A Comprehensive Approach to Array Bounds Check Elimination for Java

Feng Qian, Laurie Hendren, Clark Verbrugge

Sable Research Group
School of Computer Science, McGill University
Montreal, CANADA

http://www.sable.mcgill.ca
Introduction

• Array bounds checks are required by the semantics of the Java programming language.

• Runtime checks can slow down the program execution significantly.

• This paper presents an algorithm for proving which array references are safe (do not need bounds checks).
The Problem

- `ArrayIndexOutOfBoundsException`.
- Precise exception requirements limit code motion and loop transformations.
- A multi-dimensional array is an array of arrays (can be ragged).
- The lower bound check comes free with the upper bound check on most modern architectures.
Overview


- Two extensions:
  1. class level analysis: array field analysis.
  2. inter-procedural analysis: rectangular array analysis.
Difference Constraints

• Bounds checks for \( a[i] : 0 \leq i \leq a.length - 1 \)?

• From a program text, a static analysis is able to collect a set of inequalities.

• The system of difference constraints is represented as a constraint graph.

• The shortest path of the graph answers above query.
**Variable Constraint Graphs**

- **Nodes:** *int* type locals, *array* type locals, the constant 0 node.

- **Edges:** $e(u, v) = w$ represents the difference constraint $v - u \leq w$.

- **Ordering of edge weights:** $\bot \sqsubseteq \text{minint} \sqsubseteq \ldots \sqsubseteq c \sqsubseteq c + 1 \sqsubseteq c + 2 \sqsubseteq \ldots \sqsubseteq \text{maxint} \sqsubseteq T$
An Example of VCG

\[ s_0 : i = j + 2; \]
\[ s_1 : a[i] = \cdots; \]
\[ s_2 : i = \text{foo}(\cdots); \]
\[ s_3 : a[j] = \cdots; \]

(a) a basic block

(b) vcg after \( s_0 \)

(c) vcg after \( s_1 \)

(d) vcg after \( s_2 \)
VCG Properties

- Weighted directed edges.
- Inequalities are transitive.
- The shortest path gives the tightest constraint.
- But, it cannot represent multiplication and division.
Array-Related Liveness Analysis

- Specialized live variable analysis focusing on only variables related to array references.
- Computes accurate node sets for VCGs at interesting program points.

<table>
<thead>
<tr>
<th>Expr</th>
<th>Cond</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[i]</td>
<td>a, i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = new T[i]</td>
<td>a</td>
<td>i</td>
<td>a</td>
</tr>
<tr>
<td>i = j + c</td>
<td>i</td>
<td>j</td>
<td>i</td>
</tr>
<tr>
<td>...</td>
<td>i</td>
<td>j</td>
<td>i</td>
</tr>
</tbody>
</table>
**Variable Constraint Analysis (1)**

- Forward, flow-sensitive, optimistic analysis.
- Break basic blocks at statements with array references.
- The analysis approximates VCG edge weights.
- At a join point, merging two graphs by choosing the MAX weight for each edge, then widening the edge weights, if necessary.
Variable Constraint Analysis (2)

• The flow function for a basic block:

\textbf{INPUT} : the VCG at the entry of block.

\textbf{OUTPUT} : the VCG after taking the effect of statements of the block.

• Edge weights at the entry point are initialized to \top, others are set to \bot.

• Take conditions of branches into account (for \textit{if} (\textit{i} < \textit{n}): \textit{i} – \textit{n} \leq –1 on true path, \textit{n} – \textit{i} \leq 0 on false path.)
Tune The Analysis

- Limit the size of constraint graphs by pre-computing the set of array-related live variables.
- Widen edge weights at the confluence points.
- Enforce a good ordering in the work list:
  - the loop body is visited before the loop exit;
  - inner loops reach a fixed point before outer loops.
int[] init_array(int n) {
    int[] a = new int[n];
    for (int i=0; i<n; i++) {
        a[i] = i;
    }
    return a;
}

Step 1: order CFG nodes

Step 2: array-related live variable analysis
After

Before

After

START

{n}

\[a = \text{new int}[n];\]
\[i = 0;\]

\{a, i, n\}

if \((i < n)\)

\{a, i, n\}

Yes

No

\{a, i, n\}

\[a[i] = i;\]
\[i = i + 1;\]

return \(a;\)

END

Step 3: Variable Constraint Analysis

\begin{align*}
\text{START} & \quad \{n\} \\
\text{1} & \quad a = \text{new int}[n]; \quad i = 0; \\
\text{2} & \quad \text{if } (i < n) \quad \{a, i, n\} \\
\text{3} & \quad \text{No} \quad \{a, i, n\} \\
\text{4} & \quad \text{return } a; \\
\text{END} & \quad \end{align*}
Step 3: Variable Constraint Analysis

1. \( a = \text{new int}[n]; \quad i = 0; \)
   - \( \{n\} \)

2. if (i<n)
   - Yes: \( \{a, i, n\} \)
   - No: \( \{a, i, n\} \)

3. \( a[i] = i; \quad i = i+1; \)

4. return a;

Before

1. merge

2. widen with

3. 

1. \( a \rightarrow 0 \rightarrow n \)
   - \( i \rightarrow 0 \rightarrow 0 \)

2. \( a \rightarrow 0 \rightarrow n \)
   - \( i \rightarrow 1 \rightarrow 0 \)

3. \( a \rightarrow 0 \rightarrow n \)
   - \( i \rightarrow 1 \rightarrow 0 \)
   - \( T \rightarrow 0 \rightarrow 0 \)
Step 3: Variable Constraint Analysis

START
{n}

a = new int[n];
i = 0;
{a,i,n}

if (i<n)
{a,i,n}

No

Yes

a[i] = i;
i = i+1;

return a;

END

Before 2

Before 3

\[ i - a \leq -1 \]

\[ 0 - i \leq 0 \]
Extension 1: Array Field Analysis

- Example: final int[] m={2,4,7,9...};
- Observation: a final or private field can only be assigned a value in the declaring class.
- Find all possible array lengths using DU/UD chains.
- Identify those array fields with constant lengths.
- The constant array length is used by the intra-procedural analysis.
Extension 2: Rectangular Array Analysis

- Array Type Graph:
  
  \[ a = \text{multianewarray } T[i][j] : \text{TRUE} \rightarrow a; \]
  
  \[ a[i] = c : \text{FALSE} \rightarrow a; \]
  
  \[ a = b : a \leftarrow b. \]

- A variable 'a' has a rectangular shape if \( \text{TRUE} \sim a \) and \( \text{FALSE} \not\sim a. \)

- A special node in VCG for the length of subarrays: \( a[i]. \)
Experimental Approach

- Used as ahead-of-time analyses in the Soot framework.
- Analysis results encoded in class files as attributes.
- Modified Kaffe JIT and IBM HPCJ take advantage of such attributes.
## Characteristics of The Algorithm

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

(a) Results of the intra-procedural analysis

(b) Improvements due to field and shape analysis
Speedups in KaffeVM and HPCJ

(a) Kaffe VM

(b) HPCJ (other optimizations on)
Related Work (1): ABCD (PLDI’00)

- ABCD (R. Bodik et. al.) is based on an extended SSA form, and uses one constraint graph for a method; VCA is based on CFG, and computes small program-point specific constraint graphs.

- ABCD proves safe bounds on demand; VCA analyzes all references at once.

- ABCD is capable of catching partial redundant bounds checks; VCA is not.

- VCA analyzes lower and upper bounds at the same time, and the results can be improved by two extended analyses.
Related Work (2): Detect Array Memory Leaks (CC’00)

- R. Shaham et. al. used very similar representations as VCG to detect live ranges of array reference.

- The algorithm works on supergraphs of particular library classes (Vector).

- VCA focuses on intra-procedural analysis and uses various techniques to reduce the cost of data-flow analysis.
Conclusions

- We presented a collection of analyses for eliminating array bounds checks in Java.
  - Variable Constraint Analysis
  - Array Field Analysis
  - Rectangular Array Analysis
- Techniques reduce the cost of the data-flow analysis.
- The algorithm is effective for proving safe array references.