overhead is between 9.03 and 9.83 percentage of the execution time.

Matrix	Jove (s.)	Jove-nochecks (s.)	Overheard (%)
utm5940	0.704	0.640	9.03
s3rmt3m3	1.700	1.538	9.51
s3dkt3m2	33.99295	30.656	9.83

Table 1: Average times in seconds for the JA implementation of mvmCOO.

Table 2 presents the execution times for the four implementations of mvmCOO. In this case, the JVM is the Hotspot server, part of the described Java SDK environment. Table 3 gives the overheads (%) for each of the implementations proposed in Section 4 with respect to the JA implementation. The I-MA implementation produces the greatest overheads for all the matrices. In the middle sits the IM-MA implementation, while the BV implementation always produces the smallest overhead.

Matrix	JA (s.)	I-MA (s.)	IM-MA (s.)	VB-MA (s.)
utm5940	0.545	0.600	0.601	0.579
s3rmt3m3	1.331	1.527	1.502	1.460
s3dkt3m2	23.168	26.477	25.971	25.280

Table 2: Average times in seconds for the four different implementations of mvmCOO.

Matrix	I-MA (%)	IM-MA (%)	VB-MA (%)
utm5940	10.11	10.21	6.13
s3rmt3m3	14.72	12.83	9.64
s3dkt3m2	14.28	12.10	9.12

Table 3: Overhead (%) for the implementations of mvmCOO using the proposed classes with respect to the JA implementation.

Although this section omits them, the second set of experiments run also on a Linux/-Pentium and Solaris/Sparc platforms with the equivalent Java environment. These omitted results are consistent with those presented below.

6 Discussion

Thus far, the development has assumed that the three classes can be incorporated in the multi-dimensional array package proposed by the JSR-083. The following paragraphs try to determine whether one of the classes is the "best" or whether the package needs more than one class.

Consider first the class `MutableImmutableStateIntMultiarray1D` and an object `indx` of this class. In order to obtain the benefit of array bounds checks elimination when using `indx` as an indirection array, programs need to follow these steps:

1. to change the state of `indx` to immutable,
   `indx.passToImmutable();`

2. to execute any other actions (normally inside loops) that acess the elements stored in `indx`, and
   `...foo[ indx.get(i) ]...`

3. to change the state of `indx` back to mutable.
   `indx.returnToMutable;`

An example of these steps is the implementation of matrix-vector multiplication using class `MutableImmutableStateIntMultiarray1D` (see Figure 11).

If the third step is omitted, possibly accidentally, the indirection array `indx` becomes useless for the purpose of array bounds checks elimination. Other threads (or a thread that abandoned the execution without adequate clean up) would be stopped waiting indefinitely for the notification that `indx` has returned back to the mutable state.

Another undesirable situation can arise when several threads are executing in parallel and, at least, two threads need `indx` and `jndx` (another object of the same class) at the same time. Depending on the order of execution or both being aliases of the same object a deadlock can occur.

One might think that these two undesirable situations could be overcome by modifying the implementation of the class so that it maintains adequately a list of thread readers instead of just one thread reader. However, omission of the third step now leads to starvation of writers.

Given that these undesirable situations are not inherent in the other two strategies-classes, and that the experiments of Section 5 do not show a performance benefit in favor of class `MutableImmutableStateIntMultiarray1D`, the decision is to disregard this class.

The discussion now looks at scenarios in CS&E applications where the two remaining classes can be used. In other words the functionality of classes `ImmutableIntMultiarray1D` and `ValueBoundedIntMultiarray1D` are contrasted with the functionality requirements of CS&E applications.

Remember that both classes, although they have the same public interface, provide different functionality. As the name suggests, class `ImmutableIntMultiarray1D` implements the method `set` without modifying any instance variable; it simply throws an unchecked exception (see Figure 4). On the other hand, class `ValueBoundedIntMultiarray1D` provides the expected functionality for this method as long as the parameter `value` of the method is positive and less than or equal to the instance variable `upperBound` (see Figure 8).

In various scenarios of CS&E applications, such as LU-factorisation of dense and banded matrices with pivoting strategies, and Gauss elimination for sparse matrices with fill-in , applications need to update the content of indirection arrays [GvL96]. For example, fill-in means that new nonzero elements are created where zero elements existed before. Thus, the Gauss elimination algorithm creates new nonzero matrix elements progressively. Assuming that the matrix is stored in COO format, the indirection arrays `indx` and `jndx` need to be updated with the row and column indices, respectively, for each new nonzero element.

Given that in other CS&E applications indirection arrays remain unaltered after initialisation, the only reason for including class `ImmutableIntMultiarray1D` would be performance.

However the performance evaluation reveals that this is not the case. Therefore, the conclusion is to incorporate only the class `ValueBoundedIntMultiarray1D` into the multi-dimensional array package.

7 Conclusions

Array indirection is ubiquitous in CS&E applications. With the specification of array accesses in Java and with current techniques for eliminating array bounds checks, applications with array indirection suffer the overhead of explicitly checking each access through an indirection array.

Building on previous work of IBM's Ninja and Jalapeño groups, three new strategies to eliminate array bounds checks in the presence of indirection are presented. Each strategy is implemented as a Java class that can replace Java indirection arrays of type `int`. The aim of the three strategies is to provide extra information to JVMs so that array bounds checks in the presence of indirection can be removed. For normal Java arrays of type `int` this extra information would require access to the whole application (i.e. no dynamic loading) or be heavy weigh. The algorithm to remove the array bounds checks is a combination of loop versioning (as used by the Ninja group [MMS98, MMG00a, MMG+00b, MMG+01]), escape analysis [CGS+99] and construction of constraints based on data flow analysis (ABCD algorithm [BGS00]).

The experiments have evaluated the performance benefit of eliminating array bound checks in the presence of indirection. The overhead has been estimated between 9.03 and 9.83 percentage of the execution time. The experiments have also evaluated the overhead of using a Java class to replace Java arrays on a off-the-self JVM. This overhead varies for each class, but it is between 6.13 and 14.72 percentage of the execution time.

The evaluation of the three strategies also includes a discussion of their advantages and disadvantages. Overall, the third strategy, class `ValueBoundedIntMultiarray1D`, is the best. It takes a different approach by not seeking the immutability of objects. The number of threads accessing an indirection array at the same time is irrelevant as long as every element in the indirection array is within the bounds of the arrays accessed through indirection. The class enforces that every element stored in a object of the class is between the values of zero and a given parameter. The parameter must be greater than or equal to zero, cannot be modified and is passed in with a constructor.

The remaining problem is the overhead for programs using the class `ValueBoundedIntMultiarray1D` instead of Java arrays. The authors have proposed a set of transformations [LFG02a, LFG02b] for their Java library OoLaLa [Luj99, LFG00]. A paper that demonstrates that the same set of transformations designed for OoLaLa is enough to optimise away the overhead of using the class `ValueBoundedIntMultiarray1D` is in preparation. Future work concentrates on developing an algorithm suitable for just-in-time dynamic compilation to determine when to apply the set of transformations.

References

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivassan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[AR01] Aneesh Aggarwal and Keith H. Randall. Related field analysis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation – PLDI'01*, pages 214–20, 2001.

[Asu92] Jonathan M. Asuru. Optimization of array subscript range. *ACM Letters on Programming Languages and Systems*, 1(2):109–118, 992.

[BGS00] Rastilav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the ACM Conference on Programming Language Design and Implementation – PLDI 2000*, pages 321–333, 2000.

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Application – OOPSLA'99*, pages 35–46, 1999.

[Bla99] Bruno Blanchet. Escape analysis for object oriented languages. application to Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Application – OOPSLA'99*, pages 20–34, 1999.

[BMPP01] Ronald F. Boisvert, José E. Moreira, Michale Philippsen, and Roldan Pozo. Java and numerical computing. *IEEE Computing in Science and Engineering*, 3(2):18–24, 2001.

[BPR+97] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard Barrett, and Jack J. Dongarra. The Matrix Market: A web resource for test matrix collections. In *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137. Chapman & Hall, 1997. Matrix Market http://math.nist.gov/MatrixMarket/.

[BSPF01] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the ACM 2001 Java Grande/ISCOPE Conference*, pages 97–105, 2001.

[CG96] Wei-Ngan Chin and Eak-Khoon Goh. A reexamination of "optimization of array subscript range checks". *ACM Transactions on Programming Languages and Systems*, 17(2):217–227, 1996.

[CGS+99] Jong-Deok Choi, Manish Gupta, Maurico Serrano, Bugranam C. Sreedhar, and Samuel Midkiff. Escape analysis for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications – OOPSLA'99*, pages 1–19, 1999.

[CLS00] Michał Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM Conference on Programming Language Design and Implementation – PLDI 2000*, pages 13–26, 2000.

[EPC] EPCC. The Java grande forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/.

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Sofware – Practice and Experience*, 30(3):199–232, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison Wesley, 1995.

[Gon98] Li Gong. *Java 2 Platform Security Architecture.* Sun Microsystems, 1998. Available at http://java.sun.com/j2se/1.3/docs/.

[GRS00] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Fiel analysis: Getting useful and low-cost interprocedural information. In *Proceedings of the ACM Conference on Programming Language Design and Implementation – PLDI'00*, pages 334–344, 2000.

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the ACM Conference on Programming Language Design and Implementation – PLDI'90*, pages 272–282, 1990.

[Gup93] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, 1993.

[GvL96] Gene H. Golub and Charles F. van Loan. *Matrix Computations.* John Hopkins University Press, 3^{th} edition, 1996.

[Jav98] Java Grande Forum. *Making Java Work for High-End Computing*, November 1998. Available at http://www.javagrande.org/reports.htm.

[Jav99] Java Grande Forum. *Interim Java Grande Forum Report*, June 1999. Available at http://www.javagrande.org/reports.htm.

[Joh82] R. L. Johnston. *Numerical Methods: A Sofware Approach.* John Wiley and Sons, 1982.

[Jov] Jove: Optimizing native compiler for java technology. http://www.instantiations.com/jove/.

[JSGB00] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification.* The Java Series. Addison-Wesley, second edition, 2000.

[JSR] Multiarray package. http://jcp.org/jsr/detail/083.jsp.

[KCSL00] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David Lilja. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys*, 32(3):213–240, 2000.

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the ACM Conference on Programming Language Design and Implementation – PLDI'95*, pages 270–278, 1995.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principle and Patterns.* The Java Series. Addison-Wesley, second edition, 1999.

[LFG00] Mikel Luján, T. L. Freeman, and John R. Gurd. OoLaLa: an object oriented analysis and design of numerical linear algebra. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications – OOPSLA'00*, pages 229–252, 2000.

[LFG02a] Mikel Luján, T. L. Freeman, and John R. Gurd. Compiler transformations for optimizing OOLALA. In *Submitted to the International Conference on Supercomputing*, 2002.

[LFG02b] Mikel Luján, T. L. Freeman, and John R. Gurd. OoLaLa: Transformations for implementations of matrix operations at high abstracion levels. Technical report, Centre for Novel Computing, University of Manchester, 2002.

[Luj99] Mikel Luján. Object oriented linear algebra. Master's thesis, Department of Computer Science, University of Manchester, December 1999.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, second edition, 1999.

[MCM82] Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. In *Proceedings of a Symposium on Compiler Optimization*, pages 114–119, 1982.

[Mey97] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, 2^{th} edition, 1997.

[MMG99] José E. Moreira, Samuel P. Midkiff, and Manish Gupta. A standard Java array package for technical computing. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[MMG00a] José E. Moreira, Samuel P. Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, 2000.

[MMG+00b] José E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Marc Snir, and Richard D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.

[MMG+01] José E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Peng Wu, and George Almasi. The Ninja project. *Communications of the ACM*, 44(10):102–109, 2001.

[MMS98] Samuel P. Midkiff, José E. Moreira, and Marc Snir. Optimizing array reference checking in Java programs. *IBM Systems Journal*, 37(3):409–453, 1998.

[MS99] Gilles Muller and Ulrik Pagh Schultz. Harissa: A hybrid approach to java execution. *IEEE Sofware*, 16(2):44–51, 1999.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM Conference on Programmin Language Design and Implementation – PLDI'98*, pages 333–344, 1998.

[PBG+00] Michael Philippsen, Ronald F. Boisvert, Valdimir S. Getov, Roldan Pozo, José Moreira, Dennis Gannon, and Geoffrey C. Fox. JavaGrande – high performance computing with Java. In *PARA2000*, volume 1947 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[PM] Roldan Pozo and Bruce Miller. Scimark 2.0. http://math.nist.gov/scimark2/.

[PTH+97] Todd A. Proebting, Gregg Townsend, John H. Hartman, Patric Bridges, Scott A. Watterson, and Tim Newsham. Toba: Java for applications a way ahead of time (WAT) compiler. In *Proceedings of the USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 41–53, 1997.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot server compiler. In *Proceedings of the USENIX Symposium on Java Virtual Machine Research and Technology*, 2001.

[RR00] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointer, array indices, and accessed memory regions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation – PLDI 2000*, pages 182–195, 2000.

[SBMG00] Mauricio J. Serrano, Rajesh Bordawekar, Samuel P. Midkiff, and Manish Gupta. Quicksilver: a quasi-static compiler for java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Application – OOPSLA'00*, pages 66–82, 2000.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Conference Record of the ACM Symposium of Principles of Programming Languages*, pages 132–143, 1977.

[SOT+00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[Thi02] George K. Thiruvathukal. Java at middle age: Enabling Java for computational science. *IEEE Computing in Science and Engineering*, 4(1):74–84, 2002.

[WMMG98] Peng Wu, Samuel P. Midkiff, José E. Moreira, and Manish Gupta. Improving Java performance through semantic inlining. Technical Report 21313, IBM Research Division, 1998.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Application – OOPSLA'99*, pages 187–206, 1999.

[XMR00] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of the ACM Conference on Programming Language Design and Implementation – PLDI'00*, pages 70–82, 2000.

[XP98] Hongwei Xi and Frank Pfenning. Elinating array bound checking through dependent types. In *Proceedings of the ACM Conference on Programming Language Design and Implementation – PLDI'98*, pages 249–257, 1998.