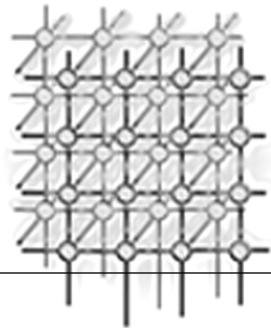

Safe Bounds Check Annotations

Jeffery von Ronne*, Andreas Gampe,
David Niedzielski, and Kleantes Psarris

*Department of Computer Science,
The University of Texas at San Antonio*



SUMMARY

The Java programming language's semantics require that out-of-bounds array accesses be caught at run-time. In general, this requires dynamic checks at the time the array element is accessed. Some of these checks can be eliminated statically during just-in-time (JIT) compilation, but the most precise analyses are too expensive to run in JIT compilers. This paper presents a framework in which thorough static range analyses can be used safely during the less-performance-critical compilation of Java source into machine-independent mobile code. In this framework, the static analysis results are used to derive proofs that certain linear inequality constraints hold. These linear constraints and their proofs are then added to the mobile code as annotations. The annotation framework is designed so that proofs can be verified efficiently. This allows the JIT compiler to safely eliminate array bounds checks during compilation without expensive runtime analysis. Experiments with a prototype system that can generate and verify these annotations, demonstrate that this framework is more precise than prior work and that verification is efficient.

1. Introduction

The semantics of the Java programming language require that out-of-bounds array accesses be caught at run-time [8]. This can be achieved by performing a runtime "bounds check" as part of each access to an array, but the overhead of doing so can be quite substantial [10]. This performance overhead is caused not only by the direct cost of conditional branches implementing the array bounds checks but also by lost opportunities for optimization and parallelization due to Java's precise exception semantics. One way to reduce this overhead is to apply a static analysis that identifies some of the array element accesses that will never cause an out of bounds exception and to optimize the program by eliminating those unnecessary bounds checks.

Because Java is designed to use a virtual machine providing platform-independence, dynamic class-loading, and safety, most optimizations are done at runtime during just-in-time (JIT) compilation. Although it is possible for JIT compilers to perform bounds check elimination, the most precise

*Email: vonronne@cs.utsa.edu



analyses (e.g., [10, 13, 15]) are too expensive to run in JIT compilers. For that reason, faster analyses (e.g., [2],[19]), which leave more unnecessary array bounds checks in the program, are used instead.

In this paper, we present verifiable annotations for SSA-based representations of Java programs, such as SafeTSA [1], that can safely represent the results of range analyses as linear inequalities. These annotations can be produced during the translation of Java source into SafeTSA. Then during program execution, an enhanced Java Virtual Machine can read these annotations, efficiently verify that these linear inequalities hold, and then perform array bounds check elimination based on those linear inequalities. In this way, the range analysis burden can be shifted from the performance-critical JIT compiler, which runs every time the program is executed, to the source-to-SafeTSA compiler, which only needs to be run once.

1.1. Linear Constraints

In order to eliminate the bounds checks associated with an array access, it is necessary to determine that the index is not smaller than the smallest legal index and that the index is not larger than the largest legal index. In Java, these constraints mean that for an array access, $a[index]$ (where $index$ is some integer expression), to be legal, then:

$$0 \leq index < a.length$$

Thus, what is required is to determine what the range of values that $index$ can take.

There have been several algorithms for eliminating bounds checks for Java. They can broadly be categorized as either recognizing specific loop forms (e.g., [10, 19]) and determining the range of the loop induction variable (when it is used as an $index$ into an array), or as performing global analysis to limit the range of program variables relative to each other (e.g., [2, 14]). The information used by these algorithms (except, perhaps, that of Moreira et al. which also does substantial loop transformation [10], and that of Würthinger et al. which relies on dynamic speculation [17]) can be expressed as a limited form of linear inequalities, known as difference constraints.

Generally, if integer program variables and the lengths of the arrays pointed to by program variables are treated as algebraic variables, then it is possible to examine the semantics of various instructions and the control flow of the program to derive linear inequalities that constrain the possible values the program variables may take at various points in the program. Those constraints can then be combined into a system of inequalities which are entailed by the program's semantics. An array access instruction (using a as the array and i as the index) does not need runtime bounds checks, if the appropriate constraints can only be satisfied when:

$$i \geq 0 \quad \text{and} \quad i < a.length$$

(where $a.length$ represents the length of the array a).

For example, the linear constraints on the right can be extracted from the program fragment on the left:

```
int l = a.length;           ①  $l \leq a.length$ , ②  $l \geq a.length$ 
if (x < l - 1) {           ③  $x < l - 1$ 
    y = x + 1;             ④  $y \leq x + 1$ , ⑤  $y \geq x + 1$ 
    a[y];
}
```



Note that, since the variables are all integers, we can change $x < y$ into $x + 1 \leq y$. In addition, the system of linear inequalities is easier to solve if we rearrange them so that they are all asserting that some linear combination of variables is less than or equal to zero:

$$\textcircled{a} \ l - a.\text{length} \leq 0, \quad \textcircled{b} \ -l + a.\text{length} \leq 0, \\ \textcircled{c} \ x - l + 2 \leq 0, \quad \textcircled{d} \ -x + y - 1 \leq 0, \quad \text{and} \quad \textcircled{e} \ x - y + 1 \leq 0$$

If we then add constraints \textcircled{a} , \textcircled{c} , and \textcircled{d} , we can see that $y - a.\text{length} + 1 \leq 0$. This is equivalent to our condition for the index y being below the upper bound of the array ($y < A.\text{length}$), so the upper-bound check is unnecessary and can be eliminated. It is not possible, however, to show from this system that y is non-negative ($-y \leq 0$), so the lower-bound check associated with this array access is necessary and cannot be eliminated without more information.

If the program has been translated into three-address code, then the constraints derived from these instructions that can be expressed as linear inequalities will involve at most three variables. Instructions that add or subtract constants can be expressed as difference constraints (i.e., linear inequalities having coefficients of only 1 and -1 and involving only two variables and a constant).

One additional issue is the scope of the constraints, which may only be valid in certain parts of the program; normally this will be the region of the program in which the variables involved in the constraint are unmodified. Tracking this can be simplified if programs are translated into static single assignment form (SSA) (where each local variable is assigned to at only one instruction and ϕ -functions are used as necessary to merge alternative values into new variables based on control flow) [6], since variable scope and the validity of constraints thus coincide. Therefore, our annotation framework uses programs in SSA form.

1.2. Integer Underflow/Overflow

Like most programming languages, Java has a primitive integer type (`int`) that cannot represent any number in the integer domain, but is instead, restricted to integers that can be stored into a 32-bit word using 2's complement representation. When integer arithmetic operations (specifically, addition, subtraction, increment, decrement, and multiplication) result in a value less than -2^{31} (-2147483648 , which we will refer to symbolically as MIN) or more than $2^{31} - 1$ (2147483647 , which we will refer to symbolically as MAX), the computation "wraps around" so that only the least significant 32-bits are retained, so for example, $1000000 * 1000000$ is -727379968 rather than 1000000000000 .

This difference between the mathematical domain of integers (\mathbb{Z}) and Java's primitive `int` type needs to be taken to account during integer range analysis. If not, the soundness of the array bounds check elimination may be compromised. Combining linear inequality constraints relies on properties, such as $(x \leq 0 \wedge y \leq 0) \Rightarrow x + y \leq 0$, which hold for standard arithmetic in \mathbb{Z} , but not for 2's complement arithmetic. Therefore, linear inequality constraints must be expressed in the domain of integer numbers. But linear inequality constraints that would seem to fall out naturally from source code statements may not hold when constraints among 2's complement variables are expressed as inequalities of variables in \mathbb{Z} . For example, consider the claim that the Java statement, $z = x + y$, would seem to imply that $z = x + y$, implies that $-[[z]] + [[x]] + [[y]] \leq 0$ (where $[[x]]$ denotes the value in \mathbb{Z} of the Java variable x). This claim may not hold if $x + y$ overflows or underflows. For example, if $x \equiv 100$, $y \equiv 2147483647$ (i.e., $2^{31} - 1$), then $z = -2147483549$ (i.e., $-2^{31} + 99$), and it is clear that $-(-2147483549) + 2147483647 + 100 \not\leq 0$.



In some cases, this can compromise the soundness of the analysis results and the Java Virtual Machines memory safety. For example, consider the following program fragment:

```
int i = 2147483647;
if (i >= 0) {
    int j = i + 100;
    if (j < a.length)
        x = a[j]
}
```

Due to an integer overflow, the actual value of j being used as an index is -2147483549 , which is clearly not within $[0, a.length)$, so the bounds check associated with the access of $a[j]$ cannot be legally removed. This has not always been correctly addressed in prior work (e.g., ABCD [2]). In the example above, the ABCD algorithm [2] would determine* that $j \geq 0$ and the lower bound is unnecessary by extracting the constraint $i - 0 \geq 0$ from the condition $i \geq 0$, and the constraint $j - i \geq 100$ from the condition $j = i + 100$ and combining them to obtain: $j - 0 \geq 100$ which can be weakened to: $j - 0 \geq 0$. Similarly, the upper bounds check can also be removed based on the condition: $j < a.length$. Thus, the ABCD algorithm would remove the bounds check, which would allow the out-of-bounds memory access to proceed. This is a potentially exploitable flaw that makes virtual machines utilizing ABCD (or similar bounds check elimination algorithms that do not properly handle integer overflow) vulnerable to buffer overflow attacks.

2. Annotating Claims and Proofs

Since Java programs are usually optimized during JIT compilation of Java bytecode, for an optimization to be profitable, it has to “pay for itself.” That is, the cost of performing an optimization must be less than the speedup in a program’s execution resulting from the optimization. In this environment, simple optimizations that cover the most common cases are often preferred to more comprehensive optimizations that produce better code but take much longer to execute. This is the approach taken by, the ABCD algorithm [2], for example.

We take a different approach, instead of trying to use a light-weight bounds check elimination algorithm that covers the most common cases, we are developing an annotation scheme that allows more expensive static analysis to be performed by the annotator once prior to distribution and prior to JIT compilation by the code consumer. The results of the analysis can be shipped to the code consumer as annotations to the bytecode, and the code consumer’s JIT can then optimize the bytecode based on these annotations. This is only safe if the annotations can be verified to be correct, otherwise the annotator could lie causing the code consumer to omit necessary bounds checks leaving the host virtual machine susceptible to buffer overflow attacks.

*In practice, constant propagation would probably prevent ABCD from eliminating the bounds check, but if i was assigned 2147483647 based on user input, the result would be as described here.



Table I. Axiomatic Linear Inequalities

Inequalities		
①	0	≤ 0
②	-1	≤ 0
③	$[[a.\text{length}]] - \text{MAX}$	≤ 0
④	$-[[a.\text{length}]]$	≤ 0
⑤	$[[x]] - \text{MAX}$	≤ 0
⑥	$-[[x]] + \text{MIN}$	≤ 0

x is a Java variable of type `int`
 a is a Java variable referencing an array

2.1. Annotated Claims

The core of our annotations are linear inequality constraints. We believe that this will allow a range of optimization algorithms to be employed by various code annotators without imposing a substantial costs in the code consumer. In addition, we add some constraints that allow information derived from the control flow into the system. The following kinds of constraints are allowed in our system:

linear inequality constraints assertions of linear inequalities involving program variables, constants, and the lengths of arrays

predicate constraints assertions that a particular boolean variable is either true or false

predicated constraints assertion that a constraint will hold if a particular predicate holds

Except for a handful of universally valid axiomatic constraints (see Table I), a constraint cannot be considered valid unless an annotation claims it to be true. These claims are anchored to a particular program point. The claims must match one of the claim rules listed in Table II, III, IV. These may be categorized as:

- the claim that a bounds check is unnecessary (Rule (0) in Table II)
- a linear inequality constraint anchored to a particular integer or array operation (Rules (1)-(14) in Table II)
- a constraint that is anchored to the join node in the control flow graph (Rule (15) in Table II)
- a predicate constraint anchored to an out-going control flow graph edge of a conditional branch instruction (Rules (16)-(17) in Table III)
- a predicated linear inequalities constraint anchored to an integer comparison operation (Rules (18)-(29) in Table IV)
- a predicated predicate anchored to a boolean logic operation (Rules (30)-(33) in Table IV)

With the exception of rule (15), the constraints are dictated by the rule and the instruction to which it is attached. The validity of a claim depends only on the semantics of the instruction to which the claim is anchored and the satisfaction of any proof obligations included in the claim rule. If the proof



obligations are discharged, then the claim is guaranteed to be valid in the region dominated by that claim's anchor.

The proof obligations can be discharged with annotated proofs consisting of:

- axiomatic constraints
- other claimed constraints which dominate the proof obligation
- the addition combinator (+)
- the modus ponens combinator (MP)

All proof obligations, except for the ones associated with (15), must be proven using only those constraints that have been claimed to be valid at a program point that dominates the claim which the proof obligation is associated with.

2.2. Linear Inequality Claims

Given the earlier discussion of linear inequalities, the rules (1)–(4) and rules (7)–(14) in Table II, should be fairly straight forward. Rules (5) and (6) say that if there has been a successful array access, then the index must be within the array's bounds; this requires no proof obligation since the first array access will either have been proved to be safe or dynamically checked. The proof obligations in (9)–(14) serve to ensure that constraints are derived from addition, subtraction, and multiplication only when they cannot be invalidated by integer overflow or underflow.

2.3. Boolean Predicates

In Java, if statements and while, do, and for loops create regions that are controlled by a boolean expression. In SafeTSA, the boolean controlling a control structure is always specified as a boolean variable, which may be a constant, a parameter, or a variable created by a comparison operator (i.e., ==, !=, <=, >=, <, or >) or a boolean logic operator (i.e., &, ^, or |).[†]

If the boolean is a variable created by the integer comparison operations: ==, <=, >=, <, > then a linear inequality constraint can be derived that is true if the boolean variable is true. Similarly, if it is a variable created by the integer comparison operations: !=, <=, >=, <, > then a linear inequality constraints can be derived which is true if the boolean variable is false. The truth of these inequality constraints is thus predicated on the boolean variable (as shown in Rules (18)–(29) in Table IV).

Predicate constraint (i.e., a boolean variable or its negation) can be derived from conditional branches that depend on a boolean variable. As shown in (16) and (17) of Table III, if the boolean b controls a conditional branch, and then the predicate b may be asserted on the control flow graph edge corresponding to the branch being taken, and therefore may be used in the region dominated by that edge. Similarly, the predicate $\neg b$ may be asserted on the control flow graph edge corresponding to the branch not being taken. Therefore, linear inequalities predicated on a boolean variable being true or false, can be used in proof obligations anchored within the region that is control dependent upon that boolean variable being true or false, respectively.

[†]In SafeTSA, the short-circuit aspect of && and || is handled with a synthesized expression-level if control structure along with a non-short-circuiting "and" or "or" instruction.



Table II. Claim Rules for Linear Inequality Constraints

Instruction Form	Rule	Claim	Proof Obligations
Arrays			
$a[i]$	(0)	access within bounds	$\begin{aligned} [[i] - [a.length] + 1] &\leq 0 \\ -[[i]] &\leq 0 \end{aligned}$
$x = a.length$	(1)	$[[x]] - [a.length] \leq 0$	—
	(2)	$-[[x]] + [a.length] \leq 0$	—
$a = \text{new } C[x]$	(3)	$[[x]] - [a.length] \leq 0$	—
	(4)	$-[[x]] + [a.length] \leq 0$	—
$a[i]$	(5)	$[[i] - [a.length] + 1] \leq 0$	—
	(6)	$-[[i]] \leq 0$	—
Algebraic			
$x = y$	(7)	$[[x]] - [[y]] \leq 0$	—
	(8)	$-[[x]] + [[y]] \leq 0$	—
$x = y + z$	(9)	$[[x]] - [[y]] - [[z]] \leq 0$	$-[[y]] - [[z]] + \text{MIN} \leq 0$
	(10)	$-[[x]] + [[y]] + [[z]] \leq 0$	$[[y]] + [[z]] - \text{MAX} \leq 0$
$x = y - z$	(11)	$[[x]] - [[y]] + [[z]] \leq 0$	$-[[y]] + [[z]] + \text{MIN} \leq 0$
	(12)	$-[[x]] + [[y]] - [[z]] \leq 0$	$[[y]] - [[z]] - \text{MAX} \leq 0$
$x = c * y$	(13)	$[[x]] - c[[y]] \leq 0$	$-c[[y]] + \text{MIN} \leq 0$
	(14)	$-[[x]] + c[[y]] \leq 0$	$c[[y]] - \text{MAX} \leq 0$
ϕ-functions			
$x_1 = \phi(x_0, x_2)$	(15)	$c[[x_1]] + \dots \leq 0$	$\begin{aligned} c[[x_0]] + \dots &\leq 0 \\ c[[x_2]] + \dots &\leq 0 \end{aligned} *$

Where x , y , and z are Java variables of type `int`, a is a Java variable containing an array reference, and c is an integer constant
 * proof obligation is based on the claimed constraint and all of the ϕ -functions in the join node. See the text for detail.

Table III. Claim Rules for Conditional Branches

Instruction Form	Rule	Claimed Constraint
$\text{if } b \text{ goto } \dots$ <i>[branch taken]</i>	(16)	b
$\text{if } b \text{ goto } \dots$ <i>[branch not taken]</i>	(17)	$-b$



Table IV. Claim Rules for Predicated Inequalities

Instruction Form	Rule	Claimed Constraint
$b = x == y$	(18)	$b \Rightarrow \quad [[x]] - [[y]] \leq 0$
	(19)	$b \Rightarrow \quad -[[x]] + [[y]] \leq 0$
$b = x != y$	(20)	$\neg b \Rightarrow \quad [[x]] - [[y]] \leq 0$
	(21)	$\neg b \Rightarrow \quad -[[x]] + [[y]] \leq 0$
$b = x <= y$	(22)	$b \Rightarrow \quad [[x]] - [[y]] \leq 0$
	(23)	$\neg b \Rightarrow \quad -[[x]] + [[y]] + 1 \leq 0$
$b = x < y$	(24)	$b \Rightarrow \quad [[x]] - [[y]] + 1 \leq 0$
	(25)	$\neg b \Rightarrow \quad -[[x]] + [[y]] \leq 0$
$b = x >= y$	(26)	$b \Rightarrow \quad -[[x]] + [[y]] \leq 0$
	(27)	$\neg b \Rightarrow \quad [[x]] - [[y]] + 1 \leq 0$
$b = x > y$	(28)	$b \Rightarrow \quad -[[x]] + [[y]] + 1 \leq 0$
	(29)	$\neg b \Rightarrow \quad [[x]] - [[y]] \leq 0$
$b = c \ \& \ d$	(30)	$b \Rightarrow c$
	(31)	$b \Rightarrow d$
$b = c \ \ d$	(32)	$c \Rightarrow b$
	(33)	$d \Rightarrow b$

Predicates associated with a boolean variables may also be inferred from other boolean predicates (associated with logic operators) using Rules (30)-(33) in Table IV.

2.4. Merging Constraints at Join nodes

Rule (15), which is associated with join nodes (i.e., nodes in the control flow graph that have more than one predecessor and may contain ϕ -functions), requires some explanation. This claim rule is a consequence of the instructions being in SSA form, and exists to allow constraints that are valid in all of the join node's predecessors to the program region dominated by the join node, even though the join node may not be dominated by any of the join node's predecessors. This can be used to annotate loop invariants, but can also be used to show that some constraint holds after an if statement because it holds at the end of the then part of the if statement and at the end of the else part of the if statement.

Rule (15) is unique in a few ways. First, the claim is anchored to a join node as a whole rather than to any individual ϕ -function contained inside of it. Second, the constraint being claimed can



be any constraint (which must be described completely in the annotation) as long as the required proof obligations are discharged. Third, one proof obligation is anchored to each of the join node's predecessors and can only use the claims which are valid at the branch instruction in that predecessor. Fourth, the proof obligations are obtained by taking the claimed constraint and substituting, for each occurrence of variables which are on the left-hand side of ϕ -functions in the join node, the variable on the right-hand side of that ϕ -function that is associated with that proof obligation's anchor.

2.5. Representing Proofs

Proof obligations can be discharged by providing a sequence of references to universal constraints, claimed constraints, and constraint combinators. There are two primitive constraint combinators:

+ (**addition**): *inequality + inequality = inequality* which takes two inequalities and combines them by adding them together to produce a new linear equality. For example,

$$\begin{array}{r} -i_2 + i_1 + 1 \leq 0 \\ + \quad -i_1 \leq 0 \\ \hline -i_2 + 1 \leq 0 \end{array}$$

MP (Modus Ponens) takes a predicated constraint, and the predicate on which the constraint is predicated, and yields an unpredicated constraint. For example,

$$\frac{\text{MP} \quad \begin{array}{c} b \Rightarrow i_1 - x_0 + 1 \leq 0 \\ b \end{array}}{i_1 - x_0 + 1 \leq 0}$$

In any particular proof, only the axiomatic constraints and constraints that are from claims that dominate the proof obligation can be used.

2.6. An Example: Bounds Check Elimination for an Idiomatic Java Loop

Consider the Java function:

```
void f(int a[]) {
    int sum = 0;

    for (int i = 0; i < a.length; i++)
        sum = sum + a[i];
}
```

Since the loop variable i is monotonically increasing, with an initial value of 0 and a maximum value one less than $a.length$, the value i should range from 0 to $a.length - 1$, so the access $a[i]$ should never cause bounds check exception.

Figures 1 and 2 shows how our annotation scheme can be used to prove this fact. The first column of Figure 1 shows the function body translated into SSA form. (This example is arranged such that each instruction is dominated by all of the instructions above it, so each claim is valid from its appearance down.) The second column indicates the claim rule being used any proof obligations resulting from that claim. Figure 2 shows the proof used to fulfill those proof obligations.



Instructions	Rule	Claims	Obligations
1 Init:			
2 $sum_0 \leftarrow 0$			
3 $i_0 \leftarrow 0$	(8)	Ⓐ $-i_0 - 0 \leq 0$	
4 Loop:	(15)	Ⓑ $-i_1 \leq 0$	Ⓛ1 $-i_0 \leq 0$, Ⓛ2 $-i_2 \leq 0$
5 $sum_1 \leftarrow \phi(sum_0, sum_1)$			
6 $i_1 \leftarrow \phi(i_0, i_2)$			
7 $t_0 \leftarrow a_0.length$	(7)	Ⓒ $t_0 - a_0.length \leq 0$	
8 $t_1 \leftarrow i_1 < t_0$	(24)	Ⓓ $t_1 \Rightarrow i_1 - t_0 + 1 \leq 0$	
9 if t_1 then goto Body else goto Done			
10 Body:	(16)	Ⓔ t_1	
11 $t_2 \leftarrow load\ a_0[i_1]$	(0)	within bounds	Ⓛ3 $-i_1 \leq 0$ Ⓛ4 $i_1 - a_0.length + 1 \leq 0$
12 $sum_2 \leftarrow sum_1 + t_2$			
13 $i_2 \leftarrow i_1 + 1$	(10)	Ⓕ $-i_2 + i_1 + 1 \leq 0$	Ⓛ5 $i_1 + 1 - MAX \leq 0$
14 repeat Loop			
15 Done:			

Figure 1. Claims and Proof Obligations for an Idiomatic For Loop

$\begin{array}{l} \boxed{1} \quad -i_0 \leq 0 \\ \text{anchored at end of Init} \\ \therefore \text{Ⓐ} \end{array}$	$\begin{array}{l} \boxed{4} \quad i_1 - a_0.length + 1 \leq 0 \\ \therefore \quad t_1 \Rightarrow i_1 - t_0 + 1 \leq 0 \quad \text{Ⓓ} \\ \text{MP} \quad \frac{t_1}{i_1 - t_0 + 1 \leq 0} \quad \text{Ⓔ} \\ + \quad \frac{t_0 - a_0.length \leq 0}{i_1 - a_0.length + 1 \leq 0} \quad \text{Ⓒ} \end{array}$
$\begin{array}{l} \boxed{2} \quad -i_2 \leq 0 \\ \text{anchored at end of Body} \\ \therefore \quad -i_2 + i_1 + 1 \leq 0 \quad \text{Ⓕ} \\ + \quad \frac{-i_1 \leq 0}{-i_2 + 1 \leq 0} \quad \text{Ⓑ} \\ + \quad \frac{-1 \leq 0}{-i_2 \leq 0} \quad \text{Ⓔ} \end{array}$	$\begin{array}{l} \boxed{5} \quad i_1 + 1 - MAX \leq 0 \\ \therefore \quad t_1 \Rightarrow i_1 - t_0 + 1 \leq 0 \quad \text{Ⓓ} \\ \text{MP} \quad \frac{t_1}{i_1 - t_0 + 1 \leq 0} \quad \text{Ⓔ} \\ + \quad \frac{t_0 - a_0.length \leq 0}{i_1 - a_0.length + 1 \leq 0} \quad \text{Ⓒ} \\ + \quad \frac{a_0.length - MAX \leq 0}{i_1 + 1 - MAX \leq 0} \quad \text{Ⓓ} \end{array}$
$\begin{array}{l} \boxed{3} \quad -i_1 \leq 0 \\ \therefore \text{Ⓑ} \end{array}$	

Figure 2. Proofs for the Claims in the Idiomatic Loop



3	(8)	
4	(15):	$-i_1 < 0 \quad \textcircled{A}, \textcircled{F} + \textcircled{B} + \textcircled{2}$.
7	(7).	
8	(24).	
9-10	(16).	
11	(0)	$\textcircled{B}, \textcircled{D} \text{MP} \textcircled{E} + \textcircled{C}$.
13	(10)	$\textcircled{D} \text{MP} \textcircled{E} + \textcircled{C} + \textcircled{3}$.

Figure 3. Information Annotated for the Idiomatic Loop

It is important to note, however, that not all of this information needs to be explicitly transmitted. Each annotation needs to indicate the claim rule, but once this is indicated, except for the constraint for \textcircled{B} , the actual constraints can be derived from just the instruction and the claim rule. Similarly none of the proof obligations need to be included in the annotation, since they are derived from the instruction and the claim rule, and the proofs only need to include the constraints they are using by reference. Figure 3 shows the information that actually needs to be encoded for our example program.

3. Verifying Annotations

Verification of these linear constraint annotations is straight forward. The system merely requires a pre-order traversal of the program's dominator tree, during which a list of active claims is maintained. When an annotation is encountered, the claim rule is checked against the instruction it is anchored to. If the type of instruction does not match the claim rule, if the referenced claims are not in the active list, or if the referenced claims do not match the kinds of constraints required by a combinator, then the annotation will be rejected. Otherwise, the proofs are checked by loading referenced claims from the active list, applying the indicated combinators, and computing a resulting proof. If this discharges the proof obligation for that claim, then the claim is added to the active list until it no longer dominates the current node (claims are added and removed from the active list in LIFO order).

4. Implementation and Results

To create a prototype Verifiable Bounds Check Elimination (VBCE) system using our annotation framework, we implemented an annotation generator and annotation verifier within the SafeTSA system, which is based on Jikes RVM 2.2.0. The annotation generator performs range analysis using a modified version of Fourier-Motzkin Variable Elimination (FMVE) and derives proofs from the elimination process. Our algorithm extends FMVE to take into account cycles involving SSA ϕ functions and the proper scoping of constraints derived from control-dependencies. The verifier follows the algorithm outlined in the previous section.



We evaluated our prototype system using the Java Grande Forum benchmarks [3].[‡] These benchmarks were compiled into SafeTSA and optimized using common subexpression elimination, which eliminates duplicate bounds checks using SafeTSA's safe-element-reference type [16]. This version of each class is used as the baseline for the evaluation.

For comparison, we also implemented Chen and Kandemir's algorithms for range analysis, verification, and bounds-check elimination using a dataflow analysis framework (VDF) [4]. This allows us to compare the two approaches to verifiable bounds check elimination in terms of precision and in terms of performance. Since VDF utilizes only a small subset of difference constraints, VBCE is expected to be more precise than VDF, but possibly, at a higher runtime cost.

4.1. Time Required for Verification and Analysis

In our first experiment, we measure the time required by VBCE and VDF to analyze and to verify the analysis results for the classes in the Java Grande Forum Benchmarks. These measurements were made running the VBCE and VDF algorithms in Sun's Java 2 SE 1.6.0.04 under a Linux 2.6.24 kernel on a machine with a 2.13 GHz Core 2 Duo processor and 2GB of RAM. Each measurement was taken 50 times and the lowest of those 50 times was used; this procedure is intended to minimize the indeterminism due to system processes and garbage collection. The results are shown in Figure 4.

By comparing the analysis and verification times, we can see that using verification instead of analysis substantially reduces the overhead of performing runtime bounds-check elimination for both VBCE and VDF. Using VBCE to verify the results of our FMVE-based analysis instead of performing the analysis at runtime reduces the runtime overhead by $5\times$ to $72\times$. This makes using the FMVE-based algorithm feasible where it would otherwise be prohibitively expensive. The reduced verification overhead (a couple milliseconds at most) is quite small compared to the total benchmark execution times (which would be measured in seconds).

Comparing the verification times for VBCE and VDF in Figure 4, it can be seen that VDF verification is generally faster than VBCE verification. In some cases, the higher performance of VDF is attributable to the fact that it is less precise and so is verifying less bounds checks. This is the case, for example, in the Crypt, and Search/Game benchmarks. In most of the other cases, the simpler structure of VDF's difference constraints make it more efficient. Another reason that VBCE verification is sometimes slower than VDF verification is that VBCE considers integer overflow, whereas VDF assumes that no integer overflow occurs.

4.2. Precision

In a second experiment, we investigated the relative precision of the two frameworks by comparing the number of bounds checks that could be proved unnecessary in our VBCE framework with the number of bounds checks that could be shown unnecessary using VDF. Table V lists the benchmarks and benchmark classes analyzed, the number of bounds checks that remained in each class after producer-

[‡]The benchmarks were modified to make some of the array bounds limits be expressed as symbolic constants rather than passed in as parameters so that more array bounds could be eliminated with intraprocedural analysis. For brevity, we only report information for the benchmark classes that actually contain bounds checks.

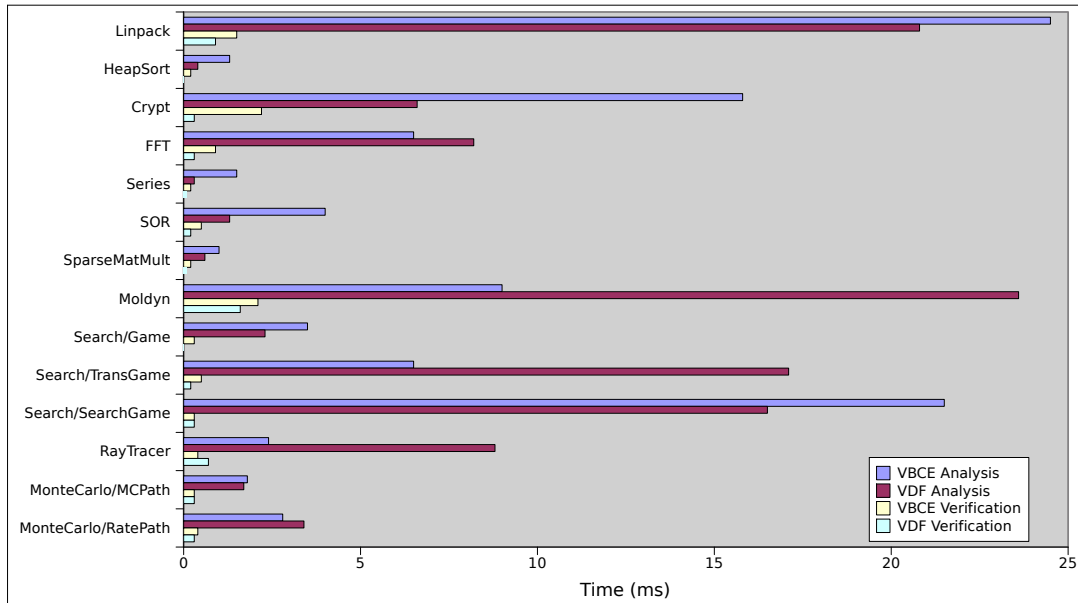


Figure 4. Time Required for Analysis and Verification with VBCE and VDF

side common subexpression elimination, and the number of bounds checks removed by VBCE and VDF respectively. As expected, VBCE was able to eliminate all of the bounds checks that VDF could eliminate. In addition, VBCE eliminated more bounds checks in about one-third of the benchmark classes.

For some benchmarks, there was a high number of bounds checks that could not be eliminated by either method. This was mainly due to properties that cannot be derived with intraprocedural analysis. In order to more precisely understand, why some of the bounds checks could not be eliminated, we examined the bounds checks found in the `cipher_idea` method of the `IDEATest` class in the `crypt` benchmark. (This is the primary computation kernel of that benchmark.) There were 28 bounds checks in this method: 22 in the outer loop and 6 in the inner loop. VBCE was able to eliminate eight of the bounds checks in the outer loop and none of the bounds checks in the inner loop. Six of the bounds checks in the inner loop and four of the bounds checks in the outer loop are prevented from being removed for two reasons. First, the index used in these accesses is an induction variable that is initialized before entering the outer loop and incremented several times inside the loops but is not involved in the loop condition. To prove that these bounds checks are unnecessary, would require finding loop invariants that constrain this variable relative to the loop index variable. Our FMVE-based analysis is not currently capable of deriving such proofs, although a proof of this type could be expressed in the verification framework. Second, these bounds checks are used on an array whose length is not directly connected to the loop condition. To relate the array's bounds to the loop



Table V. Bounds Check Eliminated in Benchmark Classes

Benchmark	Class	No. of Checks	Checks Eliminated	
			VBCE	VDF
lufact	Linpack	53	10	10
heapsort	NumericSortTest	3	1	1
crypt	IDEATest	66	17	2
fft	FFT	19	7	2
series	SeriesTest	4	2	2
sor	SOR	10	7	7
sparsematmult	SparseMatmult	7	2	2
modyn	md	21	15	15
search	Game	25	2	0
	TransGame	29	5	3
	SearchGame	24	3	1
raytracer	RayTracer	5	4	4
montecarlo	MonteCarloPath	6	4	4
	RatePath	18	4	4

condition would require interprocedural analysis. Similarly, the remaining eight bounds checks were for accesses to a third array whose length was also not related to the loop condition, but which should be the same length as another array that is involved in the loop condition. Interprocedural analysis would be required to establish that these arrays are of the same length. VDF found even less bounds checks on this benchmark, because it is restricted to difference constraints where the difference is between -7 and +8. Successfully analyzing many of the bounds checks for this benchmarks required tracking differences greater than 8.

In Linpack, several of the bounds checks could not be eliminated, because Java uses arrays of arrays to simulate multi-dimensional arrays, and it is often impossible for a static, intraprocedural analysis to determine that all of the rows of a two dimensional array are the same length, much less any particular length. In addition, there are several methods where there is a relationship between integer parameters and array parameters that would need to be known in order to eliminate array bounds checks. There is also a case where the return value of an arbitrary function is used as an index. An interprocedural analysis might be able to solve some of these problems, but Java's dynamic class loading and linking facilities could potentially invalidate such analysis.

Based on these results, we can derive some conclusions about the precision of VBCE and the verification performance of VBCE. In terms of precision, VBCE subsumes VDF. In addition, VBCE correctly handles integer overflow, whereas VDF does not. The verification performance of VBCE is generally not as fast as VDF, but both are quite fast. Compared to VDF, VBCE is more precise and more correct at a slightly higher (but still competitive) runtime verification cost.



5. Related Work

The concept of annotating programs with proofs of various properties that could then be verified was explored as part of Proof-Carrying Code [12, 11]. Proof-carrying code was based on first-order logic, and is thus more general than the linear inequality framework described here. The certifying compiler used with proof carrying code was capable of performing and proving the safety of several compiler optimizations including bounds check elimination [11]. The Special J compiler extended proof-carrying code to support the translation of Java programs into x86 assembly language [5].

We have taken some inspiration from proof-carrying code, but our focus is more limited: array bounds checks rather than program type safety, and linear constraints of integers rather than first order logic. Our more limited framework allows us to integrate with an existing Java virtual machine and continue to benefit from its machine-independence and dynamic class loading capability. The tighter focus makes our approach a direct replacement for runtime array bounds-check elimination. In addition, like the proposed virtual machine for proof-carrying code from Franz et al. [7], our approach should result in shorter, simpler proofs, and faster verification times than those based on a complete proof carrying code framework. Another difference is that our annotations are an optional addition to a type-safe mobile code format, and so unlike proof-carrying code, can be safely ignored by non-supporting virtual machine.

Besides the Special J compiler, there have been several works addressing the array bounds check problems in Java. Moreira et al. [10] used heavy-weight loop-based transformations and optimizations to optimize bounds checks in scientific applications; their goal was to provide a traditional static compiler for Java programs that provides performance approaching that of traditional optimizing compilers for Fortran, so their approach does not support just-in-time compilation and is not a general solution to the Java bounds check problem.

The ABCD algorithm [2] provides a global bounds check elimination based on extended-SSA form and difference constraints, it is quite efficient but has some limitations since it can only obtain difference constraints that can be overlaid onto the SSA graph. Menon et al. [9] extended the ABCD algorithm to produce optimized programs augmented with verifiable proof variables. The result is quite similar to our claims and proof obligations, but the verifier would be required to make judgements about facts (similar to our claims) using integer linear programming instead of checking an explicit proof; although verification performance was not reported by Menon et al. [9], we expect that integer linear programming would be slower than our approach.

Qian et al. [14] use an iterative dataflow analysis based on difference constraints to annotate bytecode with an indication of which bounds checks are unnecessary, but no mechanism is provided to verify that these annotations are correct. Chen and Kandemir [4] describe a method for annotating the fixed point of an iterative dataflow analysis of integer variable ranges which can then be verified using a single iteration of the same algorithm, but their constraints are limited to a subset of difference constraints.

Zhao et al.'s [19] optimization is restricted to limited loop forms (and is, therefore, less comprehensive than our approach) but is quite efficient during JIT compilation. Würthinger et al. [17] have developed a bounds check elimination for use in the HotSpot JIT compiler, which similarly identifies simple patterns in the source code but adds speculation to reduce the overhead of some of the bounds checks that cannot be completely eliminated statically. The base SafeTSA representation also provides special types to facilitate the producer-side removal of duplicate bounds checks [16], but this is more limited in precision than the approach described in this paper, and all of the bounds check



eliminations described in this paper were in addition to those that could be removed in SafeTSA using these types and common subexpression elimination.

With the notable exceptions of the proof-carrying code framework for the Special J compiler [5], prior work, including the ABCD algorithm [2] and Chen and Kandemir's verification system [4], has generally not addressed integer overflow as it exists in Java. Xi and Harper's [18] mention that the integer overflow problem could be solved if exceptions are raised on integer overflow, but this does not fit Java's semantics. Menon et al. [9]; mention that the integer programming used to verify their proof types operates in " \mathbb{Z} rather than 32- or 64-bit integers"; they do not, however, provide a solution or elaborate on any consequences of this. The proof-carrying code framework for Special J, however, considers integer overflow as part of rules for 32-bit signed and unsigned comparisons; these rules include an axiom that can be used to correctly prove incremented loop induction variables are within array bounds [5], but this axiom appears to be less general than rules (9)–(14) of the VBCE framework.

6. Conclusions and Future Work

In this paper, we have described verifiable annotations based on linear inequalities that facilitate bounds check elimination for Java programs. These verifiable bounds check elimination (VBCE) annotations can be produced by an annotator using thorough static analysis and then verified efficiently at runtime. This permits the just-in-time (JIT) compiler to safely eliminate array bounds checks during compilation without performing expensive runtime analysis. In addition, this framework, unlike much prior work, is sound in the presence of Java's integer overflow semantics.

We have implemented a prototype annotation generator and verifier within the context of the SafeTSA system. For comparison, we also implemented Chen and Kandemir's verifiable data flow framework for bounds check elimination (VDF) [4]. Experiments indicate that compared to VDF, VBCE is more precise at a comparable runtime verification cost.

In future work, we plan to explore extending the annotation framework to non-linear, speculative, and interprocedural constraints. We also plan to investigate integrating source-level annotations that could be used by a high-performance library author as a complement to static analysis.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their constructive comments and suggestions that helped improve the content and presentation of the paper. This research was supported in part by the Air Force Research Laboratory under grant F30602-02-1-0001 and NSF under grants EIA-0117255 and CCF-0702527.

REFERENCES

1. Wolfram Amme, Niall Dalton, Michael Franz, and Jeffery von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, volume 36, pages 137–147. ACM Press, June 2001.
2. Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 321–333, New York, NY, USA, 2000. ACM Press.



3. J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, May 2000.
4. Guangyu Chen and Mahmut Kandemir. Verifiable annotations for embedded java environments. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 105–114, New York, NY, USA, 2005. ACM Press.
5. Christopher Colby, Peter Lee, George Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107, New York, NY, USA, 2000. ACM.
6. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.
7. Michael Franz, Deepak Chandra, Andreas Gal, Vivek Haldar, Fermín Reig, and Ning Wang. A portable virtual machine target for proof-carrying code. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 24–31, New York, NY, USA, 2003. ACM.
8. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
9. Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, and Leaf Petersen. A verifiable ssa program representation for aggressive compiler optimization. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 397–408, New York, NY, USA, 2006. ACM Press.
10. José; E. Moreira, Samuel P. Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. *ACM Trans. Program. Lang. Syst.*, 22(2):265–295, 2000.
11. George Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
12. George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.
13. Thi Viet Nga Nguyen and Francois Irigoien. Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst.*, 27(3):527–570, 2005.
14. Feng Qian, Laurie J. Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for java. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 325–342, London, UK, 2002. Springer-Verlag.
15. Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, 2005.
16. Jeffery von Ronne, Wolfram Amme, and Michael Franz. An inherently type-safe ssa-based code format. Technical Report CS-TR-2006-004, Computer Science, The University of Texas at San Antonio, 2006.
17. Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the java hotspot client compiler. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 125–133, New York, NY, USA, 2007. ACM.
18. Hongwei Xi and Robert Harper. A dependently typed assembly language. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 169–180, New York, NY, USA, 2001. ACM Press.
19. Jisheng Zhao, Ian Rogers, Chris Kirkham, and Ian Watson. Loop parallelisation for the jikes rvm. In *Proceedings of the Sixth International Conference on Parallel and Distributed Computing (PDCAT'05)*, pages 35–39. IEEE Computer Society, 2005.