A Comprehensive Approach to Array Bounds Check Elimination for Java

Feng Qian, Laurie Hendren, Clark Verbrugge*
[fqian,hendren,clump]@cs.mcgill.ca

School of Computer Science, McGill University

Abstract. This paper reports on a comprehensive approach to eliminating array bounds checks in Java. Our approach is based upon three analyses. The first analysis is 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 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 such as code motion and loop

* Verbrugge's work done while at IBM Toronto Lab

transformations because the Java virtual machine specification requires precise exception handling.

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 bounds check elimination algorithm which consists of three analyses: *variable constraint analysis* (VCA for short), *array field analysis*, and *rectangular array analysis*. The combination of these analyses can prove that many array references are safe, without transforming the original program.

Variable constraint analysis 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 analysis was designed to take advantage of the fact that variables used in index expressions often have very short lifetimes—by only building graphs for live variables of interest the graphs are kept quite small. The associated worklist algorithm is also tuned in order to reduce the number of iterations. As a result, the actual running time is linear in the size of the method being analyzed.

Array field analysis is used to track the storage of array objects into class fields. By analyzing assignments to fields that have certain modifier restrictions (e.g., *private* and *final*) we are able to efficiently capture information about arrays that may not be locally-allocated, but which still have limited scope.

Finally, *rectangular array analysis* approximates the shape of multidimensional arrays. This analysis looks at the call graph for the whole application and identifies multidimensional array variables with consistent rectangular shapes.

Both *array field analysis* and *rectangular array analysis* provide information consulted by the VCA and therefore improve the analysis results.

All three analyses have been implemented using the Soot bytecode optimization framework[9], but could be easily implemented in other compilers with good intermediate representations. 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 provide dynamic results showing the number of array bounds checks eliminated, and the effect of the additional field and rectangular array analysis. We also provide runtime measurements demonstrating significant speedups for both Kaffe and HPCJ.

The remainder of the paper is structured as follows. The base VCA algorithm is presented in Section 2 and the two additional analyses are presented in Section 3 and 4. Experimental results are given in Section 5, related work is in Section 6 and conclusions are in Section 7.

2 Variable Constraint Analysis

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 $0 \leq i \leq a.length - 1$. 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* between the source and destination node. The interesting program points are the entry points of basic blocks. An array reference breaks a code sequence into two blocks, with the actual array reference starting the second block.

The fundamental idea is that the entry of each basic block 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 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 the techniques we used to improve the algorithm's performance.

2.1 The variable constraint graph

Systems of difference constraints can be represented by constraint graphs, and solved using shortest-path techniques[3]. We have adopted this approach for our abstraction.

A *node* in a variable constraint graph represents the constant zero, or a local with the *int* or *array* type. A graph *edge* has a value of $\perp$, an integer constant, or $\top$. For any constant c, the ordering $\perp < c < \top$ holds. A directed edge from node j to i with a constant c represents a difference constraint of $i \leq j + c$.

The data-flow analysis uses constraint graphs to encode flow information. It needs to change constraints between a set of variables after various statements. The information changes are reflected by operating on the variable constraint graph. In following text, we define operations (or primitives) applicable to our constraint graph.

Creating a graph: As we will see later, the set of vertices of a graph at an interesting program point can be pre-computed and never change again. There is no constraint between variables at the initial state. Thus, the initializing function accepts a set of vertices, while all edges are set to $\perp$ which means edges are uninitialized.

Adding an constraint: A new constraint is added in a graph by changing the weight of the corresponding edge. In order to keep the tightest constraints possible, the edge is assigned the minimum of its old and new weight. This operation is named *addedge* in Table 2.

Deleting a constraint: When a constraint does not hold anymore, the corresponding edge weight is set to $\top$ in the graph. Right now, a constraint is deleted when detaching a node.

Detaching a node: When a variable is assigned a new value, its old 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 removes all in and out edges.

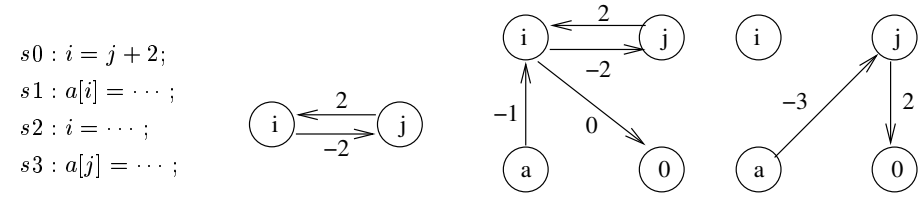
Updating a node's in and out edges: For an expression $i = i + c$, we do not kill the node i. Rather, all in-edges' weights are increased by c, and all out-edges' weights are decreased by c, to reflect the constraint changes. We call this operation *update* in Table 2.

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[3].

Merging two graphs: At confluence points we must merge constraint graphs 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 unlike adding a constraint, the merged edge weight is the maximum of the corresponding incoming edge weights.

Negative cycles: Negative cycles may exist in a constraint graph 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. In the presence of negative cycles in a path, we cannot compute the shortest path weight for nodes in the path. Leaving them unchanged is a conservative approach to keep the correctness of the analysis.

Figure 1 shows an example of constraint graphs. We are interested in the graph before $s3$ because it has an array access and we want to know whether j is in the bounds. The other two graphs only reflect the constraint changes.



(a) a basic block (b) vcg before s1 (c) vcg before s2 (d) vcg before s3

Fig. 1. The status of constraint graph changes

The statement $s1$ generates the constraints $\mathtt{i} - \mathtt{a} \leq -1$ and $\mathtt{0} - \mathtt{i} \leq \mathtt{0}$, resulting in edges from a to i and i to 0. The path from a to j implies the constraint $\mathtt{j} - \mathtt{a} \leq -3$ by adding its edge weights. In statement $s2$ i loses its contraints from a and j and the path $a \rightarrow i \rightarrow j$ ceases to exist; the constraint condition is preserved though by a new edge directly from a to j with weight -3. Thus the constraint $\mathtt{j} - \mathtt{a} \leq -3$ is still in effect before $s3$, even when i was redefined. The upper bound check for $s3$ can therefore be proved safe (we can not derive the safe lower bound from this simple example, because it only implies $0 - j \leq 2$).

2.2 Data-flow analyses

We developed two data-flow analyses in our intraprocedural algorithm. A special live-local analysis, which is relatively simple, determines the set of local variables which are related to array references. A more complicated analysis performs abstract execution of the method, and gets a conservative approximation of constraints among live locals. The first analysis limits the number of nodes in a constraint graph and therefore reduces the computation of the second analysis.

Array-related liveness analysis 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 1 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.

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

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

Table 1. Liveness for array references

Variable constraint analysis We use a forward, flow-sensitive, optimistic data-flow analysis to approximate a variable constraint graph for each important point in a method body.

The analysis is based on the control-flow graph of basic blocks as we explained before. The entry of each basic block is associated with an input VCG whose vertices are array-related live locals. 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 work-list algorithm which computes an output 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.

At any program point the set of interesting variables is known from array-related liveness analysis. The abstraction computed by our analysis is all-pairs shortest paths of a variable constraint graph. But instead of computing the shortest paths at every program point, we only perform such computation at the confluence point. In other places, we do simple operations on the graph. The abstract information that changes is the weights associated with edges. For any constant c, the ordering $\perp \sqsubset c \sqsubset c+1 \sqsubset c+2 \sqsubset \ldots \sqsubset maxint \sqsubset \top$ must hold.

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 statement is given in Table 2. Variables i, j and a represent nodes in the graph, and c is an integer constant. Each graph has a 0 node.

The first column shows the kinds of statement which have effect on a VCA. The second column lists the constraints can be generated from the statement in the first column. The third column shows the node of which constraints should be bypassed. The last column gives operations on the constraint graph according to the statement. The rules in Table 2 use several primitives, which were defined in section 2.1.

At a confluence point P, we use a set of output graphs from predecessors and the old input graph of P to compute the new input graph. We firstly call the *merge* operation to union all ouput graphs from predecessors, then apply a

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

Table 2. Statements generating constraints

special operation called widening on each new graph edge weight by comparing it to the old graph edge weight. The widening operation looks at the changing trend of an edge weight. If the weight is increasing, we set it to $\top$ directly. But if the new weight is less than the old weight, we will discard the new weight and use the old one. The widening technique speeds up the symbolic execution and also stops infinite loops correctly.

Walking through a CFG in its topological order can speed up data-flow analysis. However, a simple depth-first search (DFS) 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 work-list for recalculation. The work-list is handled as a heap 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 Array Field Analysis

The base analysis presented in the previous section does not handle arrays stored in fields. In Java applications, programmers may use fields to hold some constant value for code modularity and clarity. A class field with the `private` or `final` modifier can only be assigned a value in the class declaring that field. Based on this observation, we developed a simple analysis detecting a field holding a fixed length array object.

For each class C, *array field analysis* examines the class fields. Let F_C be the set of array-type fields modified by `private` or `final` declared in C. If F_C is non-empty, then a table τ_C is created, and for each $f \in F_C$ an entry $\tau_C[f]$ is created and initialized to $\perp$. Each method m declared in C is then considered. Since the Soot framework provides typed locals, and ensures that a `putfield` or `putstatic` is always in the form of an assignment from a local to a field, a simple pre-scan of the types of locals of m can be used to avoid further processing of methods that cannot change the value of any $f \in F_C$.

For each method m that might change an array field, the body of m is scanned. Let $f = \ell$ be an assignment to some $f \in F_C$. A value $\delta(\ell)$ is computed as follows:

1. If ℓ is a `newarray` or `multianewarray` operation, then extract the array length expression d and return $\delta(d)$.
2. If ℓ is a local variable, the UD-DU chains provided by the Soot framework are used to locate the definitions of ℓ. If ℓ has more than one definition point, return $\top$, otherwise for a definition $\ell = x$ return $\delta(x)$.
3. If ℓ is an integer constant c, return c.
4. Otherwise, return $\top$.

The table information $\tau_C[f]$ is then updated by merging the existing value for $\tau_C[f]$ with the computed $\delta(\ell)$ according to Table 3; note that $\delta(\ell)$ is never $\perp$.

	$\perp$	c1	$\top$
c2	c2	c1 : c1==c2 $\top$: else	$\top$
$\top$	$\top$	$\top$	$\top$

Table 3. The rule for updating the field table.

When the intraprocedural VCA analysis meets an array type field read of the form $a = o.f$ where o has class type C, it consults the array field analyzer to get

the value $\tau_C[f]$. If $\tau_C[f]$ has a constant value c, we can analyse this statement as if it was `a = new T[c]` (see rule in Table 2).

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 array field analysis focuses only on the first dimension of array objects.

4 Rectangular Array Analysis

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.

A multidimensional array can be allocated by explicit `new` instruction, or an array initializer. The 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 2(a) shows a typical Java example, and Figure 2(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 2(c).

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

a) An array             b) Compiled code by          c) Recovered code
   initializer             javac and jikes
```

Fig. 2. 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, locals, 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.[1]

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.

To use rectangular array information, the constraint graph has some special nodes to represent the subarrays. For example, we use `A[` to represent the second dimension length of `A`.

5 Experimental Results

We have implemented the algorithm in the context of the Soot framework. In this section we present and discuss the experimental results that we have obtained. The results are grouped into three categories:

1. We measured the dynamic characteristics of the *variable constraint analysis* in terms of two most important factors affecting the algorithm's performance: the *size* of variable constraint graphs and the *number of iterated blocks* to reach the fixed point.
2. Then we show the results of the base intraprocedural analysis, followed by the *array field analysis* and *rectangular array analysis* as they are added in separately, and finally combined. The results are presented as percentages of lower and upper bound checks that can be proved safe.
3. Our analyses results are encoded in the attributes of class files. To measure the real impact to the run-time performance of Java programs, we modified Kaffe JIT and HPCJ compiler to read and take advantages of such attributes. The run-time measurements show speed-ups in most of benchmarks.

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

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

large overhead, as do LCS, MCO and three sub-benchmarks in scimark2. These are all typical examples of array-intensive programs. Other benchmarks in our study serve as examples of normal programs which are less array dedicated.

5.1 Dynamic characteristics of the algorithm

Table 4 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 have 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: the average size never exceeds 10 nodes, and the maximum size is no more than 13. These are quite practical factors.

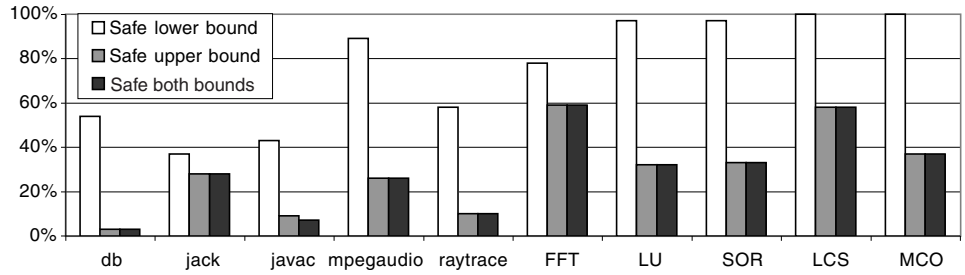
	Graph size		Blocks	Iter	NonZero
	(avg)	(max)		(avg)	Blocks
db	3.17	6	280	1.28	89
jack	2.5	6	2076	1.04	1892
javac	2.45	6	3347	1.27	1631
mpegaudio	3.42	10	6987	1.10	6670
raytrace	2.56	6	626	1.31	476
scimark2	5.8	12	388	1.79	301
LCS	9	13	59	2.8	55
MCO	4.6	11	98	2.0	95

Table 4. Characteristics of the algorithm

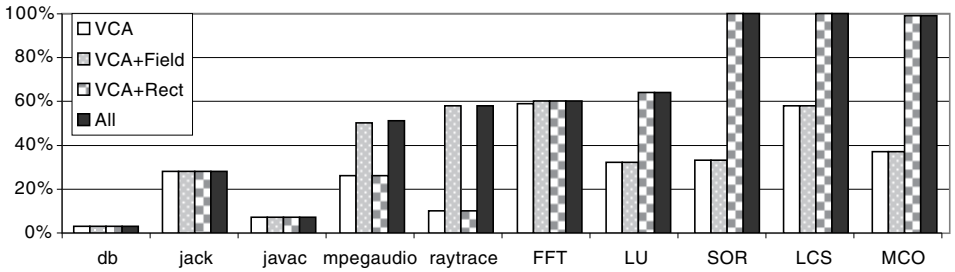
The *Iter* column is the average number of times a block is processed as the analysis iterates toward a fixed point. It is a good indicator of how long the analysis will run, and suggests that in a practical sense the running time of our algorithm is linear in the code size. There is an impact due to 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.

5.2 Dynamic results and discussion

Figure 3(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.



(a) Results of the base analysis



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

Fig. 3. Dynamic Results of VCA

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.

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 3(b)). Rectangular analysis also proves to be very application-dependent. It is of benefit only to those benchmarks using multidimensional 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.

Relative runtime performance improvements for the instrumented versions of the Kaffe JIT and HPCJ are given in Figure 4. 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 [6]: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 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.

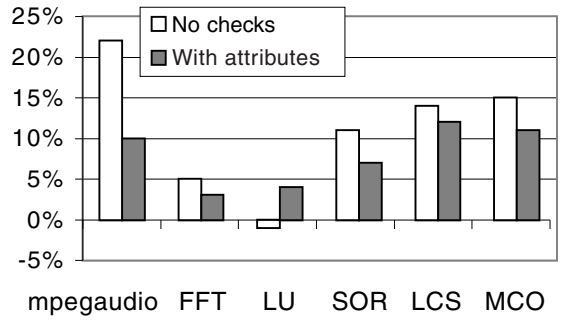
Finally, note that every array access is an object access, and so null pointer checks are also required at these points. Depending on machine architecture and how objects are organized this check can be combined with the array bounds check, and so removing the latter may require inserting explicit null pointer checks [9]. Best performance results therefore occur when both kinds of checks are eliminated; our results include this optimization. Kaffe results were gathered on a dual Pentium II, 400MHz, 384Meg of memory, Linux OS kernel 2.2.8 and glibc-2.1.3; HPCJ results are from a Pentium III 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.

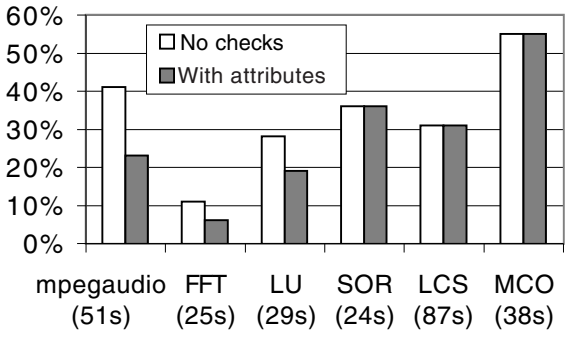
6 Related Work

Array bounds check optimization has been performed for other languages, such as Pascal, Fortran, and Ada[8] for a long time. The problem of runtime overhead of array bounds checks was first addressed in [7]. R. Gupta[4, 5] 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.

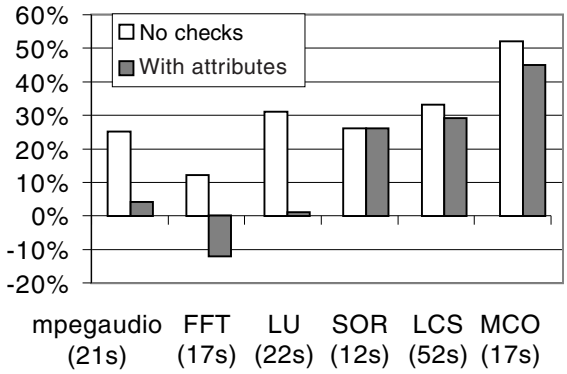
More recently, Bodik et. al.[1] presented an algorithm called ABCD (Eliminating Array Bounds Checks on Demand) for general Java applications, The algorithm uses a different form of constraint graphs to solve bounds checks. It builds an extended SSA form for a method body. The e-SSA guarantees that all uses (by name) of a variable are bounded by the same constraints, the value range, at runtime. Based on the new form, a constraint graph is constructed, where nodes are locals and constants, and weighted edges are constraints rep-



(a) Kaffe



(b) HPJC (other optimizations off)



(c) HPJC (other optimizations on)

Fig. 4. Speedup for Kaffe and HPCJ

resenting inequality relationship between nodes. The relationship between array and index is inferred by a customized depth first search.

VCA has some similarity to this approach in that both are using inequality graphs to represent constraints. However, there are several differences between our algorithm and ABCD approach:

1. The ABCD 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 does not rely on any underlying program representation form, it uses a fixed number of small program-point specific constraint graphs.
2. Based on e-SSA form, the ABCD algorithm can be used in a demand-driven manner. Each demand (query) is solved individually, and may be performed on selected array references that occur in hot spots. Although each query is relatively expensive, ABCD does have an overall speed advantage over VCA. The VCA approach is designed to analyze all array references at once, and is intended for off-line usage. Our experimental results show that our techniques for reducing the size of the graphs and reducing the number of iterations works well to keep the cost of VCA reasonable.
3. The VCA approach keeps constraints of lower and upper bounds in the same graph, which is not the case in the ABCD approach.
4. In our algorithm, the constraint graph serves as the basis of other two analyses. For certain types of applications, the impacts of these analyses can be significant. Currently it is not clear how class fields and multidimensional arrays information can be used to help the ABCD algorithm.
5. ABCD is capable of catching partial redundant bounds checks. VCA is not able to do that currently.

VCA's primary advantage is in its interaction and integration with other analyses. In isolation, VCA is capable of recognizing nearly the same percentages of safe upper bounds on the SPEC JVM98 benchmarks as reported in[1]. However, when combined with *array field analysis* and *rectangular array analysis*, VCA can outperform ABCD significantly. Experiments show that VCA with *rectangular array analysis* is very effective on micro benchmarks using two-dimensional arrays. In addition, we have provided complete experimental results showing runtime speedups. We also think the approach of formulating a problem in constraint graphs and solving it by using data-flow analysis can be useful for other problems.

R. Shaham et. al. [11, 10] described an algorithm for identifying live regions of arrays to detect array memory leaks in Java. Although in a very different experimental setting, their representation and analysis are very similar to VCA. In both cases constraint graphs and data-flow analyses are used to compute inequalities between variables. However, their focus is on finding relationships between special class fields across method boundaries based on supergraphs of a few particular library classes. Although the supergraph can make our *field analysis* more powerful, our VCA approach focuses on intraprocedural analysis for general Java applications, and we handle different statements in more detail.

Another important aspect of our VCA approach is that we use different techniques to reduce the cost of data-flow analysis, such as limiting constraint graph node size, and enforcing iteration in pseudo-topological order.

Compared 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.

To target the scientific programs which use multidimensional arrays frequently, our rectangular array analysis provides very important information to the VCA, which helps the conservative VCA remove almost one 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. Currently our attributes are not verifiable, we are just using them as an experimental tool to convey dataflow facts from our tool to the ahead-of-time or JIT compiler.

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. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill and MIT Press, 1990.
4. 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.
5. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems, 2(1-4):135–150*, 1993.
6. S. Hoxey, F. Karim, B. Hay, and H. Warren, editors. *The PowerPC Compiler Writer's Guide*. IBM Microelectronics Division, 1986.
7. 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.
8. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
9. P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing java using attributes. In *Proceedings of Compiler Construction, 2001*, pages 334–554, 2001.
10. R. Shaham. Automatic removal of array memory leaks in Java. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, September 1999. Available at http://www.math.tau.ac.il/~ rans/thesis.zip.
11. R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in java. In D. A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 50–66, Berlin, Germany, March 2000. Springer.