McGill University
School of Computer Science
Sable Research Group

A Comprehensive Approach to Array Bounds Check Elimination for Java

Sable Technical Report No. 2000-4

Feng Qian
Laurie Hendren

Sable Research Group
McGill University
3480 University Street
Montreal, Quebec
Canada H3A 2A7
[fqian,hendren]@cs.mcgill.ca

Clark Verbrugge

IBM Toronto Lab
1150 Eglinton Ave E.
Toronto, Ontario
Canada M3C 1H7

clarkv@ca.ibm.com

November 17, 2000

Abstract

This paper reports on a new approach to eliminating array bounds checks in Java. Our approach is based upon a flow-sensitive intraprocedural analysis called *variable constraint analysis* (VCA). This analysis builds a small constraint graph for each important point in a method, and then uses the information encoded in the graph to infer the relationship between array index expressions and the bounds of the array. Using VCA as the base analysis, we also show how two further analyses can improve the results of VCA. *Array field analysis* is applied on each class and provides information about some arrays stored in fields, while *rectangular array analysis* is an interprocedural analysis to approximate the shape of arrays, and is useful for finding rectangular (non-ragged) arrays.

We have implemented all three analyses using the Soot bytecode optimization/annotation framework and we transmit the results of the analysis to virtual machines using class file attributes. We have modified the Kaffe JIT, and IBM's High Performance Compiler for Java (HPCJ) to make use of these attributes, and we demonstrate significant speedups.

1 Introduction

The Java programming language is becoming increasingly popular for the implementation of a wide variety of application programs, including loop-intensive programs that use arrays. Java compilers, as well as compilers for other languages such as Ada and Fortran, translate high-level programs to Java bytecode and this bytecode is either executed by a Java virtual machine (usually including a JIT compiler), or it is compiled by an ahead-of-time compiler to native code. In either case, the Java specifications require that exceptions be raised for any array access in which the array index expression evaluates to an index out of bounds.

A naive JIT or ahead-of-time compiler inserts checks for each array access, which is clearly inefficient. These checks cause a program to execute slower due to both direct and indirect effects of the bounds check. The direct effect is that the bounds check is usually implemented via a comparison instruction, and thus each array access has this additional overhead. The indirect effect is that these checks also limit further optimizations because the Java virtual machine specification requires precise exception handling. This limits code movement and also limits many effective loop transformations which are commonly used in high-performance C and Fortran compilers [13]. Furthermore, this same precise exception requirement limits program transformations that optimize the runtime checks. For example, checks cannot be moved to earlier program points if this changes the exception behaviour of the program.

The problem of eliminating array bounds checks has been studied for other languages and static analyses have been shown to be quite successful. However, array bounds check analysis in Java faces several special challenges. Firstly the length of an array is determined dynamically, when the array is allocated, and thus the length (or upper bound) of the array may not be a known constant. Secondly, arrays in Java are objects, and these objects may be passed as references through method calls, or may be stored as a field of some object. Thus, there may be a non-obvious correspondence between the allocation site of an array and the accesses to the array. Thirdly, multi-dimensional arrays in Java are not necessarily rectangular, and so reasoning about the lengths of higher dimensions is not simple. Finally, techniques that require transforming the program or inserting checks at other earlier program points are not as applicable in Java as in other languages with less strict semantics about exceptions.

This paper describes a flow-sensitive, intraprocedural algorithm called *variable constraint analysis* (VCA for short) that can prove that many array references are safe, without transforming the original program. The algorithm builds a constraint graph for each array reference, and then uses the graph to infer the relationship between the index of the array reference and the array's length. The algorithm was designed carefully to take advantage of the fact that variables used in index expressions often have very short lifetimes, and thus building graphs for only live variables of interest leads to very small graphs. Further, we tuned the worklist algorithm to reduce the number of iterations. As a result, the actual running time is linear in the size of the method being analyzed.

We have improved the base VCA algorithm using two additional analyses: *array field analysis* is applied on each class and provides information about some arrays stored in fields, while *rectangular array analysis* is an interprocedural analysis to approximate the shape of arrays.

All three analyses have been implemented using the Soot bytecode optimization framework, but could be easily implemented in other compilers with good intermediate representations. The Soot framework converts bytecode from class files into a typed 3-address representation called Jimple, and the analysis is implemented on this representation. In order to convey the results of the analysis to virtual machines we use the tagging/attributing capabilities of Soot to tag each array access instruction to indicate if the lower bound and/or upper bound checks can be eliminated. The Soot framework then produces bytecode output, with the tag information stored in the attributes section of the class files. Virtual machines or ahead-of-time bytecode-to-nativecode compilers can then use these attributes to avoid emitting bounds checks based on the attributes. We have instrumented both the Kaffe JIT and IBM HPCJ ahead-of-time compiler to read these attributes.

We have experimented with 10 benchmark programs, including 5 specJVM benchmarks, 3 kernels from the `scimark2` suite[1] and 2 array-based benchmarks we implemented according to standard algorithms. First, we measured the complexity of our base VCA analysis, measuring both the maximum and average sizes of the constraint graphs, and the average number of times each block was analyzed. These results show that the analysis is practical, with small graph sizes (maximum size 13) and a low number of iterations (average always less than 3).

We then measured the dynamic behaviour of array bounds checks and compared the synthetic case when all bounds checks are removed (an upper bound of what could be achieved with static analysis) and the results of our analysis. Not surprisingly, we found that it was much harder to eliminate the upper array bounds check than the lower array bounds checks. We showed that the base VCA algorithm could eliminate 3% to 60% of both the lower and upper bounds checks for array references, while adding the array field analysis and rectangular array analysis improved these results so that in five of the benchmarks we could eliminate 60% or more checks and in three of those cases we eliminate more than 99% of the checks. We also provide runtime speedups, and we showed significant speedups for both Kaffe and IBM's HPCJ.

The remainder of the paper is structured as follows. In Section 2 we give a brief introduction to the Soot framework. The base VCA algorithm is presented in Section 3 and the two additional analyses are presented in Section 4. Experimental results are given in Section 5, related work is in Section 6 and conclusions are in Section 7.

2 The Soot Framework

Soot is a Java bytecode optimization and annotation framework[16, 18]. Soot reads in a bytecode class file, converts it to an intermediate representation form called Jimple, which is typed 3-address code. Static analyses and transformations are performed on the Jimple IR. After that, the Jimple code is written back to the class file bytecode format.

2.1 Using Annotations

Soot can also be used as a bytecode annotation framework[15]. Because the bytecode is a relatively high-level instruction set, it hides some low-level operations behind the bytecode instructions. For example, a virtual machine implicitly performs the array bounds checks for array access bytecodes, such as `iaload`, `iastore`, etc. However, at the bytecode level, even if we know that an array access bytecode has an index in the safe range, it is impossible to represent such information in the bytecode itself. The attributes of a class file provide an alternative way to pass the results of a static analysis, which cannot be conveyed by the bytecode, to underlying systems. A JIT or ahead-of-time compiler can generate more efficient native code when it uses the annotation information.

Based on this idea, we developed an array bounds check elimination algorithm on the Jimple IR, and the analysis result is encoded in the attributes of a class file. An attribute-aware virtual machine or compiler can use these attributes to optimize the native code it produces.

[1] Available at `http://math.nist.gov/scimark2`.

2.2 Jimple: a typed 3-address IR

Jimple is a 3-address (stackless) intermediate representation of bytecode. It simplifies the representation of more than two hundred types of bytecode instructions to about seventeen types of Jimple statements. A Jimple statement is a typical 3-address code, which is suitable for many analyses and optimizations.

Locals in Jimple code are typed by a static type inference system[5]. The operands of a statement have declared types. Based on these types we can determine if a method involves arrays by examining the types of its locals.

Static analysis on Jimple is simplified since each Jimple statement has only one complex feature. Figure 1 shows an example. An assignment from a field reference to an array reference is achieved by using a local variable. The focus of the first statement is the field reference, and the second statement emphasizes the array reference.

```
a[i] = o.f;            $r1 = o.f;
                       a[i] = $r1;

a) Java code           b) Jimple code
```

Figure 1: Example of Jimple representation

To further improve the results of analysis, local variables are split using DU-UD webs, which is a simple alternative to SSA form. Figure 2 shows an example of the original Java code and the Jimple code.

```
a = new int[10];        r1 = new int[10];
a[i] = ... ;            r1[i1] = ...;
  ...                     ...
a = o.f;                r2 = o.f;
a[i] = ... ;            r2[i1] = ...;


a) Java code            b) Jimple code
```

Figure 2: Example of DU-UD Web

3 Variable Constraint Analysis

3.1 Overview

The objective of our variable constraint analysis is to determine the relationships between array index expressions and the bounds of the array. In Java, an array access expression of the form `a[i]` is in bounds if $0 \leq i < a.length$. If the array access expression is out of bounds an `ArrayIndexOutOfBoundsException` must be thrown, and this exception must be thrown in the correct context.

Our base analysis is intraprocedural and flow-sensitive. For each program point of interest, we use a *variable constraint graph* (VCG) to approximate the relationships between variables. The VCG is a weighted directed graph, where nodes represent variables, constants, or other symbolic representations; and edges have a weight to represent the difference constraint from the source to destination node.

The fundamental idea is that each program point of interest has a VCG to reflect the constraints among variables at that program point. These VCGs are approximated using an optimistic work-list-based flow analysis. By reducing the size of the graphs, careful design of the work-list strategy, and the appropriate use of widening operators, we have developed an efficient and scalable analysis.

In the the remainder of this section we introduce the concept of the variable constraint graph which is the essence of our algorithm. We then describe the data-flow analysis, and finally we outline the techniques we used to improve the algorithm's performance.

3.2 The variable constraint graph

Difference constraints can be represented as constraint graphs, and solved as shortest paths[4]. We have adopted this approach for our abstraction. We define a variable constraint graph as follows.

A *node* in a variable constraint graph represents one of:

1. an *int* type local which is related to some array index or array object length;
2. an *array* type local which is used represent the length of the array;
3. the constant zero; or
4. an abstract representation for fields, array elements, and common sub expressions (used only in Section 4).

An *edge* in a variable constraint graph is associated with one of:

1. $\perp$, the edge is uninitialized;
2. an integer constant; or
3. $\top$, there is no **constant** constraint from the source to the destination.

A directed edge from node j to i with a constant c represents a constraint of $i \leq j + c$. Figure 3 shows four examples. In each case a statement or expression is given, and directly below is the resulting variable constraint graph. The graphs resulting from the statements `i=j+2` and `i=2` are obvious. However, the bottom two examples require further explanation.

For a statement with the array access expression `a[i]`, two things can happen. If `i` is in bounds, then control continues to the next instruction via normal control flow, whereas if `i` is out of bounds, control is via the exception path. Assuming the normal control path, there are two constraints. First, `i` < `a.length`, which is encoded by the edge from `a` to `i` (recall that the node labelled `a` represents the length of `a`), and second $0 \leq$ `i`, which is encoded by the edge from `i` to `0`.

Conditional tests can also provide new constraints, and a test of the form `if (i < j)` generates a constraint for `i` < `j` for the true branch (encoded by the graph at label `target:`), and `i` $\geq$ `j` for the false branch (encoded by the graph at label `next:`).

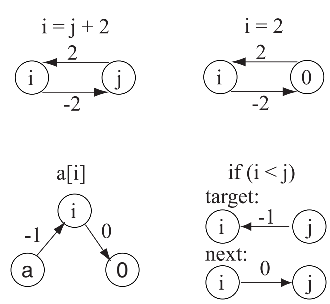


Figure 3: Graph edges representing constraints.

A variable constraint graph has the following important properties.

Directed Edges: Instead of keeping equality relationships, an assignment statement produces two directed edges between nodes. In the first example in Figure 3, the graph represents two inequalities: $i \leq j + 2$, and $j \leq i - 2$. This approach unifies the graph representation. It has another advantage of preserving constraints. For example, if the statement is followed by some statements:

$$s1 : a[i] = \cdots;$$
$$s2 : i = \cdots;$$
$$s3 : a[j] = \cdots;$$

The statement $s1$ generates the constraint $i \leq a - 1$, which makes $j \leq a - 3$. Statement $s2$ kills information about i. Thus at statement $s3$, i has lost its constraints from a and j, but $j \leq a - 3$ is still preserved.

Inequality edges are transitive: A path from a_1 to a_n can be represented by a series of constraints, for example the constraints in Figure 4 are:

$$a_2 \leq a_1 + w_1$$
$$a_3 \leq a_2 + w_2$$
$$\ldots$$
$$a_n \leq a_{n-1} + w_{n-1}$$

which implies an edge from a_1 to a_n with weight $\sum_1^{n-1} w_i$.

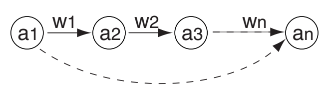


Figure 4: Transitivity of inequality edges

Shortest path gives the tightest constraint: Several paths may exist from a source to a destination node in the graph. Each path represents some constraints from different sources. However, only the shortest path gives the most accurate approximation. Any non-shortest paths are conservative estimations; they are correct, but not as precise. For example, if there are two paths from i to j with path lengths of sw and ow where $sw \leq ow$. The constraint $i \leq j + ow$ is implied by $i \leq j + sw$.

3.2.1 An example

To understand how a constraint graph can represent the constraints of locals, we look at a simple example. Figure 5(a) is a piece of code from insertion sorting and 5(b) is the corresponding Jimple 3-address code.

```
key = a[i];
j=i-1;
while (j>=0 && a[j]>key)
{
    a[j+1]=a[j];
    j--;
}
```

a) Java source code

```
    key = a[i];
    j = i-1;
label0:
    if (j<0) goto lable1
P1: $i1 = a[j];
    if ($i1 <= key) goto label1
    $i2 = j + 1;
    $i3 = a[j];
P2: a[$i2] = $i3;
    j = j-1;
    goto label0
label1:
    ...
```

b) Jimple 3-address code

Figure 5: A simple example.

Figure 6 shows the variable constraint graph (VCG) for the program from Figure 5. At program point $P1$ the constraint graph is as indicated by the solid lines. Variable j has constraints of $0 \leq j \leq a - 2$. At program point $P2$ more constraints have been generated, as indicated by the dotted lines. Note that the graph only contains variables which are relevant for index expressions or array lengths (for example, there are no nodes for `$i1` or `$i3`, since they are not relevant). Further, given the VCG at program point $P2$, we can check to see if $a[\$i2]$ is a safe reference by examining the constraints between the nodes for 0 and $\$i2$ to check the lower bound, and examining the constraints between the nodes for $\$i2$ and a to check the upper bound. The shortest path from the node $\$i2$ to the node 0 is -1, indicating $0 \leq \$i2 - 1$ or rewritten, $1 \leq \$i2$, thus the lower bound check can be eliminated. The shortest path from node a to node $\$i2$ is -1, indicating $\$i2 \leq a.length - 1$, thus the upper bound check can be eliminated.

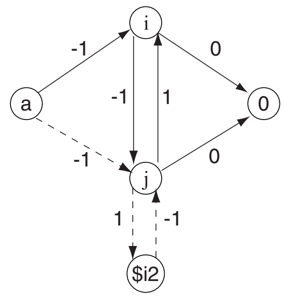


Figure 6: Constraint graphs before program points P1 and P2: solid edges are before P1, and all edges are before P2.

3.2.2 Negative Cycles

It is possible that a negative cycle exists in a variable constraint graph at some points of the data-flow analysis, before the fixed-point is reached. However, after reaching the fixed point, every reachable point in the program should have a graph without negative cycles. For example, if a negative path from a to b to c, and back to a, the edge weight is w_a, w_b, and w_c while $w_a + w_b + w_c < 0$. So we have

$$b \leq a + w_a$$
$$c \leq b + w_b$$
$$a \leq c + w_c$$

Adding both sides, we get, $0 \leq w_a + w_b + w_c$, which is a contradiction.

It is possible to have a graph with negative cycles for programs with unreachable code due to useless branches. For example: `if (i < j) { if (j < i) { P:...}}` would lead to a negative cycle at program point `P:`, but of course this point is never reached.

3.2.3 Advantages

Although there are many abstractions that can be used for array bounds check elimination, the constraint graph offers several advantages, including:

1. A constraint graph can preserve information, even when a variable is redefined.
2. The lower and upper bounds relationships can be represented in the same graph.
3. It is flexible, and can be extended to hold other information. For example, in Section 4, we show how to include information about fields and the second dimension of rectangular arrays.

3.3 Data-flow analysis

3.3.1 Overview

In our intraprocedural algorithm, we use a forward, flow-sensitive, optimistic data-flow analysis to approximate a variable constraint graph for each important point in a method body. Given the Jimple 3-address representation of a method body, we build a control flow graph (CFG) of basic blocks, where a statement with an array reference breaks a basic block into two smaller basic blocks. Thus, an array accessing statement will always appear at the top of a basic block. Each basic block is associated with an input VCG. The initial state of each graph has $\perp$ for all edges, except the entry point graph which has all $\top$ edges. The analysis is driven by a worklist algorithm which computes an ouput VCG based on the input VCG and the effect of the statements in the basic block. When processing a conditional branch statement, it may generate different constraints for the target block and the next block. After reaching a fixed point, the information for each array access statement, S, is encoded by the VCG associated basic block starting with S.

3.3.2 Abstraction

The abstraction computed by our analysis is a VCG. At any program point the set of interesting variables is known, so the set of nodes is fixed. There is one node for each variable of interest, plus a node representing the constant 0. The abstract information that changes is the weights associated to edges. For any constant c, the ordering $\perp \sqsubset c \sqsubset c+1 \sqsubset c+2 \sqsubset \ldots \sqsubset maxint \sqsubset \top$ must hold.

3.3.3 Flow Functions

The base analysis deals only with local variables, which cannot be aliased, nor can they be modified by method calls. Thus, the effect of each statement on a VCG is quite straightforward. The flow function for each kind of relevant Jimple statement is given in Table 1. Variables i, j and a represent nodes in the graph, and c is an integer constant. Each graph has a 0 node.

The rules in Table 1 use several primitives, which are defined below.

Adding an edge:

When adding a new constraint to the graph, the edge should be assigned a new weight. However, there may already exist, possibly transitive, constraints, and we want to keep the tighter constraint, which is the smaller value of new and old edge weight.

```
addedge(from, to, weight)
    oldweight = edge(from, to).weight
    if (oldweight > weight)
       edge(from, to).weight = weight
```

Updating a node's in and out edges:

For an expression $i = i + c$, we do not kill the node i. Rather, all in-edges' weights are decreased by c, and all out-edges' weights are increased by c, to reflect the constraint changes.

Detaching a node:

When a variable is assigned a new value, its old constraint edges should be removed before adding new ones. However, the edges may be part of some paths connecting other nodes, and we wish to retain this information. Thus, the `detachnode` primitive first builds edges from each predecessor to each successor, and then deletes all in and out edges.

stmts	gen	detach	operations
$i = c$	$i <= c$ $c <= i$	i	detachnode(i) addedge(0,i,c) addedge(i,0,$-c$)
$i = j + c$	$i <= j + c$ $j <= i - c$	i	detachnode(i) addedge(j,i,+c) addedge(i,j,$-c$)
$i = i + c$			update(i,c)
$a = newT[c]$	$a <= c$ $c <= a$	a	detachnode(a) addedge(0,a,c) addedge(a,0,$-c$)
$a = newT[i]$	$a <= i$ $i <= a$	a	detachnode(a) addedge(i,a,0) addedge(a,i,0)
$i = a.length$	$i <= a$	i	detachnode(i) addedge(a,i,0)
$a[i]$	$i <= a - 1$ $0 <= i$		addedge(a,i,-1) addedge(i,0,0)
$if\ (i < j)$	target: $i <= j - 1$ else: $j <= i$		 addedge(j,i,-1) addedge(i,j,0)
$i = \cdots$		i	detachnode(i)

Table 1: Statements generating constraints

```
detachnode(a)
for each predecessor p of a
  for each successor s of a
    edge(p, s).weight = edge(p,a).weight
                      + edge(a,s).weight

edge(p,a) = TOP
for each successor s of a
  edge(a,s) = TOP
```

Making the shortest path:

A constraint graph also provides methods to find the shortest path between two nodes or of all pairs. It implements single-source shortest paths and all-pairs shortest paths algorithms[4]. If the method detects a negative cycle existing in paths, it aborts the operation. This is a conservative decision. As stated before, there should not be any negative cycles at reachable program points after reaching the fixed point.

3.3.4 Merging VCGs

At confluence points we must merge VCGs coming from more than one predecessor. All predecessor graphs will have the same set of nodes, but their edges may have different weights. Thus, merging graphs is done by simply merging edge weights. Note that this is different than adding an edge to a graph. Adding edges implies the new and old constraints are existing at the same time, and the tighter one gives the most precise information. Merging edges means different constraints from multiple paths are all possible. So the merge constraint should be able to contain all possibilities, as thus we must use the maximum value. The complete merging table is given in Table 2.

It is important to note that when computing the merge of an edge $p \to q$ from two graphs $G1$ and $G2$ we need not use the value stored on the edges, rather we can get a more precise answer by using the shortest paths. Thus, we merge the shortest path from p to q in $G1$ with the shortest path from p to q in $G2$.

	$\perp$	c1	$\top$
$\perp$	-	c1	$\top$
c2	c2	MAX(c1, c2)	$\top$
$\top$	$\top$	$\top$	$\top$

Table 2: Merge two edge weights

3.4 Improving the performance of the algorithm

A naive implementation of the algorithm requires a large volume of computation to reach the fixed point. We can analyze the expensive parts of the algorithm. There are two factors dominating the performance of the algorithm: the variable constraint graph size and the time that the data-flow analysis takes to reach the fixed point. In this section, we describe some techniques we have used to reduce the performance overheads in our algorithm.

3.4.1 Limiting the size of constraint graphs

A variable constraint graph contains nodes of locals and edges between them. The size of the graph can be reduced by including only those locals that are used to compute an index or an array object length in the future. A smaller constraint graph allows faster computation of shortest paths, and may also reduce the number of iterations required for the fixed-point computation.

In order to determine the nodes which should be in the variable constraint graph, we apply a special live locals analysis, which collects only those variables relating to array references. As with ordinary liveness analysis, it is a backward flow analysis. Table 3 provides the key flow functions. The first column gives the types of statements or expressions that may generate or kill live locals. The second and third column should be used together. Only when at least one of the local(s) in the condition set are live, does the statement generate live locals in the *gen* set. Note that array references generate live locals without any conditions.

stmt/expr	cond	gen	kill
`a[i]`		{a, i}	
`a=new T[i]`	{a}	{i}	{a}
`i=a.length`	{i}	{a}	{i}
`i=j` $\pm$ `c`	{i}	{j}	{i}
`i= ...`			{i}
`if (i op j)`	{i,j}	{i, j}	

Table 3: Liveness for array references

One can easily extend the liveness analysis to accommodate other special nodes, such as class fields, array elements, and common subexpressions.

3.4.2 Reaching the fixed point early

Given the long chains in the ordering for edge weights, the ordinary fixed-point computation is too expensive. We reduce the number of iterations by applying a widening at loop entry points. At these points we replace the ordinary merge operation which uses the maximum value with a widening implemented as follows. If an edge's previous weight was not $\perp$ and the current weight increases, the edge is set to $\top$. Thus, it is clear that an edge's weight at loop headers can change two times at most.

3.4.3 Ordering the nodes of a CFG

Walking through a CFG in its topological order can speed up data-flow analysis. However, a simple depth-first search algorithm cannot guarantee an optimal order for the successors of a loop exit node. For our analysis, we prefer to visit the loop body before the loop exit. To enforce a good ordering we perform a DFS from exiting nodes of the CFG in reverse order first; then the DFS from the starting node can consult the order of reversed DFS when it meets a loop exit allowing us to put loop body nodes before loop exits.

Our work list algorithm puts the successors of a node, whose *out* set changes, onto the worklist for recalculation. The worklist is handled as a priority queue using the order computed as above. By enforcing this order we ensure that inner loops reach a fixed-point before the outer loops. Experiments show this is very effective way of making our data-flow analysis run efficiently.

3.5 Running time analysis

The performance of our algorithm is decided by two factors: the size of constraint graphs and the number of iterations required to reach a fixed-point. The nodes of a constraint graph consists of locals, therefore, the graph size is bounded by the number of locals in a method. Liveness analysis can limit the graph size even further, and our experiments confirm the graphs are small in practice.

For a control-flow graph without cycles, the data-flow analysis takes linear time to reach the fixed point. However, most of methods contain loops. At a loop entry, the special widening step of comparing an edge weight with before makes the edge weight reach a fixed-point quickly. The same path can not be visited more than twice. So the upper bound of the analysis depends on the depth of loops and the number of nodes in the loops. It can be represented as $|N| + \sum |LD| * |LN|$, where N is the total number of nodes in a CFG, LD is the loop depth, and LN is the number of nodes in the loop.

4 Extensions

The base algorithm presented in the previous section does not handle arrays stored in fields, nor can it handle multidimensional bounds checks.

In Java applications, programmers may use fields to hold some constant value for code modularity and clarity. If an analysis can detect a field holding a fixed length array object, the information allows us to extend the VCA analysis to include these fields.

Another opportunity lies in rectangular arrays. Because multidimensional arrays in Java can be ragged, it is more difficult to get good array bounds analysis for multidimensional arrays. However, in scientific programs arrays are most often rectangular. Thus, we have developed a whole-program analysis using the call graph to identify rectangular arrays that are passed to methods as parameters.

4.1 Array field analysis

A class field with modifier *final* or *private* can only be assigned a value in the class declaring that field. (The *final* type field has more restrictions.) By scanning all methods of the class, we summarize all assignments to fields with array types. If all assignments to a field are assigning arrays with the same constant length, this length can be used in the VCA analysis. Our experience shows that this usually happens for a field with an initializer, where all assignments are made in the constructors. For simplicity, our implementation of field analysis focuses only on the first dimension of array objects.

4.2 Rectangular array analysis

Java defines a very loose structure for multidimensional arrays. A multidimensional array object can have a ragged shape (different rows in an array may have different lengths); sub-arrays can be sparse in memory or aliased; and array objects can be assigned to variables of type `java.lang.Object`. All of these properties make array bounds analysis hard. However, most real applications, especially scientific programs, use rectangular multidimensional arrays.

In order to find all arrays that are rectangular, we must find all cases where a rectangular array is allocated, and we must track those allocations to their eventual uses.

If a programmer allocates a new multidimensional array using a statement of the form `new int[10][10]`, this instruction is translated into a `multianewarray` bytecode instruction which allocates rectangular arrays. However, a multidimensional array initializer is compiled by *javac* or *jikes* as individual allocations to give a potentially ragged array of array objects. An array of arrays is created, then each element is assigned a subarray object. Figure 7(a) shows a typical Java example, and Figure 7(b) shows the resulting bytecode. We use a simple pattern matcher that can find this idiom and recover a rectangular array's creation from its sparse representation to a dense one, as shown in Figure 7(c).

```
int[][] a = {{1},
             {2}};
```

a) An array initializer

```
a = newarray (int[])[2];
$r2 = newarray (int)[1];
$r2[0] = 1;
a[0] = $r2;
$r3 = newarray (int)[1];
$r3[0] = 2;
a[1] = $r3;
```

b) Compiled code by javac and jikes

```
a = multianewarray int[2][1];
$r2 = a[0];
$r2[0] = 1;
$r3 = a[1];
$r3[0] = 2;
```

c) Recovered code

Figure 7: Recover the creation of rectangular arrays

After finding all the creation sites for rectangular arrays, we then perform a simple whole program analysis to find which variables must be associated with rectangular arrays. To achieve this we build an array type propagation graph. The graph nodes consist of two special nodes for **TRUE** and **FALSE**, plus nodes representing method parameters, method locals, method returns, class fields, and array elements. To minimize the size of the graph we only include nodes for those variables whose static types indicate that they are multidimensional array objects.

A variable in the graph is connected to the **TRUE** node if it is assigned a new multi-array expression, `a=multianewarray T[i][j]`. A variable a is connected to the **FALSE** node if it appears in the statement $a[i] = c$ and a is a multidimensional array. An assignment $a = b$ adds an edge between a and b. To handle assignments due to parameter passing, we add edges between actual arguments and formals for each method call. For virtual and interface calls we use a conservative call graph to find all potential target methods. If a local is passed to or gets a return value from a method which is out of our analysis context (i.e. we do not have the method body to examine), we make a conservative assumption and connect the variable to the **FALSE** node.[2] Figure 9 gives the propagation graph of the example in Figure 8.

```
public class R
{
    public static void main(String[] args)
    { int[][] a = new int[10][9];
      int[][] b = CM(a);
    }

    static int[][] CM(int A[][])
    { int m = A.length;
      int n = A[0].length;

      int[][] B = new int[m][n];

      for (int i=0; i<B.length; i++)
      { int[] Ai = A[i];
        int[] Bi = B[i];

        for (int j=0; j<n; j++)
  p1:        Bi[j] = Ai[j];
      }
      return B;
    }
}
```

Figure 8: Rectangular array example.

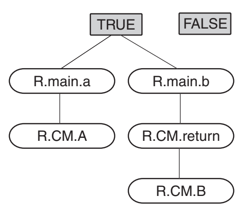


Figure 9: Propagation graph

After building the propagation graph, we want to find all nodes which are reached starting at the **TRUE** node (were allocated as rectangular), and are **not** reached starting at the **FALSE** node (may have become ragged). We achieve this as follows. First we traverse the graph, starting from the **FALSE** node, marking these nodes as reachable from **FALSE**. Then we traverse the graph starting at the **TRUE** node, finding all reachable nodes that are not marked **FALSE**. This set indicates that the members are always assigned rectangular arrays.

4.3 Extending the base VCA analysis

The field and rectangular array analysis provides additional information about class fields and multidimensional arrays, and we have extended the base VCA algorithm to utilize this information.

When the intraprocedural VCA analysis meets an array type field read of the form `a=o.f;`, and we have shown field `f` to have constant length c, we can analyse this statement as if it was `a = new T [c]` (see rule in Table 1).

[2]For our experiments, we analyzed only the benchmark code, and treated the library as out of context.

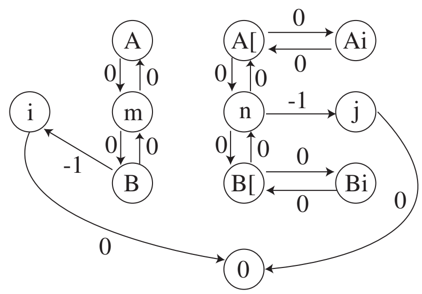


Figure 10: Constraint graph before p1

To use rectangular array information, the constraint graph has some special nodes to represent the sub-arrays. In our rectangular example (figure 8), a special node $A[$ is used to represent the second dimension length of A (figure 10 shows the constraint graph before p1). Since the VCA analysis will determine that local variable n is equal to $A[$, it is possible to determine that all array references are safe in the program.

4.4 Other enhancements

The variable constraint graph can be extended to accommodate some extra nodes, such as class fields and array references. We have done this in a very conservative way, assuming the worst-case aliasing and side-effect information. With these conservative assumptions we did not find much improvement in the result. More accurate side-effect information may improve the situation.

We did following extension to our intraprocedural algorithm. In the liveness analysis, we also add fields, array elements, and common sub expressions as locals to the live local sets. For example, $a.f$, $a[i]$, and $i * j$ can be added into the live local sets and the constraint graph can add edges connecting them to other nodes.

But it should be conservative when dealing with an assignment to a field or array element since we do not have alias information. Detailed operations are:

1. $a = \cdots$
   If a is an array type local, all array element of $a[*]$ should be killed. If it is a reference type local, all fields of $a.f$ should be killed.

2. $a.f = \cdots$
   Fields of $*.f$ should be killed.

3. $i = \cdot$
   When i is an integer variable. Array elements of $*[i]$ should be killed, and all expressions containing i, such as $i * j$, should be killed.

4. $m(a)$
   When an array or reference type local is passed to a method, all related fields and array elements should be killed since we do not know the side effect of the method call.

5 Experimental Results

5.1 Benchmarks

To demonstrate the effectiveness of our algorithms, we have implemented them in the context of the Soot framework. We chose several benchmarks including both general and numerical ones: as well as Spec and scimark2, LCS, an implementation of a Longest Common Subsequence algorithm, and MCO, an algorithm for finding an optimal order of matrix multiplication.

Before doing experiments, we measured the overhead of array bounds checks within each benchmark. In the Spec benchmarks we found 'mpegaudio' has a large overhead, as do LCS, MCO and three sub-benchmarks in scimark2. These are all typical examples of array-intensive programs using two-dimensional arrays. Other benchmarks in our study serve as examples of normal programs which are less array dedicated.

First, we measured the graph size and the number of iterated blocks, two of the most important factors affecting the algorithm's performance. In Subsection 5.3, we show the results of the base intraprocedural analysis, followed by the field analysis and rectangular array analysis as they are added in separately, and finally combined. The last section reports benchmark results for the instrumented runtime environments (Kaffe and HPCJ) making use of the data computed by the comprehensive algorithm. All numbers are dynamic measurements.

5.2 Dynamic characteristics of the algorithm

Table 5.2 shows some of the dynamic properties of our algorithm applied to the different benchmarks. The *Blocks* column gives the number of basic blocks in the program, while the *NonZero Blocks* column gives the number of blocks that have non-empty live sets for local variables, and so non-empty constraint graphs. Only NonZero blocks were used in the calculation of average and maximum constraint graph sizes, and every (non-empty) constraint graph includes at least one node for the constant zero. From this, the size of the constraint graphs is quite reasonable: average size never exceeds 10 nodes, and maximum size no more than 13. These are quite practical factors.

	Graph size (avg)	Graph size (max)	Blocks	Iter (avg)	NonZero Blocks
db	3.17	6	280	1.28	89
jack	2.5	6	2076	1.04	1892
javac	2.45	6	3347	1.27	1631
mpegaudio	3.42	10	6987	1.10	6670
raytrace	2.56	6	626	1.31	476
scimark2	5.8	12	388	1.79	301
LCS	9	13	59	2.8	55
MCO	4.6	11	98	2.0	95

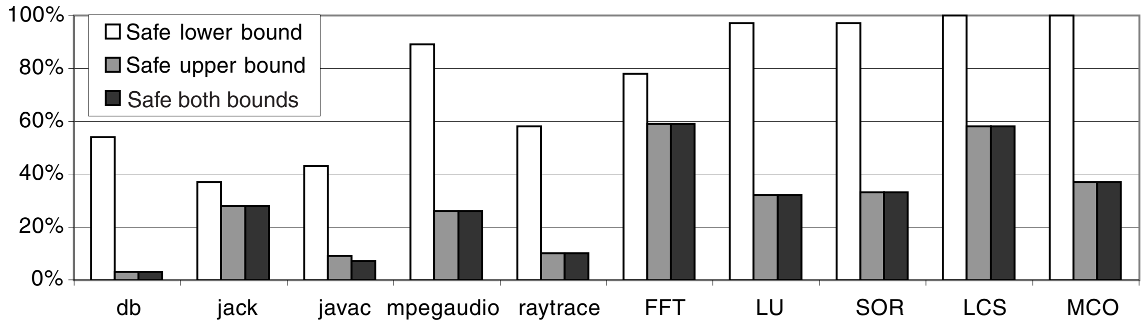
Table 4: Characteristics of the algorithm

The *Iter* column is the average number of times a block is processed as the analysis iterates toward a fixed point. It is a good indicator how long the analysis will run, and suggests that in a practical sense the running time of our algorithm is linear in the code size. There is an impact due to loop nesting; in small benchmarks, LCS, MCO and scimark2, the code bodies are dominated by nested loops and hence, the factor is higher than other benchmarks. Nevertheless, the factor remains relatively small.

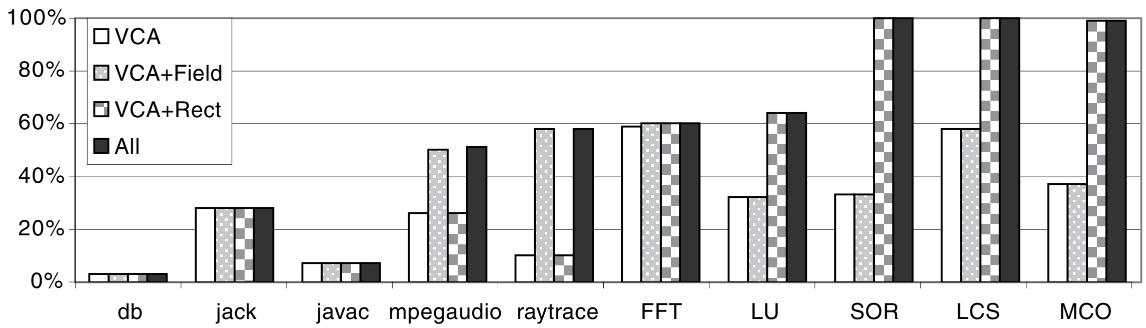
5.3 Dynamic Measurements

Figure 11(a) shows the percentage of bounds checks our basic intraprocedural analysis is able to detect are safe to remove. Note that these are dynamic statistics, obtained by instrumenting the class files and inserting profiling instructions before each array reference bytecode. Lower bounds and upper bounds are measured separately in the first two bars for each benchmark, while the last bar gives the percentage of array references with both safe checks.

The intraprocedural algorithm can determine that a fairly high percentage of the lower bound checks are safe. Safety of upper bound checks is more difficult to ascertain. Still, the results for the array-intensive benchmarks (rightmost five) are encouraging; these are the benchmarks which will benefit the most, and also in which we achieve the best results.



(a) Results of the base analysis



(b) Improvements due to field and shape analysis (both bounds safe)

Figure 11: Dynamic Results of VCA

Figure 11(b) gives the percentage of cases where both upper and lower bounds checks could be determined to be safe. The second and third bars are from the basic intraprocedural algorithm augmented with either array field analysis or rectangular array analysis; the last bar represents the intraprocedural algorithm with both array field and rectangular array analyses.

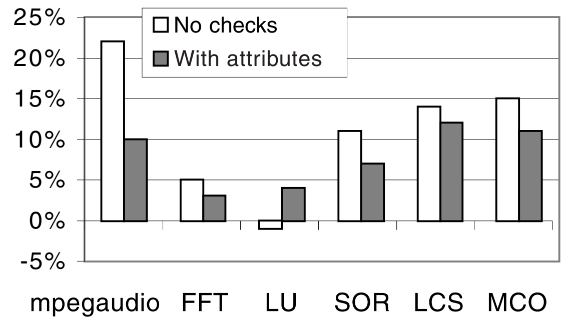
By analyzing the fields holding constant length array objects, the intraprocedural analysis can get more information about field accesses. The success of this method, however, depends on the application: 'mpegaudio' and 'raytrace' improve greatly, while others are more or less unaffected (Figure 11(b)). Rectangular analysis also proves to be very application-dependent. It is of benefit only to those benchmarks using multi-dimensional arrays. LU, SOR, and LCS and MCO improve dramatically with the addition of this analysis.

The last experiment shows the result of the combined use of field and rectangular analyses. Because these are essentially independent analyses, the combined improvement is close to the sum of the improvements seen individually. With most of our benchmarks this brings the percentage of checks we could eliminate to 50% or more; again, array-intensive benchmarks fare best, and in some cases we identify almost 100% of array bounds checks as safe.

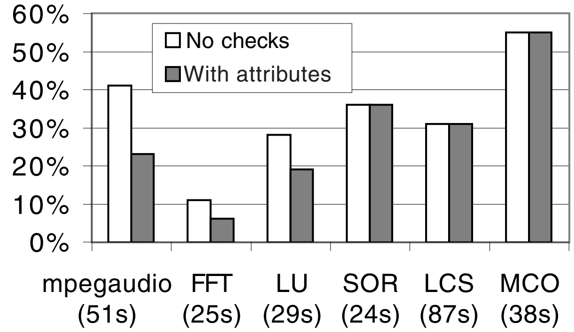
5.4 Runtime Improvements

Relative runtime performance improvements for the instrumented versions of the Kaffe JIT and HPCJ are given in Figure 12. Both systems were modified to read the array attribute information stored within the class file and to apply that data during code generation. If array bounds checks are required, a test-and-branch code sequence is inserted prior to the array access. Note that a well-known optimization for bounds checking involves making use of the 2's-complement representation of integer values to perform just a single unsigned comparison that encompasses both upper and lower bound checks (see [9]:144); this data thus represents the use of attribute information only when both bounds are declared safe.

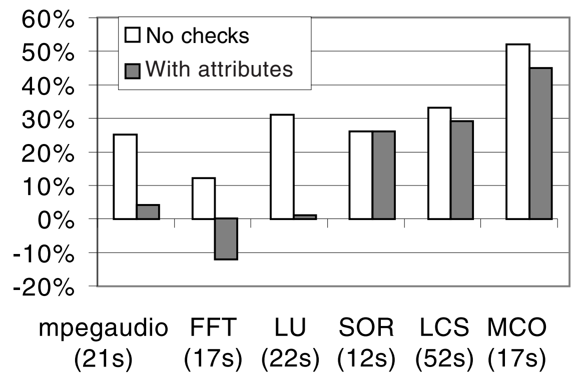
If an array access is deemed safe from the attribute information, no such checks are created—this is done



(a) Kaffe



(b) HPJC (other optimizations off)



(c) HPJC (other optimizations on)

Figure 12: Speedup for Kaffe and HPCJ

during actual (just-in-time) code generation for Kaffe, and at an internal, intermediate stage for HPCJ. In the latter case, this eliminates the potential array bounds exception that may restrict subsequent internal optimizations, resulting in different code output. For this reason we present results with and without HPCJ's own optimizations applied.

Finally, note that every array access is an object access, and so null pointer checks are also required at these points. Depending on machine architecture and how objects are organized this check can be combined with the array bounds check, and so removing the latter may require inserting explicit null pointer checks [15]. Best performance results therefore occur when both kinds of checks are eliminated; our results include this optimization. Kaffe results were gathered on a dual PII, 400MHz, 384Meg of memory, Linux OS kernel 2.2.8 and glibc-2.1.3; HPCJ results are from a PIII 500MHz, 192Meg, Windows NT.

In each case the result of using the intraprocedural analysis combined with both field and rectangular analyses is compared with the effect of artificially disabling all bounds checks. A couple of cases (LU in Kaffe, LU and FFT in HPCJ (opt)) exhibit interesting anomalous results that we have been able to attribute to code cache effects. In all other cases, however, we achieve significant performance increases, roughly corresponding to the quality of information we were able to collect.

6 Related Work

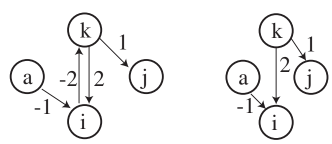
Array bounds check optimization has been performed for other languages, such as Pascal, Fortran, and Ada[14], for a long time. The problem of runtime overhead of array bounds checks was first addressed in [10]. R. Gupta[7, 8] extended their work by using data-flow analysis to move checks out of loops. These algorithms were working on languages that do not require precise exceptions, which allow an exception to be thrown before the original exception point.

The Java language requires precise exception handling, which limits the movement and modification of bounds checks. There are various ways to attack this problem. Two JIT compilers [3, 17] have implemented bounds check elimination, and both of them extended the data-flow analysis algorithm with versioning which creates a special check-free loop body. Midkiff et. al. [11, 12] proposed a Java *Array* package and loop versioning technique to overcome the bounds check overhead in Java scientific programs.

More recently, Bodik et. al. [1] presented an algorithm called ABCD for general Java applications, which uses a different form of constraint graphs to solve bounds checks. Our VCA shares some similarities with theirs, both are using inequality graphs to represent constraints. However, there are several significant differences. Their algorithm is based on an extended SSA form, and uses one graph to summarize constraints from all statements in a method. Thus, the control-flow information is included in the constraint graph. Our VCA approach uses program-point specific constraint graphs and it does not rely on any underlying program representation form. Further, VCA is more precise than ABCD. Figure 13 illustrates this point. Given the program segment in Figure 13(a), our VCA algorithm builds the constraint graph shown in Figure 13(b), whereas the ABCD algorithm builds the graph shown in Figure 13(c). Note that in the ABCD graph, constraints are only encoded along the direction of the control flow (for example, the assignment `i = k + 2;` results only in one edge, from `k` to `i`). Given this graph, it is not possible to find the safe upper bound at $p2$. However, since VCA collects a separate graph for $p2$, and the constraint gained from $p1$ is also applied to i and q, it is possible to show that the bounds are safe at program point $p2$. Another key difference is that the VCA approach can keep constraints of lower and upper bounds in the same graph, which is not the case in the ABCD approach. Finally, it is much easier to extend the VCA approach to handle multi-dimensional arrays (as demonstrated in this paper). Certainly, there is a tradeoff in speed for accuracy. The ABCD algorithm was designed to be part of a JIT compiler, and so the speed of the analysis is key. In our case we have shown reasonable speed, but it it probably better suited to an ahead-of-time analysis.

```
    i = k + 2;
p1: a[i] ... ;
    j = k + 1;
p2: a[j] ... ;
```

(a) example



(b) a part of VCA graph before p2

(c) ABCD graph for all statements

Figure 13: Comparing the VCA and ABCD constraint graphs.

The general idea of using the single-source shortest-path of an inequality graph to solve systems of difference constraints has been stated in [4](p.539-p.545).

Comparing with other algorithms, our VCA works on bytecode level and does not change the program. The analysis results are encoded in the class file attributes. Thus, there are no problems with precise exception semantics. It is capable of preserving information from various sources. Although it uses a relatively sophisticated abstraction for the data-flow analysis, the techniques used in the algorithm reduce the overhead to a minimum. VCA can be very easily extended to take advantage of results from other analyses. We demonstrated how the two extended algorithms can improve the analysis results dramatically for array intensive benchmarks.

Field analysis has been used for other optimizations such as object inlining and escape analysis[6]. Our field analysis algorithm was designed particularly for array objects. It is simple, efficient, and very effective. To target the scientific programs which use multi-dimensional arrays frequently, our rectangular array analysis provides very important information to the VCA, which helps the conservative VCA remove almost hundred per cent bounds checks in some typical applications. To the best of our knowledge, very few other works takes advantage of knowing array shapes. Further, we believe the array shape information can also help memory layout of array objects in a virtual machine[2].

7 Conclusions

In this paper we have presented a collection of techniques for eliminating array bounds checks in Java. Our base analysis, variable constraint analysis (VCA), is a flow-sensitive intraprocedural analysis that approximates the constraints between important program variables at program points corresponding to array access statements. The analysis has been made efficient by reducing the size of the graphs, choosing an appropriate worklist order, and applying a widening at loop entry points. As shown in the experimental results, the size of the graphs is small (around 10 nodes for our benchmarks), and the average number of iterations per basic block is always less than 3.

In order to improve the precision of the base VCA analysis, we have described two additional techniques. Array field analysis is applied to each class to find those array type fields that always hold an array with a fixed constant length. Rectangular array analysis is applied to a whole program to find those variables that always refer to rectangular, non-ragged, arrays. Given the information from these analyses, the intraprocedural VCA analysis was improved to include information about fields, and upper dimensions for multi-dimensional arrays.

Our analyses were implemented in the Soot optimization/annotation framework, and we provided dynamic results that showed that effectiveness of the base VCA analysis and the incremental improvements due to field and rectangular array analysis. These results were quite encouraging and demonstrated that almost all checks could be eliminated for those benchmarks with very regular computations. We also provided experimental results for Kaffe and IBM's HPCJ to demonstrate that significant runtime savings can be achieved as a result of the analysis.

Our next phase of work will be to integrate a side-effect analysis into the framework, and improve upon information for arrays stored in objects. We would also welcome the opportunity to provide our attributed class files to other groups in order to see the runtime impact on other virtual machines.

References

[1] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation(PLDI)*, pages 321–333, Vancouver, BC, Canada, June 2000.

[2] M. Cierniak and W. Li. Optimizing Java bytecodes. *Concurrency, Practice and Experience*, 9(6):427–444, 1997.

[3] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation(PLDI)*, pages 13–26, Vancouver, BC, Canada, June 2000.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill and MIT Press, 1990.

[5] E. M. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium 2000*, Lecture Notes in Computer Science, pages 199–219, Santa Barbara, June 2000.

[6] S. Ghemawat, K. Randall, and D. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation(PLDI)*, pages 334–344, Vancouver, BC, Canada, June 2000.

[7] R. Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, White Plains, NY, June 1990.

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

[9] S. Hoxey, F. Karim, B. Hay, and H. Warren, editors. *The PowerPC Compiler Writer's Guide*. IBM Microelectronics Division, 1986.

[10] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, pages 114–119, June 1982.

[11] S. Midkiff, J. Moreira, and M. Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.

[12] J. Moreira, S. Midkiff, and M. Gupta. A Standard Java Array Package for Technical Computing. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.

[13] J. Moreira, S. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, Mar. 2000.

[14] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[15] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. Sable Technical Report 2000-2 (revised), Sable Research Group, McGill University, Sept. 2000.

[16] Soot - a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[17] 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.

[18] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In D. A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Germany, March 2000. Springer.