A Verifiable, Control Flow Aware Constraint Analyzer for Bounds Check Elimination

David Niedzielski Andreas Gampe Jeffery von Ronne Kleanthis Psarris

Department of Computer Science
The University of Texas at San Antonio
{dniedzie,agampe,vonronne,psarris}@cs.utsa.edu

Abstract

The Java programming language requires that out-of-bounds array accesses produce runtime exceptions. In general, this requires a dynamic bounds check each time an array element is accessed. However, if it can be proven that the array index does not exceed the bounds of the array, then the bounds check can be eliminated. We present a new algorithm based on extended Static Single Assignment (eSSA) form that builds a directed, weighted inequality graph representing control flow qualified, linear constraints among program variables derived from program statements. Our system then employs a customized depth-first search exploration algorithm to reason about relationships among variables, and provides a verifiable proof of it's conclusions. This proof can be passed to and verified by a runtime system to minimize the run time performance impact of this analysis. Our system is novel in several respects. It takes into consideration both control flow and data flow when analyzing the constraint system; it handles general linear inequalities instead of simple difference constraints; and it provides verifiable proofs for its claims. We present experimental results which demonstrate that this method eliminates more bounds checks, and when combined with runtime verification, results in a lower runtime cost than prior work. Overall, this method improves performance by up to about 10% on the reported benchmarks.

1. Introduction

The Java Virtual Machine specification requires that all array accesses are checked at run time, and that an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an *ArrayIndexOutOfBoundsException* to be thrown. However, these run time checks (especially those that occur for array accesses nested in loops) detract from performance, and so eliminating provably redundant checks can significantly increase the the performance of Java programs.

Redundant bounds check elimination for Java programs thus relies on the optimizer's ability to determine that the index used to index an array is always both strictly less than the array's upper bound and also non-negative. If the optimizer is able to make that determination, then the bounds check can be safely eliminated. If this analysis is performed at run-time, then it must obviously be done as efficiently as possible so as not to adversely impact performance. However, techniques that sacrifice precision for efficiency are not able to detect as many redundant checks, and the resulting run time performance is sub-optimal. Alternatively, if optimizations were able to be performed at compile time, then it is possible to devote more time to the analysis in an attempt to locate more redundant checks. However, any system that relies solely on compile time analysis is susceptible to a malicious optimizer that claims that an unsafe access is actually safe, thereby compromising system integrity. Therefore, the results of the compile time analysis must be encoded, communicated to the run time system, decoded, and verified at run time.

Traditional approaches generally sacrifice performance for precision, or vice versa. For example, in an attempt to reduce the cost of the analysis, some systems [3, 12] restrict the relationship among the variables to being only simple difference constraints of the form $x - y + c \leq 0$, where x and y are program variables, and c is a constant. Other more precise techniques [7, 2] are not efficient enough to be used at run time, nor are capable of producing verifiable proofs.

We have developed the Constraint Analysis System (CAS), a symbolic program constraint analyzer that detects redundant checks efficiently enough to minimally impact compilation/analysis time. CAS provides a proof of its redundancy claims that can be passed to and verified by a run time system. Furthermore, in order to exploit as much known information as is possible, CAS accepts and uses general linear relationships among variables. That is, our system processes linear constraints over program variables of the form

$$\sum_{1 \leq i \leq n} a_i x_i + c \leq 0 \tag{1}$$

CAS determines if a *proposed inequality constraint* (a logical relationship among program variables that is not directly implied by program statements) is consistent with those constraints that are *known* to hold because they are directly derived from statements in the program (*program constraints*). Using a specialized *Inequality Graph (IG)*, CAS attempts to determine if a proposed constraint cannot hold over a particular path in the CFG. If it can make this determination, it produces a sequence of constraints and variables (a path in the IG) corresponding to this control flow path. This set produces an inconsistent inequality $0 < c \leq 0$ when a sequence of *elementary row operations* are sequentially applied to the constraints in order to eliminate the variables. Exactly one of the n constraints in this sequence is a proposed constraint, whereas the remaining $n - 1$ constraints are program constraints. Since program constraints were derived from the program itself, they are by definition consistent with program semantics. Therefore, the constraint which is responsible for the inconsistency must be the proposed constraint. CAS produces only constraint and variable sequences that correspond to valid paths in the CFG. Note that CAS employs negated logic: proposed constraints state an array access is unsafe – in other words, when CAS finds an inconsistent proposed constraint, it means that an array access is *safe* along the identified path. Although this may seem counter-intuitive, it ensures that CAS

is able to supply a verifiable proof for its claims [1]. When exploring a path in the IG, CAS explores all possible control flow paths. Using a set of validity rules and essentially mathematical induction, CAS collapses and summarizes the effect of assignment loops such that its conclusions hold regardless of the number of loop iterations actually taken.

CAS is novel in several respects. First of all, it employs a new graph traversal algorithm which uses elementary row operations to sum edge weights and thereby reason about system consistency in a control flow aware way. Secondly, it distinguishes between edges in the graph derived from program statements from those proposed by the user in order to further restrict the search space. Lastly, CAS produces a verifiable sequence of constraint combinations leading to a contradiction that can be quickly inspected and validated as authentic by a properly-equipped run time system. This enables the run time system to assure itself that a proposed constraint cannot hold over a particular control flow path in the program.

2. Background

CAS is an efficient algorithm for determining the logical consistency of proposed constraints in the context of the semantics of a particular program. CAS combines and extends two techniques in a novel manner. CAS uses a program's eSSA form to build an *Inequality Graph (IG)* to represent the constraints that hold over the program's variables. It also uses a modified depth-first search exploration algorithm and elementary row operations to reason about variable relationships and deduce other relationships from those known to hold. We now briefly discuss these underlying ideas.

2.1 eSSA Representation

One of the unique characteristics of our system is that it takes into account enough program control and data flow to make semantically meaningful conclusions about how program variables relate to one another. It is able to do so because the inequality graph used by our system is derived from statements in an eSSA [3] representation of the program. eSSA extends traditional SSA [8] representation in that it adds so-called "π-assignments" to reflect what is known about variables as a result of control flow paths taken subsequent to conditional statements in the program.

eSSA's π-assignments create new variables (aliases) along control flow paths that are dominated by the outcome of a conditional expression. A constraint's reference to a π-variable allows the assertions CAS makes to be control flow path qualified. Said another way, the π-variables referenced by CAS's constraints identify a particular control flow path within the program. These π-variables are later combined via an SSA ϕ-assignment at a control flow merge point. For example, the following code fragment in SSA form:

```
read(a);
read(b)
if (a < b) then
   {Block A}
else
   {Block B}
end
{Rest of program}
```

would be transformed into the equivalent eSSA fragment:

```
read(a)
read(b)
if (a < b) then
   a_1 = pi(a)     /* a_1 < b_1 in Block A       */
   b_1 = pi(b)
   {Block A}       /* a,b replaced with a_1, b_1 */
else
   a_2 = pi(a)     /* a_2 >= b_2 in Block B      */
   b_2 = pi(b)
   {Block B}       /* a,b replaced with a_2, b_2 */
end
a_3 = phi(a_1, a_2)
b_3 = phi(b_1, b_2)
{Rest of program} /* a,b replaced with a_3, b_3 */
```

2.2 Elementary Row Operations

CAS reduces the problem of determining whether a proposed constraint holds given the logic of a program to deciding whether a system of linear inequalities is consistent or not. Several of the techniques used to solve this and related problems, such as Gaussian, Gauss-Jordan, and Fourier-Motzkin elimination, operate on the principle of iteratively reducing the system to simpler but equivalent forms until it is able to determine consistency (or solutions). One of the fundamental concepts underlying such techniques is that of *elementary row operations*.

Elementary row operations are transformations to the set of linear inequalities which do not change the solution set of the system. In essence, techniques such Gaussian elimination and Gauss-Jordan elimination reduce a matrix which represents a system of linear equations into an equivalent but simpler form by performing only the following row operations:

Row Switching A row within the matrix can be switched with another row

Row Multiplication Each element in a row can be multiplied by the same non-zero constant

Row Addition A row can be replaced by the sum of that row and a multiple of another row

A related technique that determines the consistency of a system of linear inequalities via elementary row operations is Fourier-Motzkin Variable Elimination.

2.3 Fourier-Motzkin Variable Elimination

Fourier-Motzkin Variable Elimination (FMVE) [13] is a technique used to determine if a system of linear inequalities has a real solution. It works by systematically applying elementary row operations to eliminate variables from the system until it the consistency is readily decidable. It eliminates a variable by combining all upper bounds on a variable x with all lower bounds on x. Whenever a lower bound on x is paired with an upper bound on x, a new inequality constraint is produced in which x does not appear. That is, given the following lower bound on x_j

$$\sum_{i \neq j} a_i x_i + c_1 \leq a_j x_j$$

and the following upper bound on x_j

$$b_j x_j \leq \sum_{i \neq j} b_i x_i + c_2$$

we can combine the two bounds to produce

$$b_j \sum_{i \neq j} a_i x_i + b_j c_1 \leq a_j \sum_{i \neq j} b_i x_i + a_j c_2$$

[1] Alternatively, the proposed constraints could imply that accesses are safe, and if CAS was *unable* to find an inconsistency, then the bounds check would be fully redundant. Although appropriate for a fully run-time implementation, this approach sacrifices verifiability, as CAS can only prove the validity of inconsistencies, not the validity of their *absence*

which does not contain the variable x_j. Note that only elementary row operations were used to effect this elimination, ensuring that the solution set remains unchanged. After all variables have been eliminated, the system contains constraints of the form $c \leq 0$. If all such constraints are valid (that is, c is negative or zero), then the original system is consistent. Otherwise, the original system is inconsistent.

FMVE's elimination of a variable x from a inequality system of n variables can be thought of as casting the shadow of an n-dimensional polyhedron onto the the x axis. After x's elimination, all constraints in which it appears are removed from the system. As mentioned earlier, the system will eventually be reduced to a system of inequalities between constants which can be easily inspected for consistency.

3. The Constraint Analysis System (CAS)

The Constraint Analysis System (CAS) takes a different approach. Rather than eliminating a variable from all constraints at the same time, CAS constructs an *inequality graph* (IG) and explores paths in this graph to reason about relationships among variables which cannot hold given the dataflow relationships in the program. We now describe these two facets of CAS: representation via the IG and reasoning via IG exploration.

3.1 The CAS Inequality Graph

CAS builds a directed, weighted inequality graph $IG = \{V, E\}$ to represent the constraints the program's logic imposes on program variables. The set of vertices $\{V\}$ represents variables in the program and the set of edges $\{E\}$ represents linear relationships (constraints) among those variables. As CAS explores a path in the IG, it calculates the cumulative path "weight" using elementary row operations instead of simple arithmetic addition. That is, when the subpath $e_j \rightarrow v \rightarrow e_k$ is traversed, CAS logically eliminates v from the sum of the two weights using elementary row operations. It is important to remember that in this context, the sum of the weights of two adjacent edges in CAS is not a simple arithmetic sum, but rather a linear inequality that results from the application of elementary row operations to the two edges to eliminate the intermediate vertex (variable).

3.1.1 Vertices in the IG

The IG contains a vertex for each variable in the eSSA representation of the program, plus a pseudo variable $\{ZERO\}$ which represents the constant 0. This special vertex ensures uniform edge representation, and is added to constraints whose coefficients are all negative or all positive: as each constraint c added to the system, CAS examines the coefficients of its terms. If all coefficients in c are negative, then the term $+\{ZERO\}$ is added to the LHS of c. Likewise, if all terms in c have positive coefficients, then the term $-\{ZERO\}$ is added to the LHS of c.

3.1.2 Edges in the IG

The edges in $\{E\}$ represent linear constraints among the program's variables. Rather than a simple constant, the weight of an edge in the IG is an inequality relating the source and target vertices. The IG is constructed such that an edge e_j arriving at a vertex x_n is a lower bound on x_n, while an edge e_k leaving x_n is an upper bound on x_n. Let c_{nj} represent the coefficient of variable x_n in edge e_j. Then $c_{nj} > 0$ and $c_{nk} < 0$. There are two kinds of edges in E:

program edges represent linear constraints over variables that are derived directly from program statements, and are therefore known to be mutually consistent with program semantics

proposed edges represent linear constraints over variables that are added by the user to be tested for consistency with the set of program edges, and which therefore may or may not be consistent with program semantics

Each edge in E has several properties, including:

type represents the kind of eSSA program statement from which this constraint is derived.

weight is a linear inequality that describes a constraint over a subset of variables in V

Modified Variable (MV) represents the variable $v \in \{V\}$ which is written in the statement from which this constraint is derived. If the statement is not an assignment, this property is null.

Read Set (RS) represents the set of variables $v \in \{V\}$ which are read in the statement from which this constraint is derived.

Direction for assignment constraints, a flag indicating whether the direction of the edge in the IG is consistent with or opposite the direction of data flow.

CAS distinguishes between various types of edges in order for it to determine valid paths through the IG. CAS recognizes the following edge types:

Inequality An inequality relationship between variables derived from conditional expressions

Equality An equality relationship between variables derived from conditional expressions

Assignment An assignment from a linear function of program variables to another program variable

PiAssignment An eSSA π-assignment of one variable to another

PhiAssignment An eSSA ϕ-assignment of one variable to another

PhiBackEdgeAssignment An eSSA ϕ-assignment of one variable to another resulting from a back edge in the CFG (forming a loop).

Because proposed edges represent logical relationships between variables and not assignments, their type is always **Inequality** (edges representing logical equalities are internally converted to a pair of **Inequality** edges). CAS proves a proposed edge $e_p \in \{E\}$ is inconsistent with the program by finding a path $p = e_1, v_1, e_2, v_2, \ldots, e_p, \ldots, e_n, v_n$ whose cumulative weight is a positive constant. If such a path is found, then p represents a control flow path over which e_p cannot hold. This in turn means that the bounds check represented by the proposed constraint is redundant. Otherwise, e_p is assumed to hold, and the corresponding bounds check cannot be eliminated. Any number of proposed constraints can be tested in this manner.

An edge from v to u with weight w exists in $\{E\}$ if there is a constraint c in which v has a positive coefficient while u has a negative coefficient. The edge weight w is simply the LHS of the inequality. For example, given the linear-inequality constraints

$$\begin{aligned} 2x_1 - 3x_2 + 4x_3 - x_4 + 5 &\leq 0 \\ -x_1 - 4x_3 + 2x_4 - 3 &\leq 0 \\ -x_3 + 2 &\leq 0 \\ x_4 - 3 &\leq 0 \end{aligned}$$

CAS logically creates the following edges in the IG:

Source	Target	Weight
x_1	x_2	$2x_1 - 3x_2 + 4x_3 - x_4 + 5$
x_3	x_2	$2x_1 - 3x_2 + 4x_3 - x_4 + 5 \leq 0$
x_1	x_4	$2x_1 - 3x_2 + 4x_3 - x_4 + 5 \leq 0$
x_3	x_4	$2x_1 - 3x_2 + 4x_3 - x_4 + 5 \leq 0$
x_4	x_1	$-x_1 - 4x_3 + 2x_4 - 3 \leq 0$
x_4	x_3	$-x_1 - 4x_3 + 2x_4 - 3 \leq 0$
$\{ZERO\}$	x_3	$-x_3 + \{ZERO\} + 2 \leq 0$
x_4	$\{ZERO\}$	$x_4 - \{ZERO\} - 3 \leq 0$

3.2 Path Exploration in CAS

In order to determine whether one or more proposed constraints are consistent given the semantics of the program. CAS explores paths in the IG, beginning with one of the proposed edges, and thenceforth traversing only program edges. Each path explored contains exactly one proposed edge, and all proposed edges are individually considered as part of CAS's testing. If in the course of the exploration, CAS discovers a path whose cumulative weight is an inconsistent inequality, then the path represents a control flow path over which the proposed constraint (edge) cannot hold. The specific path through the control flow graph on which the proposed constraint cannot hold is identified based on the π-variables traversed along the path.

CAS determines consistency by performing a depth-first search exploration of the graph, "summing" edge weights via elementary row operations that eliminate a variable from the weight as it is traversed in the path. It begins by exploring the edge arising from a proposed constraint. When CAS arrives at a vertex x_n via an inbound edge e_{in}, CAS examines the potential outbound edges. For each outbound edge e_{out} CAS compares the types of e_{in} and e_{out} along with information about the edges it has already explored along the path to restrict which outbound edges it is allowed to traverse. This restriction is necessary to ensure that paths in the IG represent valid control flow paths. A cycle p in the IG explored by CAS may cause all intermediate variables to be eliminated. If so, then $w(p)$ is of the form $c \leq 0$. If c is positive and non-zero, then CAS has detected an inconsistency, and the proposed edge in p represents a constraint which is not consistent with the program edges along the control flow path represented the p.

Vertices in the IG are traversed in a way that is consistent with program semantics. That is, before an outbound edge is selected after having arrived at a vertex via an inbound edge, a set of efficient tests (described next) are performed to determine whether the traversal is consistent with the semantics of the program. If it is, then the resulting path represents a plausible control flow path in the program. Otherwise, the edge is not traversed. This has a twofold effect:

1. Resulting paths through the inequality graph represent valid relationships between variables given the program control flow structure
2. Because the majority of edge traversals are prevented by the tests, the number of potential paths grows much more slowly. As CAS's running time depends on the number of constraints, this has an important performance impact.

When CAS is considering an outbound edge after having arrived at a vertex via a particular inbound edge along a path through the inequality graph, CAS checks the following conditions (called the *compatibility conditions*). The edge is discarded if any of the compatibility conditions are violated.

Condition 1: Multiple Writes to a Common Variable Consider the following code fragment:

```
x = y
...
x = z
```

Each of these assignments is represented in the CAS system as a pair of constraints:

$$\begin{aligned} x - y + 0 &\leq 0 \quad (2)\\ -x + y + 0 &\leq 0 \quad (3)\\ x - z + 0 &\leq 0 \quad (4)\\ -x + z + 0 &\leq 0 \quad (5)\\ & \quad (6)\end{aligned}$$

Since constraints 3 and 5 are lower bounds on x, while 2 and 4 are upper bounds on x, constraints 2 with 5 and 3 with 4 might be combined to produce, respectively,

$$\begin{aligned} z - y + 0 &\leq 0\\ y - z + 0 &\leq 0\end{aligned}$$

The combinations of these two constraints, in turn imply that $y == z$. However, the above combinations do not take into account program semantics. There is no relationship that can be assumed between variables z and y based on the above code fragment. Therefore, before performing a combination, CAS checks if the upper and lower bound constraints arise from assignment instructions that write to a common variable. If so, the combination is disallowed.

Condition 2: Mutually Exclusive π-assignments Consider the following code fragment:

```
if (x < y) then
   <if part>
else
   <else part>
end
```

An eSSA representation would be:

```
if (x < y) then
   x1 = pi(x)
   y1 = pi(y)
   <if part>
else
   x2 = pi(x)
   y2 = pi(y)
   <else part>
end
```

Each of the π-assignments, as well as the inequality relationships implied by the conditional expression are represented in the CAS system:

$$x1 - x + 0 \leq 0 \quad (7)$$
$$-x1 + x + 0 \leq 0 \quad (8)$$
$$y1 - y + 0 \leq 0 \quad (9)$$
$$-y1 + y + 0 \leq 0 \quad (10)$$
$$x1 - y1 + 1 \leq 0 \quad (11)$$
$$x2 - x + 0 \leq 0 \quad (12)$$
$$-x2 + x + 0 \leq 0 \quad (13)$$
$$y2 - y + 0 \leq 0 \quad (14)$$
$$-y2 + y + 0 \leq 0 \quad (15)$$
$$y2 - x2 + 0 \leq 0 \quad (16)$$

Note that, for example, constraints 8 and 13 are upper bounds on x, while constraints 7 and 12 are lower bounds on x. Likewise, constraints 10 and 15 are uppers bounds on y, while 9 and 14 are lower bounds on y. Combining corresponding bounds leads to results which imply that $x1 == x2$ and $y1 == y2$, even though no such relationship can be construed from the original code fragments. Therefore, CAS checks for and prevents combinations of constraints that arise from π-assignments from opposite sides of the same conditional statement.

Condition 3: Combining Inverted Constraints For simplicity, all constraints in CAS are uniformly represented as inequalities. This requires that program assignments and equality relationships be represented by a pair of inverted inequality relationships. Combining inverted constraints results in the useless constraint $0 \leq 0$. In an attempt to improve performance by limiting the number of constraint combinations performed, CAS checks if the upper and lower bounds arise from the same program statement. If so, the combination is bypassed. Using the example from Condition 2 above, this rule prevents the combinations of constraints 1 with 2, 3 with 4, 6 with 7, and 8 with 9.

Condition 4: Multiple Uses of the Same ϕ-assignments CAS conceptually explores the eSSA graph formed by program assignments and implied equality and inequality relationships to discover inconsistent paths introduced by proposed constraints. CAS explores all possible unique control flow paths through this graph in order to uncover all potential inconsistencies given the semantics of the program. Since any non-trivial program will produce a graph with cycles, CAS tracks its use of constraints which arise from ϕ-assignments. The CAS algorithm terminates when it has explored (used) all constraints derived from ϕ-assignments. When CAS is considering an upper bound and lower bound combination, it checks whether either is derived from a ϕ-assignment it has already used in the same path through the eSSA graph. If so, the combination is disallowed, eventually guaranteeing termination of the algorithm.

Condition 5: Consistent Control Flow direction Because assignment constraints are duplicated and inverted in CAS so that all constraints follow the less-than-or-equal-to-zero paradigm, each constraint in CAS which represents a dataflow has a "direction" flag associated with it that indicates whether the direction of the edge is consistent with or opposite data flow. For example, given the program assignment $a = b + 2$, two constraints are generated, $a - b - 2 \leq 0$ and $-a + b + 2 \leq 0$. The former constraint is represented by an edge in the IG from a to b with a weight of -2. Since the direction of this arc is opposite the flow of data, its direction flag is -1. Alternatively, the latter constraint is represented by an edge from b to a with a weight of $+2$. Since the direction of this edge is consistent with the direction of data flow, its direction flag is set to $+1$. Note that since equalities and inequalities do not involve data flow, their direction flag is always 0. This condition causes all edges in a path taken in the IG to have uniform direction (either $+1$ or 0, or else -1 and 0).

Condition 6: Unique Equality/Inequality Edges in Path This condition simply ensures that an equality or inequality edge is taken at most once in a path in the IG. Such re-traversals do not contribute anything new to the path, and preventing them improves performance.

Condition 7: Negative Weight Assignment Subcycles An inconsistency in CAS occurs when a cycle involving a proposed constraint is detected with a positive weight. CAS propagates edge weights with elementary row operations as a path is explored. If a negative weight assignment subcycle is detected (as in a count-down loop), CAS immediately abandons the path, since the negative weight cycle might be traversed an arbitrary number of times. Since each such traversal decrements the path weight, any positive weight CAS calculates before or after the cycle could be compensated for by a sufficient number of traversals of this subcycle. Therefore, CAS conservatively abandons the path when such a cycle is detected.

3.3 Integer vs. Real Solutions

CAS distinguishes between integer and real solutions to system of linear inequalities by maintaining two constant terms when calculating the cumulative weight for a path. The first and weaker constraint must be satisfied if there is a real solution to the system. The second constant term represents a stronger constraint which if satisfied guarantees an integer solution to the system. However, if the real constraint is satisfied but the integer solution is not, then there is definitely a real solution to the resulting system but there may or may not be an integer solution. This is the same idea behind the Omega Test's [11] algorithm.

3.4 Arithmetic Overflow

The CAS system finds symbolic solutions in the domain of integers $\mathbb{Z}$. Java's primitive integer type `int`, however, is restricted to integers that can be stored in 2's complement 32-bit words and "wrap around" when operations underflow/overflow. For this reason, the verification system requires supplementary proofs that the arithmetic operations in the source code from which constraints are drawn invalidate those constraints through arithmetic underflow or overflow. These are generated by examining the inconsistent paths generated by CAS, identifying the edges derived from arithmetic operations and recursively invoking the CAS on a proposed constraint representing the underflow or overflow condition.

3.5 Detailed Description

The CAS algorithm can intuitively be viewed as a graph exploration, where the nodes in the graph represent variables in the program, and the edges represent linear relationships between the vertices. An edge e arriving at a node v means that e is a lower bound on v (that is, v has a negative coefficient in the linear constraint represented by e). Likewise, an edge e leaving a node v represents a lower bound on v (that is, v has a positive coefficient in the linear constraint represented by e). Traversing the graph is performed by adding edge weights as each edge is traversed. Each edge contains sufficient information to allow the compatibility tests to determine whether a pair of edges can be traversed in sequence without violating the semantics of the program. There are essentially four steps to using the CAS system:

1. Add program constraints (edges)

```
1)     int A[] = new int[y]; /* y == A.length */
2)     x0 = 0
3) L:  x1 = phi(x0, x3)
4)     if (x1 >= y) goto E:
5)     x2 = pi(x1)
6)     y1 = pi(y) /* x2 < y2 */
7)     A[x2] = ...
8)     x3 = x2 + 1
9)     goto L:
10) E: x4 = pi(x1)
11)    y2 = pi(y) /* y2 <= x4 */ ...
```

Figure 1. Example code fragment in eSSA form

2. Add proposed constraints (edges)
3. Test the System for Consistency
4. Reset the system

Steps 2, 3, and 4 can be repeated as often as is necessary. We now consider each of these steps in turn.

3.5.1 Add Program Constraints

In this phase, program constraints are added to the system as they are parsed. For example, consider the following Java code fragment:

```
int A[] = new int[y];
for (int x = 0; x < y; x++)
{
   A[x] = ...
}
```

An equivalent eSSA representation is shown in Figure 1. From this representation, we can derive the program constraints shown in Table 1. For each constraint, the last column in Table 1 lists the statement in Figure 1 from which it is derived. The proposed constraints will be added in the next step.

The pseudo-variable $A.length$ is assigned based on the array allocation in statement 1. Constraints 11 and 20, which represent the inequality relationships between $x2$ and $y1$ and between $x4$ and $y2$, are based on the $TRUE$ and $FALSE$ outcomes, respectively, of the conditional in statement 4. The other constraints are derived in a straightforward manner from program assignments. Program constraints are added and added to the system via the function **build_base_system()** and **add_constraint()**.

3.5.2 Add Proposed Constraints

Once the program constraints have been added, proposed constraints are added. For example, suppose we wish to determine if the upper bound check in the array access statement `A[x2]` (statement 7 in Figure 1) is necessary. This is accomplished by adding and testing constraint(s) which states that the safety condition is violated ($A.length \leq x_2$). In the same way, to test that the lower bounds check is unnecessary, we also add the proposed constraint $x2 + 1 \leq 0$. The complete system of inequalities is shown in Table 1.

Table 2 shows the edges in the IG that results from the preceding constraints. The "type" column indicates the edge type (Pi=PiAssignment, Phi=PhiAssignment, PhiB=PhiBackEdgeAssignment, A=Assignment, I=Inequality) as well as the direction (I=Independent, R=Reverse, F=Forward).

The function **add_proposed_constraint()** adds a proposed constraint to a collection of proposed constraints. Note that since proposed constraints are the first edges traversed when building a path, they are not queued to variables.

Procedure `build_base_system`

```
/* Add constraints to the system as program
   statements are parsed                          */
Variables = {};
Constraints = {};
Proposed_Constraints = {};
foreach linear constraint c derived from a program's eSSA
representation do
    c = new Constraint();
    c.weight = inequality relating program variables;
    c.type = type of constraint (see Section 3.1.2);
    c.read = {Variables read by c};
    c.write = {Variables written by c};
    c.traversedNodes ={};
    c.traversedConstraints ={};
    add_constraint(c);
    if c.type ≠ Inequality then
        /* Create a inverted copy of c and add
           it                                     */
        r = copy of c with coefficients negated;
        add_constraint(r);
    end
end
```

Procedure `add_constraint(`c`)`

```
/* Add a new constraint to the system          */
if each term in c has a negative coefficient then
    add the term +{ZERO} to c;
end
if each term in c has a positive coefficient then
    add the term -{ZERO} to c;
end
foreach variable v referenced by c do
    if v ∉ Variables then
        Variables.add(v);
    end
    if c is an upper bound on v then
        v.UBs.add(c);
    end
end
Program_Constraints.add(c);
```

Procedure `add_proposed_constraint(`p`)`

```
if each term in p has a negative coefficient then
    add the term +ZERO to p;
end
if each term in p has a positive coefficient then
    add the term -ZERO to p;
end
Proposed_Constraints.add(p);
```

Constraint #	Constraint	Statement #
1	$x0 + 0 \leq 0$	2
2	$-x0 + 0 \leq 0$	2
3	$A.length - y + 0 \leq 0$	1
4	$-A.length + y + 0 \leq 0$	1
5	$x0 - x1 + 0 \leq 0$	3
6	$-x0 + x1 + 0 \leq 0$	3
7	$x2 - x1 + 0 \leq 0$	5
8	$-x2 + x1 + 0 \leq 0$	5
9	$y1 - y + 0 \leq 0$	6
10	$-y1 + y + 0 \leq 0$	6
11	$x2 - y1 + 1 \leq 0$	4
12	$x2 - x3 + 1 \leq 0$	8
13	$-x2 + x3 - 1 \leq 0$	8
14	$x3 - x1 + 0 \leq 0$	3
15	$-x3 + x1 + 0 \leq 0$	3
16	$x4 - x1 + 0 \leq 0$	10
17	$-x4 + x1 + 0 \leq 0$	10
18	$y2 - y + 0 \leq 0$	11
19	$-y2 + y + 0 \leq 0$	11
20	$y2 - x4 + 0 \leq 0$	4
21	$A.length - x2 + 0 \leq 0$	(Proposed)
22	$x2 + 1 \leq 0$	(Proposed)

Table 1. Program/Proposed Constraints for Example

Source	Target	Weight	Type/Dir
$x0$	$ZERO$	$x0 - ZERO + 0 \leq 0$	A/R
$ZERO$	$x0$	$-x0 + ZERO + 0 \leq 0$	A/F
$A.length$	y	$A.length - y + 0 \leq 0$	I/I
y	$A.length$	$-A.length + y + 0 \leq 0$	I/I
$x0$	$x1$	$x0 - x1 + 0 \leq 0$	Phi/F
$x1$	$x0$	$-x0 + x1 + 0 \leq 0$	Phi/R
$x2$	$x1$	$x2 - x1 + 0 \leq 0$	Pi/R
$x1$	$x2$	$-x2 + x1 + 0 \leq 0$	Pi/F
$y1$	y	$y1 - y + 0 \leq 0$	Pi/R
y	$y1$	$-y1 + y + 0 \leq 0$	Pi/F
$x2$	$y1$	$x2 - y1 + 1 \leq 0$	I/I
$x2$	$x3$	$x2 - x3 + 1 \leq 0$	A/F
$x3$	$x2$	$-x2 + x3 - 1 \leq 0$	A/R
$x3$	$x1$	$x3 - x1 + 0 \leq 0$	PhiB/F
$x1$	$x3$	$-x3 + x1 + 0 \leq 0$	PhiB/R
$x4$	$x1$	$x4 - x1 + 0 \leq 0$	Pi/R
$x1$	$x4$	$-x4 + x1 + 0 \leq 0$	Pi/F
$y2$	y	$y2 - y + 0 \leq 0$	Pi/R
y	$y2$	$-y2 + y + 0 \leq 0$	Pi/F
$y2$	$x4$	$y2 - x4 + 0 \leq 0$	I/I
$A.length$	$x2$	$A.length - x2 + 0 \leq 0$	I/I
$x2$	$ZERO$	$x2 - ZERO + 1 \leq 0$	I/I

Table 2. Inequality Graph (IG) for Example

3.5.3 Test the System for Consistency

Once the IG has been built, CAS begins its analysis by exploring paths within it. If a cycle in the IG is discovered whose weight consists of an inconsistent constraint of the form $0 < c \leq 0$, the CAS Analyzer returns the path through the IG that resulted in the inconsistency, as well as the resulting cumulative constraint. If the path includes π-variables, then the proposed constraint is inconsistent along the particular control flow path on which the π-variables were defined.

Procedure `visit`(*path, LB, v*)

Input: path: LIFO queue of edges and vertexes we've traversed
Input: LB: vector representing cumulative weight of path
`/* Perform a recursive depth-first search exploration of the graph */`
v = last vertex on path;
rc = continue_search;
if *LB is an inconsistent constraint of the form* $0 < c \leq 0$ **then**
    InconsistentPaths.add(path);
end
path.push(v);
foreach *outbound edge UB from v* **do**
    if *LB* $\longrightarrow$ *v* $\longrightarrow$ *UB does not violate any compatibility condition* **then**
        c = new constraint that combines LB and UB to eliminate v via elementary row operations;
        path.push(UB);
        if *c has one or more terms* **then**
            foreach *vertex u for which c is a LB* **do**
                rc = visit($path$, *c*, *u*);
            end
        else
            foreach *vertex u reachable via* UB **do**
                rc = visit($path$, *c*, *u*);
            end
        end
    end
    path.pop(UB);
end
path.pop(v);

Testing of the system for inconsistencies involves two functions. The function **visit(path, LB, v)** recursively explores the IG. Upon invocation, $path$ is a sequence of vertices (variables) and edges (constraints) traversed thus far. LB is a cumulative constraint representing the sum of all elementary row operations performed to eliminate the intermediate vertices along the taken path, and v represents the vertex (variable) in the IG that is next to be eliminated (traversed). Upon arrival at v via LB, **visit()** first checks if LB is an inconsistent constraint (that is, a constraint representing the illogical relation $0 < c \leq 0$. If so, the inconsistent path is added to a list of inconsistent paths discovered thus far. Next, **visit()** examines each outbound edge in the array $v.UBs$, and checks the compatibility conditions to determine if a particular UB is a valid edge to take considering the path taken thus far and the current vertex. If so, elementary row operations are applied to eliminate v from LB and UB, forming a new cumulative constraint c. This constraint represents the "traversal" of v from LB to UB. Next, **visit()** is recursively called for each vertex u reachable from c (i.e. each variable u for which c is a lower bound).

The function **test()** is the driver for the recursion. It iteratively selects each proposed constraint and initiates exploration via the call to **visit()**.

```
Procedure test
  /* Test each proposed constraint for
     consistency with base system                    */
  InconsistentPaths = {};
  foreach pc ∈ Proposed_Constraints do
      foreach Variable v for which pc is a LB do
          path = p, v;
          rc = visit(path, p);
      end
  end
  return InconsistentPaths;
```

3.5.4 Reset the System

Once all inconsistent constraints (if any) arising from the augmented system have been identified, the user can reset the system in preparation for another cycle of adding proposed constraints and testing the system. Resetting the constraint system simply removes all proposed constraints, and leaves the CAS system with just the base system, to which more proposed constraints can be added in an iterative fashion.

4. CAS Complexity

We now examine the computational complexity of the CAS algorithm. In the discussion that follows, let v and c represent the number of variables and constraints in the system, respectively. The complexity is dominated by the recursive **visit()** routine presented in Section 3.5.3.

4.1 Worst Case Scenario

The worst case occurs if each variable in the system occurs in each constraint, and each constraint represents an upper bound on half of the variables, and lower bounds on the other half. Because ϕ-assignments and π-assignments have exactly two terms, this scenario precludes practical programs with any branches or conditionals. Nonetheless, it does demonstrate that CAS is potentially prohibitively expensive in the theoretical worst case. Each of the v eliminations necessary to create a reduced constraint produces $\frac{c}{2}^2$ new constraints. It takes $O(v)$ time to combine the terms, as well as carry out the compatibility tests in lines 24-29. Thus, the total worst case running time is:

$$O((\frac{c}{2})^{2^v} v)$$

4.2 Difference Constraints

In the best (and much more likely scenario), every constraint is a difference constraint, and each constraint is a simple inequality relationship between two variables of opposite signs and a constant. Previous work [3] suggests simple difference constraints are sufficient to eliminate the majority of array bound checks. We again assume no conditionals or branches. In this case, in a system with c constraints, there can be no more than $\frac{c}{2} + 1$ variables, since each assignment is represented by two constraints, and each variable appears in exactly two assignments, as follows:

$$\begin{aligned} v_2 &= v_1 + c_1 \\ v_3 &= v_2 + c_2 \\ &\dots \\ v_{n-1} &= v_{n-2} + c_{n-2} \\ v_n &= v_{n-1} + c_{n-1} \end{aligned}$$

In this case, each variable v_j appears in four constraints, as follows:

1. $v_j - v_{j-1} - cj - 1 \leq 0$
2. $-v_j + v_{j-1} + c_{j-1} \leq 0$
3. $v_{j+1} - v_j - c_j \leq 0$
4. $-v_{j+1} + v_j + c_j \leq 0$

In order to eliminate a variable, four potential combinations could be done: 1+2, 3+4, 2+3, and 1+4. However, condition 3 from section 3.2 prevents combinations 1+2 and 3+4. Since two combinations per variable elimination are done, and since v variables are eliminated, and if it takes $O(v)$ to perform the compatibility tests, the complexity becomes $O(v^2)$. This does not change even if every variable is involved in a fixed number of conditional branches.

We have heretofore assumed that it takes $O(v)$ time to perform the compatibility tests in the filter. This assumes a naive implementation of the tests, where bit strings are maintained for ϕ-variables and π-sources, as well as variables read, written, and eliminated. Actually, different data structures could be used that would require only $O(|e|)$ time per variable elimination, where e represents the number of eliminations performed so far. Thus, the first elimination would take $O(1)$ time, whereas the last elimination would take $O(v)$ time. This represents only a marginal improvement however, as the asymptotic run time is still $O(v^2)$x. Furthermore it is not clear that this theoretical improvement in theoretical run time would result in any practical improvement over the naive implementation because of the extra complexity inherent in maintaining more complicated data structures.

5. Experimental Results

CAS was in implemented in the SafeTSA compiler [1] and was used to identify redundant bound checks and produce verifiable proofs of its redundancy claims. These proofs were encoded in the form of annotations and were passed to a run-time system for verification. Additionally, we added an annotation verifier and annotation directed optimization into the SafeTSA classloader and JIT compiler of the SafeTSA virtual machine (which is based on Jikes RVM 2.2.0).

For comparison, we also took the implementation of ABCD [3] found in the Jikes RVM Optimizing bytecode compiler [5] and ported it to work with the SafeTSA data structures. Several features of SafeTSA required extensions to the original algorithm, most notably, SafeTSA's type infrastructure. SafeTSA includes several explicit type coercion instructions to simplify type-checking; these were accommodated for in ABCD by extending ABCD's existing Global Value Numbering system. In addition, since the existing ABCD implementation analyzed only upper-bounds checks, we extended ABCD to support lower-bounds checks.

We evaluated our prototype system using the Java Grande Forum benchmarks [4].[2] These benchmarks were compiled into SafeTSA and optimized using common subexpression elimination, which eliminates duplicate bounds checks using SafeTSA's safe-element-reference type. This version of each class was used as the baseline to which CAS and ABCD were applied.

All of the experiments were conducted on a 1.5GHz G4 PowerMac with 1GB of RAM running a Linux 2.6.15 kernel. All timing measurements were made by repeatedly running the benchmark program in a fresh virtual machine at least 200 times. The first fifty

[2] 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 conservative, intraprocedural analysis.

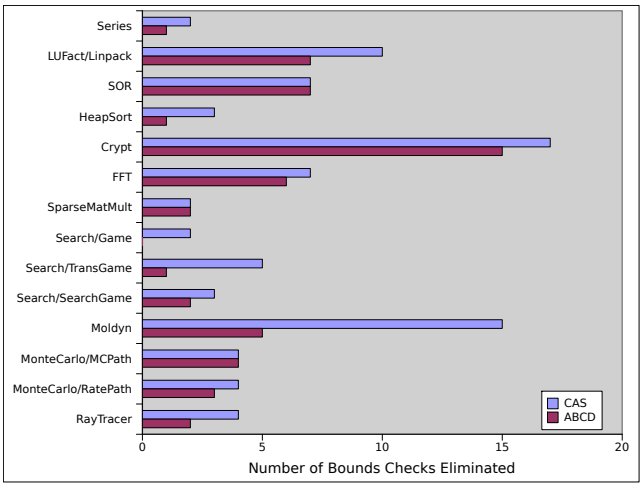

Figure 2. Precision of CAS and ABCD

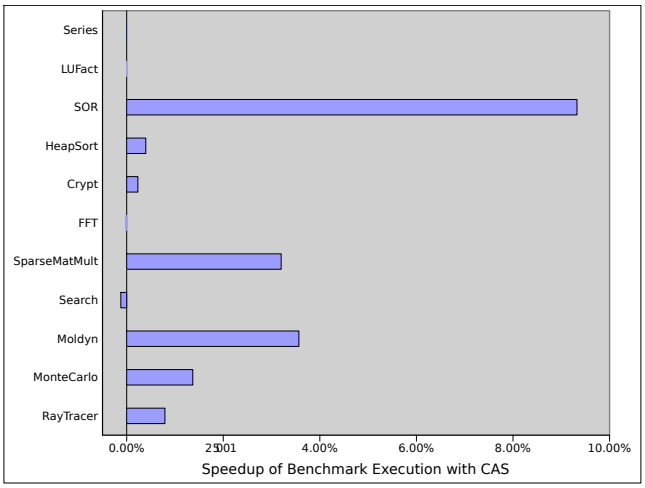

Figure 4. Overall Execution Speedup with CAS

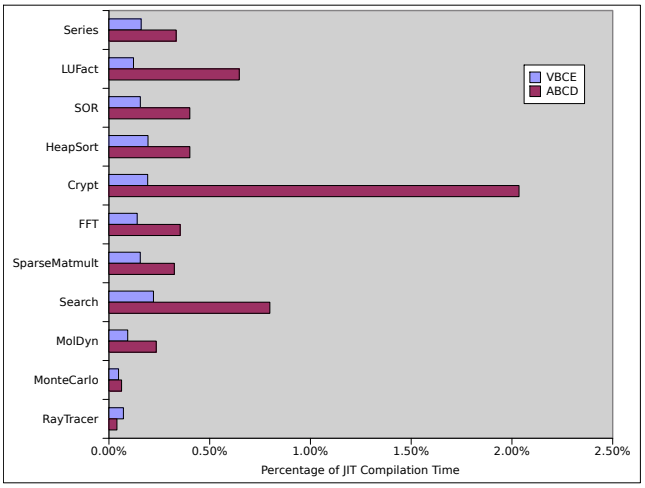

Figure 3. Runtime Cost of CAS and ABCD

runs were discarded and the mean of the subsequent runs is reported.

Figure 2 shows the number of bounds checks (beyond those eliminated by common subexpression elimination) that were eliminated by CAS and ABCD, respectively, in each of the benchmark classes. Since CAS can reason about general linear constraints while ABCD is limited to a subclass of difference constraints (which is a subclass of linear constraints), one would expect CAS to eliminate more bounds checks than ABCD. For our benchmarks, all of the bounds checks removed by ABCD were also eliminated by CAS, and in nearly three-quarters of the classes, CAS's extra precision results in more eliminated bounds checks. Notably, for the the Moldyn benchmark, CAS was able to eliminate 15 bounds checks where ABCD could only remove 5. Manual inspection of several bounds checks revealed that these were bounds checks that the ABCD is unable to prove because ABCD can only reason about a restricted form of difference constraints. In particular, it only supports constraining the variable being defined by an instruction relative to the variables being used by that same instruction. This restriction speeds up ABCD significantly (since it allows ABCD to piggy back on the existing compiler data structures) but does so at the cost of precision.

Figure 3 shows the cost of the runtime verification of CAS's proofs vs. the runtime use of ABCD as a percentage of baseline JIT compilation time. For CAS this is primarily the time required to verify annotations at run-time, whereas for ABCD this is the time required to carry out the ABCD algorithm. With one exception (RayTracer), verification has a lower runtime cost than ABCD. For some of the benchmarks (notably Crypt, Search, and LUFact), verification is several times faster. The primary reason that verification outperforms the ABCD algorithm is that the verification component only needs to verify those bounds checks that are actually unnecessary, whereas ABCD checks all bounds checks. Compared to the total JIT compilation time, however, both methods impose a very small runtime cost (up to about 0.2% for verification and up to about 2.0% for ABCD).

Figure 4 shows the speedup in total benchmark execution time resulting from the use of verification compared to the baseline SafeTSA version of the benchmark. The bounds check elimination resulted in improvements of nearly 10% on the SOR benchmark and over 3% on the SparseMatMult and MolDyn benchmarks. On the other benchmarks the speedup was small or negligible; this is because most of the bounds checks that could be eliminated in those benchmarks were not in the benchmark's inner loop. Since time spent in verification was relatively small (a couple milliseconds) compared to the total execution time (15s–200s), the effect of verification cost is not noticeable.

6. Related Work

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 [3] provides a runtime global bounds check elimination based on extended-SSA (eSSA) form and difference constraints, it is quite efficient but has some limitations since it can only obtain difference constraints that can be overlayed onto the eSSA graph. Menon et al. [9] extended the ABCD algorithm to produce optimized programs augmented with verifiable proof variables. Like ABCD, CAS represents programs using eSSA, but CAS's inequality graph is more general and allows it to reason

about general linear inequalities. Linear programming techniques such as Fourier-Motzkin Variable Elimination [13] also handle such inequalities, but CAS is more efficient because of the path pruning that takes place as the graph is explored. Furthermore, our system takes into account control flow when choosing paths through the graph as well as when presenting proofs.

Qian et al.[12] use an iterative dataflow analysis based on difference constraints to annotate bytecode with an indication of which bounds checks are unnecessary, but it does not provide verifiable proofs of its claims. Chen and Kandemir [6] describe a method for annotating the fixed point of a 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 [15] 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. [14] 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.

7. Future Work and Conclusion

We present in this paper a lightweight theorem prover suitable for determining whether proposed relationships among variables are consistent with the semantics of a particular code fragment. CAS uses eSSA representation to form a graph of variable relationships with sufficient control flow information to ensure that its conclusions are semantically meaningful. It then finds paths in the graph which traverse a proposed constraint and represent an inconsistent constraint system via a novel depth-first search algorithm that respects program control flow and semantics. This information can be used to determine that array bounds checking is not necessary along particular control flow paths. We present experimental results that demonstrate that CAS is able to find more redundant checks than previous work, and improves the runtime of JAVA benchmarks up to 10%.

Our results show how CAS is useful when used at compile time to identify fully or partially redundant bounds checks, and having its conclusions encoded in the form of annotations which are efficiently verified at run-time. In future work, we intend to examine CAS's usefulness in bound check elimination as a runtime component of a JVM. This will make optimization available to a much larger set of JAVA classfiles, rather than just those that were previously analyzed by CAS and annotated.

Also, proposed constraints currently assume a bounds check is unsafe and CAS determines those control flow paths on which the check is unnecessary. We intend in future work to modify CAS to reverse that paradigm: where proposed constraints represent safe conditions and CAS identifies those situations where a bounds check is required.

Würthinger et al. [14] have demonstrated the considerable additional gains can be made by speculatively optimizing for the case where all bound accesses are safe and adding an additional check for sufficient conditions. We intend to incorporate this approach by enhancing CAS's analysis to automatically produce sufficient safety conditions that can be inserted and tested before entering loops to ensure access safety.

References

[1] Wolfram Amme, Jeffery von Ronne, and Michael Franz. Ssa-based mobile code: Implementation and empirical evaluation. *ACM Trans. Archit. Code Optim.*, 4(2):Article 13, 2007.

[2] William Blume and Rudolf Eigenmann. Demand-driven, symbolic range propagation. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, London, UK, 1996. Springer-Verlag.

[3] 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.

[4] 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.

[5] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeño dynamic optimizing compiler for java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM.

[6] 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.

[7] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1978. ACM Press.

[8] 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.

[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] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM Press.

[12] 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.

[13] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley and Sons, 1986.

[14] 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.

[15] 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.