ABCD: Eliminating Array Bounds Checks on Demand

Rastislav Bodík
University of Wisconsin
bodik@cs.wisc.edu

Rajiv Gupta
University of Arizona
gupta@cs.arizona.edu

Vivek Sarkar
IBM T.J. Watson Research Center
vsarkar@us.ibm.com

Abstract

To guarantee typesafe execution, Java and other strongly typed languages require bounds checking of array accesses. Because array-bounds checks may raise exceptions, they block code motion of instructions with side effects, thus preventing many useful code optimizations, such as partial redundancy elimination or instruction scheduling of memory operations. Furthermore, because it is not expressible at bytecode level, the elimination of bounds checks can only be performed at *run time*, after the bytecode program is loaded. Using existing powerful bounds-check optimizers at run time is not feasible, however, because they are too heavyweight for the dynamic compilation setting.

ABCD is a light-weight algorithm for elimination of Array Bounds Checks on Demand. Its design emphasizes simplicity and efficiency. In essence, ABCD works by adding a few edges to the SSA value graph and performing a simple traversal of the graph. Despite its simplicity, ABCD is surprisingly powerful. On our benchmarks, ABCD removes on average 45% of dynamic bound check instructions, sometimes achieving near-ideal optimization.

The efficiency of ABCD stems from two factors. First, ABCD works on a *sparse* representation. As a result, it requires on average fewer than 10 simple analysis steps per bounds check. Second, ABCD is *demand-driven*. It can be applied to a set of frequently executed (hot) bounds checks, which makes it suitable for the dynamic-compilation setting, in which compile-time cost is constrained but hot statements are known.

1 Introduction

The advent of safe mobile computing in general, and Java in particular, has brought about two significant changes for optimizing compilers. First, type safety requires expensive run-time semantic checks (*e.g.,* bounds checks, null checks, type checks, etc.). Second, because the mobile bytecode must be verifiably typesafe when it is loaded, the optimizer can remove semantic checks from bytecode only at run time. This paper addresses the problem of eliminating redundant bounds checks using lightweight techniques suitable for the time-constrained dynamic-compilation setting.

PLDI 2000, Vancouver, British Columbia, Canada.

Bounds checks cause programs to execute slower for two reasons. One is the cost of executing the bounds checks themselves since they can occur quite frequently and involve a memory load of the array length and two compare operations. Even more importantly, the presence of bounds checks greatly limits the application of code optimizations. Precise exception semantics (as in Java) requires that all code transformations preserve the program state at an exception point, and also the order in which exceptions occur. As a consequence, the application of traditional optimizations must be restricted to prevent side-effect-causing instructions from moving across any exception points. Since the exception points introduced by array bounds checks can be frequent, the scope over which optimizations are applicable can be severely restricted.

When designing optimizations for Java, we face two conflicting goals: the optimization algorithm must be *fast* yet *general enough* to globally identify bounds checks that are fully redundant, or even partially redundant. The following observations provide insight into meeting the challenge for keeping the cost of dynamic optimization low:

- We should employ demand-driven-analysis techniques since they are highly suitable for dynamic optimization. Demand-driven analysis can be used to focus attention on "hot" bounds checks; *i.e.,* the bounds checks with the highest execution frequencies. Furthermore, demand-driven analysis is well suited to dealing with incremental updates of data flow information after program transformations because it sidesteps the need for expensive initialization phases needed in exhaustive analyses.

- We should develop algorithms that reuse representations that are commonly used in compilers. Our approach in this work is to start with SSA form [CFR+91] (*i.e.,* we assume it to be already available) and develop an algorithm that works by simple traversals of a cheaply computable extension of the SSA graph.

Many of the existing algorithms for array-bounds-check elimination are heavyweight (*e.g.,* those based on theorem provers [SI77,Nec98,NL98,XMR00]) and are therefore not suitable for deployment in a dynamic-optimization environment. Some simpler algorithms (*e.g.,* those based upon value-range analysis [Har77, Pat95, RR99]) cannot eliminate partially redundant checks. Algorithms that can eliminate partial redundancy [MCM82, Gup90, Gup94, Asu92, KW95] operate upon dense program representations (*e.g.,* the control flow graph) and rely upon exhaustive iterative data flow analyzers. Thus, they, too, do not meet our dynamic-compile-time requirements.

In this paper, we introduce a new algorithm, called ABCD, for elimination of array bounds checks on demand. The ABCD algorithm makes the following contributions:

1. *Sparse Representation:* ABCD uses a novel *sparse* representation called the *inequality graph*, which is built from an extended SSA representation. This representation can be cheaply constructed from a given SSA representation of the program. We will see that the inequality graph representation is powerful enough to enable all array-bounds checks to be eliminated from the BubbleSort program in Figure 1. (To the best of our knowledge, no other existing Java compiler can fully eliminate all the bounds checks in this example.)

2. *Demand-Driven Analysis:* ABCD employs a demand-driven approach in which the algorithm proceeds by propagating *inequality assertions* that need to be verified to eliminate a bounds check. Therefore, the optimization can be performed *incrementally* by starting with the "hot" bounds checks.

3. *Generality:* The effectiveness of ABCD stems from its ability to remove both *fully* and *partially redundant* checks. Through simple traversals of the sparse inequality graph representation, our algorithm is able to locate insertion points for checks required for eliminating partial redundancy. To remain within the tight compile-time constraints of a dynamic compiler, ABCD exploits recent results in redundancy elimination, which show how simple profile-based algorithms can achieve nearly complete removal of redundancies [BGS98].

   For the sake of efficiency, the assertion checking (theorem proving) is restricted to the five classes of "constraint-generating" statements described in Section 2.

4. *Empirical evaluation:* Our experience shows that, despite its simplicity, ABCD is efficient, effective, and simple to implement. We have implemented ABCD in the Jalapeño optimizing compiler [BCF+99]. Our results show that ABCD removes on average 45% of all dynamic bounds checks, while performing fewer than 10 analysis steps per bound check.

The rest of the paper is organized as follows. Section 2 gives an overview of ABCD. Section 3 presents our sparse program representation. Section 4 describes the constraint system used for detecting redundant checks and Section 5 shows how to solve the constraint system to remove *fully* redundant bounds checks. Section 6 extends the removal to *partially* redundant bounds checks. Section 7 outlines possible extensions to the ABCD algorithm. Section 8 presents experimental evaluation of the ABCD algorithm. Finally, Section 9 compares ABCD with existing work.

2 Overview of ABCD

A bounds check "**check** $A[x]$" is redundant if $0 \le x < A.length$ whenever the check is executed. ABCD optimizes the lower-bound check ($0 \le x$) and the upper-bound check ($x < A.length$) as two independent problems.[1] In this paper, we restrict our attention to the optimization of upper-bound checks. The (dual) algorithm for lower-bound checks can be derived trivially.

A straightforward approach to detecting redundant checks is to **i**) construct a constraint system at each program point, perhaps by propagating the constraints using dataflow analysis, and then **ii**) apply a theorem prover at the point of the bounds check. Both of these are expensive steps and ABCD streamlines them in the following ways:

[1] We forgo the (rare) optimization opportunities created by the interplay of the two problems for the sake of simplicity.

```
limit = a.length;
st = -1;
while (st < limit) {
   st++;
   limit--;
   for (j = st; j < limit; j++) {
      if (a[j] = a[j+1]) ...
   }
   for (j = limit; --j >= st; ) {
      if (a[j] = a[j+1]) ...
   }
}
```

Figure 1: **The running example: Bidirectional Bubble Sort.** The figure shows a relevant fragment of a program from the Symantec benchmark suite. A bounds check is performed before each of the four array accesses. To simplify the presentation, the rest of the paper omits the second *for* loop. ABCD can eliminate all four bound checks in this example.

1. Instead of constraint propagation, ABCD builds a single, program-point-independent constraint system. This constraint system is sparse; in fact, it is nothing more than SSA form with a few extra edges.

2. Instead of relying on a theorem prover, ABCD performs a simple, demand-driven traversal of the sparse representation. Despite its limited power, our "traversal prover" was able to eliminate 45% of dynamic upper-bound checks in our experiments.

The remainder of this section outlines the main components of ABCD: the types of constraints, the sparse constraint system and its graph representation, the constraint solver, and finally the handling of partially redundant bounds checks.

The Constraints. ABCD is efficient because it operates on simple *difference constraints* [Pra77, CLR92] of the form $x - y \le c$, where x is a program variable, y is a program variable or a symbolic literal (e.g., the length of an array), and c is an integer constant. Restricting the constraint form is what enables ABCD's efficiency: constraints on *pairs* of variables can be represented in the sparse, global constraint system; and the *difference* relationships can be processed with the simple "traversal solver." Other work (*e.g.,* [Sho81]) has also found it beneficial to restrict attention to this class of inequalities.

Furthermore, ABCD gathers only constraints that can be obtained with a local examination of the program code, without any prior global analysis. The following five types of program statements have been found to generate constraints useful for bounds-check elimination:

$C1$	array-length literal	$x := A.length$
$C2$	constant assignment	$x := c$
$C3$	constant increment or decrement	$x := y + c$
$C4$	conditional branches	**if/while** $x \le y$
$C5$	array-bounds check	**check** $A[x]$

Constraints $C1$-$C3$ are generated by restricted forms of assignments. Constraint $C4$ exploits invariants generated, on each exit of a branch, by the branch's conditional expression. Finally, constraint $C5$ exploits a *successful* array-bounds check "**check** $A[x]$," which guarantees that $x < A.length$. Note that all five of these constraint types can be expressed as difference constraints. If a variable is defined in an assignment that generates more complex constraints (e.g., those outside $C1$-$C3$), then ABCD considers the variable unconstrained (unless it also appears in statements $C4$-$C5$). In general, disregarding existence of some constraints is safe, as it will, at worst, hide from us some redundant checks.

Sparse Global Representation of Constraints. Constraints generated by program statements $C1$–$C5$ are program-point specific. Each of them is guaranteed to hold only in its *live range*, which is, informally, the range of CFG nodes between the generating statement and its dominance frontier or a killing definition, whichever is closer. To encode the constraint live ranges compactly (*i.e.,* globally rather than per node), ABCD splits live ranges of program variables via SSA-style renaming. In effect, the renaming converts a flow-sensitive constraint system into an equivalent flow-insensitive system, as was done for pointer analysis in [HH98].

ABCD splits live ranges of variables such that **i**) each resulting live range refers to a *unique* variable name, and **ii**) for any variable v_i, the live range of v_i is no larger than the live range of any constraint involving v_i. When expressed using the unique variable names, the constraint system becomes program-point-independent (global). Moreover, inheriting the SSA properties, the constraint system is sparse, in that it directly connects statements that generate related assertions, thus speeding up the constraint-solving process.

As in SSA, we split live ranges by inserting "dummy" assignments. To split live ranges of constraints $C1$–$C3$, the standard SSA ϕ-assignments are sufficient. However, constraints $C4$ and $C5$ require additional splitting, which we perform by inserting π-assignments at conditionals and bound checks. We call the resulting representation the *extended SSA* (*e*-SSA) form. The *e*-SSA form for the running example is shown in Figure 3.

Solving the Constraint System. To simplify the presentation, we view the sparse constraint system not as a system of inequalities, but instead as a graph, called the *inequality graph*. On the graph, substitutions of constraints can be viewed as a graph traversal, which we exploit for formulating a simple "traversal prover" for our constraint system.

The inequality graph is an extension of the SSA *value graph* [AWZ88]. There is one node for each *e*-SSA-variable, constant, or array-length literal (e.g., the value $A.length$); edge weights constrain the differences between pairs of nodes. Given an inequality graph, an (upper) bounds check "**check** $A[x]$" is redundant if the shortest path from $A.length$ to x has negative length, which corresponds to the bounds check condition, $x - A.length < 0$, being always true. The inequality graph for the running example is shown in Figure 4.

The "traversal solver" for our constraint system is thus equivalent to a shortest-path computation; a demand-driven version of the solver amounts to computing the shortest path between a pair of vertices ($A.length$ and x). The situation is complicated, however, by the fact that the inequality graph is a *hypergraph* [Ber73, GLPN93, RR96], which operates on a generalized notion of the shortest path (see Section 4).

Handling Control Flow. The reason why ABCD's constraint system induces a non-standard notion of the shortest path is that constraints must be treated in two different ways: Along a given control flow path, each variable is bounded by the *strongest* constraint generated on that path. In contrast, across a set of control flow paths, a variable is bounded by the *weakest* constraint produced by any of the paths. This "max-min" semantics of the inequality graph is encoded via two kinds of nodes. The *max* nodes are the ϕ-nodes, which correspond to control flow merges. The remaining nodes act as the *min* nodes.

```
1. Build e-SSA form:
       insert π-nodes (see Section 3)
       compute SSA form (e.g., using [CFR+91])
2. Build the Inequality Graph G_I:
       (see Table 1)
3. Remove redundant checks:
       for each check n of the form "check A[x]" do
           -- see Figure 5
           if demandProve(G_I, ⟨x − A.length ≤ −1⟩) then
               remove n from the program
       end for
```

Figure 2: **The ABCD algorithm.**

The inequality graph also enables elimination of partially redundant bound checks. Partially redundant checks are redundant along some, but not necessarily all, control flow paths [MR79]. A typical example is a loop-invariant check: its outcome is the same in each loop iteration and therefore it can be optimized by being executed once, before the loop is entered. While fully redundant bounds checks can be deleted, partially redundant checks require insertion of compensation checks that ensure that the check becomes redundant along all control flow paths.

We identify the insertion points using the ϕ-nodes in the inequality graph. As was shown in [CCK+97] in the context of *expression* elimination, insertion points correspond to a certain subset of in-edges of the ϕ-nodes. We extend this approach to the inequality graph. Because the insertion performed by ABCD may be speculative, we rely on profiling data to determine profitability of the optimization. Our approach relies upon recent results that demonstrate that speculative insertion is nearly as effective as non-speculative techniques that perform complete redundancy removal using code duplication [BGS98].

The ABCD Algorithm. The entire ABCD algorithm for fully redundant checks is outlined in Figure 2. The first step transforms the program into *e*-SSA form; the second steps connects *e*-SSA variables with constraints; the third step solves the constraint system and removes checks that have been proven to be redundant.

3 The Extended SSA Form

ABCD encodes the scope of constraints by transforming the program into the *extended SSA* (*e*-SSA) form, described in this section. The salient property of *e*-SSA is that constraints generated in an *e*-SSA program are valid *wherever* their variables are *live*. This implicit encoding obviates the need to explicitly qualify constraints with their CFG scope, which results in a compact constraint system. Another important property is that constraints expressed on the *e*-SSA program define a *sparse* constraint system that can be solved efficiently.

To understand the relationship between live ranges of variables and scopes of constraints, consider a constraint $x - y \leq c$ generated in a CFG node n (for example by statement $x := y + c$). A safe rule for delineating the scope of this constraint is to restrict it to nodes at which x and y have the same values as they had at node n, which is typically approximated by requiring that the values of x

and y originate from the same definition sites as they did at node n. Given this rule, the scope can be represented by enumerating nodes where this "same-definitions" condition holds. ABCD uses a more efficient approach: it splits and renames live ranges of variables so that variables that appear in a constraint are used (*i.e.,* are live) only within the constraint's scope.[2] In effect, the renaming converts a flow-sensitive constraint system into an equivalent flow-insensitive system, without a loss of precision.

ABCD splits variable live ranges by means of assignments, either existing or new ones. As in the SSA form, each assignment writes into a unique variable name, which uniquely names live ranges. Assignments are needed at two points:

Case 1) When a constraint is created: Live ranges of constraints $C1$–$C3$ already start at assignments, whose target variables can be directly renamed. Constraints $C4$ start at exits of conditionals, and constraints $C5$ start at bounds checks. To capture these two constraint types, ABCD inserts π-assignments, as will be discussed below.

Case 2) When a constraint is killed: The scope of a constraint $u - v \leq c$ generated at node n is terminated (**i**) at a node where u or v is redefined, (**ii**) at a control flow join that can be reached by a path from n along which u or v are redefined, or (**iii**) at a control flow join that is not dominated by n.

The standard SSA form sufficiently splits variable live ranges to encode constraints $C1$–$C3$. The SSA renaming is performed as follows [AWZ88]: New definitions, called ϕ-assignments, are placed (recursively) at control flow join points that are reachable by different definitions of a variable v. Renaming the targets of original assignments to v enforces cases 1 and 2-i above; renaming the targets of ϕ-assignments enforces cases 2-ii and 2-iii. After the renaming, each use of a variable is dominated by its single reaching definition, which guarantees the desired property that v has the same value in its entire live range.

To express constraints $C4$ and $C5$, case 1 requires additional renaming. The renaming is performed by introducing π-assignments on the exits of conditionals and bounds checks. Consider the conditional statement shown below on the left:

if $(i_1 \leq 10)$ **then**		**if** $(i_1 \leq 10)$ **then**
		$i_2 := \pi(i_1)$
		// $i_2 \leq 10$ here
else	$\Longrightarrow$	**else**
		$i_3 := \pi(i_1)$
		// $i_3 > 10$ here
end if		**end if**
		$i_4 := \phi(i_2, i_3)$

The π-assignments are inserted, for each variable appearing in the conditional expression, into the CFG out-edges of the conditional branch.[3] Thanks to the π-assignemnts, each outcome of the conditional is associated with a distinct variable name, which serves as a "hook" for attaching the constraints generated by the branch. Although π-assignments are inserted into CFG edges, in the textual representation of the program used throughout this paper, π-assignments are placed at the beginning of the basic blocks targeted by the branch.

Similarly, to generate a new name for constraints $C5$, a π-assignment is inserted after each bounds check. A bounds check can be viewed as a special *if* statement that will transfer program's control into the exception handler if the check fails; if the check does not fail, a useful constraint is generated:

check $a[i_1]$		**check** $a[i_1]$
	$\Longrightarrow$	$i_2 := \pi(i_1)$;
		// $i_2 < a.length$ here

The constraint $C5$ must be expressed on the new name i_2, rather on i_1, otherwise it could erroneously lead to elimination of some bound checks, including the generating **check** itself.

Example 1 (*e*-SSA Form) Figure 3 depicts the running example before and after the conversion to *e*-SSA form. For the sake of brevity, the second *for* loop from Figure 1 is omitted. To reduce the number of ϕ-assignments induced by π-assignments, no ϕ-assignment for `limit` is inserted into the *for* loop. This is safe because there are no uses of $\texttt{limit}_4$ in the loop. □

4 The Constraint System and the Inequality Graph

Once the program is converted into *e*-SSA form, the constraints generated throughout the program can be "connected" into a single, flow-insensitive constraint system. Thanks to the simplicity of constraints considered by ABCD (difference constraints of two variables), this constraint system can be represented as a weighted directed graph and can be solved with relatively efficient graph traversal algorithms. This section presents the graph representation of the constraint system and its properties. The following section presents an efficient solver for the constraint system.

Representing simple constraint systems with graphs is a standard technique; a common example is the *constraint graph* [Pra77, CLR92]. Our *Inequality Graph* (G_I) generalizes the constraint graph in that it allows representing program's control flow, as follows. First, G_I allows us to detect whether a given bounds check is redundant along *all* control flow paths. Second, G_I maintains enough program structure information to perform code motion of partially redundant bounds checks (see Section 6). The solution of the constraint system represented by G_I is computed similarly to how it is computed on the constraint graph — using a notion of the shortest path (see Section 5).

Informally, vertices of the inequality graph represent program variables (in *e*-SSA form), constants, and literals. An edge connects two vertices if the program generates a constraint on their difference (variables for which the program does not generate a constraint will create disconnected vertices in G_I).

Definition 1 (The inequality graph G_I) Given a program P in *e*-SSA form, the *Inequality Graph* of P is a weighted, directed graph $G_I = (V, E, d)$ with a distinguished set of vertices $V_\phi \subseteq V$:

- $V = \{v_i\} \cup \{A_j.length\} \cup \{c_k\}$, where v_i is a program variable, $A_j.length$ is a program literal denoting the length of array A_i, and c_k is an integer constant appearing in the program P.
- E contains a directed edge $u \to v$ iff P generates a constraint $v - u \leq c$ according to any rule in Table 1. The weight $d(u \to v)$ of the edge is $c \in \mathbb{N}$.
- V_ϕ is a distinguished subset of vertices, $V_\phi \subseteq V$, such that $v_i \in V_\phi$ iff variable v_i is defined in program P using an *e*-SSA ϕ-assignment. □

[2] To reduce the amount of program transformation, we actually allow the variables appearing in the constraint to be live outside the scope, as long as they are simultaneously live only in the scope.

[3] Note that π-assignments are analogous to the *switch* operators in the dependence flow graph [JP93].

Original program:

```
limit := A.length
st := −1
while:          while (st < limit)


if (st < limit) {


   st := st + 1
   limit := limit − 1
   j := st
   for:         for (j := st; j < limit; j++)

   if (j < limit) {


      check A[j]         A[j]

      t := j + 1
      check A[t]         A[j+1]

      j := j + 1         j++
      goto for
   }                     end for
   goto while
}                        end while
```

***e*-SSA form:**

```
  limit_0 := A.length
  st_0 := −1
 while:
  limit_1 := φ(limit_0,limit_3)
  st_1 := φ(st_0,st_3)
  if (st_1 < limit_1) {
     st_2 := π(st_1)
     limit_2 := π(limit_1)
     st_3 := st_2 + 1
     limit_3 := limit_2 − 1
     j_0 := st_3
   for:
     j_1 := φ(j_0,j_4)
     if (j_1 < limit_3) {
        j_2 := π(j_1)
        limit_4 := π(limit_3)
        check A[j_2]
        j_3 := π(j_2)
        t_0 := j_3 + 1
        check A[t_0]
        t_1 := π(t_0)
        j_4 := j_3 + 1
        goto for
     }
     goto while
  }
```

Figure 3: **The running example before and after the conversion into *e*-SSA form.**

Recall that Table 1 defines the edges of G_I for the elimination of upper-bound checks; the elimination of lower-bound checks is based on analogous, but separate, inequality graph.

Example 2 (The Inequality Graph G_I) Figure 4 shows the inequality graph G_I for the running example. The vertices are the program entities from Figure 3: (*e*-SSA) variables, the literal *A.length*, and the constant -1 . The set V_ϕ contains the three ϕ-assignments. Note that, thanks to *e*-SSA form, a vertex represents the definition as well as all uses of a variable. *e*-SSA form also guarantees that G_I is not a multigraph. Finally, note that the names in V denote, interchangeably, both the program entities and the vertices of the inequality graph. □

It is worth mentioning why we decided to represent constraints generated by program assignments with inequalities, rather than with equalities, which appear to be the more intuitive choice, and which would also allow using the same inequality graph for both upper- and lower-bounds check elimination. The first motivation is to represent all constraints uniformly, exclusively with inequalities. The more important motivation is to formulate a *consistent* constraint system in the presence of the constraints $C4$. Consider the $C4$ rules in Table 1. If the assignments $v_j := \pi(v_i)$ and $v_k := \pi(v_i)$ were translated into constraints

$$v_j = v_i \quad \text{and} \quad v_k = v_i,$$

rather than into $v_j \leq v_i$ and $v_k \leq v_i$, as it is done by ABCD, the constraint system would imply

$$v_j = v_k.$$

Similarly, the system would imply $w_s = w_t$ for the other π-assignments in $C4$. As a result, the constraints generated on the exit of the conditional in rule $C4$, namely

$$v_j \leq w_s \quad \text{and} \quad w_t \leq v_k - 1,$$

would be inconsistent, as no value of the program variables v and w could satisfy the constraint on v_k:

$$v_k = v_j \leq w_s = w_t \leq v_k - 1.$$

To avoid the inconsistency, ABCD translates the assignments $v_j := \pi(v_i)$ and $v_k := \pi(v_i)$ into constraints

$$v_j \leq v_i \quad \text{and} \quad v_k \leq v_i,$$

which makes variables v_j, v_k mutually unconstrained, reflecting the fact that v_j and v_k are never live simultaneously at any node in the program and thus their values should not be compared.

We are now ready to define the system of difference constraints.

Definition 2 (The constraint system) The constraint system of an inequality graph $G_I = (V, E, d)$ with a distinguished set of vertices V_ϕ is a set of inequalities

$$v \leq \begin{cases} \max\limits_{u \to v}\{u + d(u \to v)\} & \text{if } v \in V_\phi, \\ \min\limits_{u \to v}\{u + d(u \to v)\} & \text{if } v \in V - V_\phi. \end{cases} \tag{1}$$

constraint type	generating statements	generated constraint	edge / edge weight	
$C1$	$v_i := A_j.length$	$v_i \leq A_j.length$	$A_j.length \rightarrow v_i$	0
$C2$	$v_i := c$	$v_i \leq c$	$c \rightarrow v_i$	0
$C3$	$v_i := v_j + c$	$v_i \leq v_j + c$	$v_j \rightarrow v_i$	c
$C4$	**if** $v_i \leq w_r$ **then**	$v_j \leq v_i$	$v_i \rightarrow v_j$	0
	$v_j := \pi(v_i)$	$w_s \leq w_r$	$w_r \rightarrow w_s$	0
	$w_s := \pi(w_r)$	$v_j \leq w_s$	$w_s \rightarrow v_j$	0
	else	$v_k \leq v_i$	$v_i \rightarrow v_k$	0
	$v_k := \pi(v_i)$	$w_t \leq w_r$	$w_r \rightarrow w_t$	0
	$w_t := \pi(w_r)$	$w_t \leq v_k - 1$	$v_k \rightarrow w_t$	-1
$C5$	**check** $A_k[v_i]$			
	$v_j := \pi(v_i)$	$v_j \leq A_k.length - 1$	$A_k.length \rightarrow v_j$	-1
control flow	$v_i := \phi(v_j, v_k)$	$v_i \leq \max\{v_j, v_k\}$	$v_j \rightarrow v_i$	0
		$(v_i \in V_\phi)$	$v_k \rightarrow v_i$	0

Table 1: **The edges of the inequality graph for elimination of upper-bounds checks.**

for each $v \in V$. A feasible solution for the constraint system is an assignment of integer values to variables in V that satisfies all constraints. □

Handling Control Flow. ABCD extends the standard difference-constraint system [Pra77] in order to express the control flow of the program. The standard system consists of a set of equations

$$v - u \leq d(u \rightarrow v) \quad \forall v, u \in V$$

which is equivalent to the following set of equations

$$v \leq \min_{u \rightarrow v}\{u + d(u \rightarrow v)\} \quad \forall v \in V. \tag{2}$$

To see how ABCD extends the standard difference-constraint system, compare equations 1 and 2: a set V_ϕ of distinguished vertices is constrained not by the *strongest* constraint that holds on its in-edges, but instead by the *weakest* one. To reflect the semantics of vertices in their mnemonics, we refer to vertices in V_ϕ as *max* vertices and the vertices in $V - V_\phi$ as *min* vertices.

Because V_ϕ corresponds to ϕ-assignments, Eq. 1 ensures that a bounds check is redundant along all incoming control flow paths. As an example, consider variable $st_1 \in V_\phi$ defined in Figure 4 by $st_1 := \phi(st_0, st_3)$. The following constraints hold for arguments of the ϕ-assignment (the first line follows from the fact that array length is non-negative, which could be represented as an edge in G_I):

$$st_0 \leq -1 \leq A.length - 1$$
$$st_3 \leq st_2 + 1 \leq limit_2 \leq limit_1 \leq limit_0 \leq A.length$$

i.e., the control flow predecessor corresponding to st_3 constrains the value of st_1 less than the predecessor corresponding to st_0. The constraint on st_1 that holds along *all* incoming control flow paths is $st_1 \leq A.length$, the weaker of the two constraints, as is (correctly) computed by Eq. 1: $st_1 \leq \max\{st_0, st_3\}$.

Consistency. Because G_I may contain negative cycles, its constraint system may seem inconsistent, due to implying $v \leq v + c,\ c < 0$. Informally, the consistency of out constraint system is guaranteed by the presence of at least one *max* vertex in each cycle, which breaks the cycle if it is negative. Consider the negative cycle $limit_1, limit_2, limit_3$ in Figure 4. Because each negative cycle strengthens constraints, the weakest constraint at the *max* vertex $limit_1$ must come from outside the cycle. By propagating the constraint on $limit_0$ from outside the cycle, the *max* node effectively breaks the cycle.

More formally, let C be a negative cycle in G_I. Because each cycle in G_I is created as a result of cyclic control flow, C must contain at least one ϕ-assignment whose one or more arguments are defined outside the cycle. Let v_1, defined by $v_1 := \phi(v_2, v_3)$, be such a vertex. From the constraint system we have $v_1 \leq \max\{v_2, v_3\}$. Assume that v_2 is the argument defined outside the negative cycle C. Hence, v_3 "closes" the negative cycle: $v_3 \leq v_1 + c,\ c < 0$. Therefore, $v_1 \leq \max\{v_2, v_1 + c\} = v_2$, which yields an equivalent constraint system without the negative cycle C.

Redundancy of a Bounds Check. We now relate the solution of the constraint system to the redundancy of a bounds check.

Definition 3 (Fully redundant check) An upper-bound check "**check** $A_i[v_j]$" is fully redundant if $v_j \leq A_i.length - 1$ whenever the check is executed. □

In terms of the constraint system, a bounds check is redundant if the check is true under each feasible solution of the constraint system. In other words, a bounds check is redundant if it is implied by the constraint system.

Theorem 1 Let b be an upper-bound check "**check** $A_i[v_j]$" in program P, and let G_I be the inequality graph of P. The check b is redundant if for any feasible solution D to G_I, the $D(v_j) \leq D(A_i.length) - 1$. □

Proving redundancy of the bounds check "**check** $A_i[v_j]$" thus entails computing the greatest value of $\Delta(A_i.length, v_j) = D(v_j) - D(A_i.length)$, for all feasible solutions D of G_I. The bounds check "**check** $A_i[v_j]$" is redundant if $\Delta(A_i.length, v_j) \leq -1$. We call the value $\Delta(u, v)$ the *distance* between vertices u and v in G_I. Alternatively, the distance $\Delta(u, v)$ can be defined as the smallest value d such that adding constraint $v \leq u + d$ into the constraint system will not change the set of feasible solutions of the system.

The problem of computing the distance in G_I is a generalization of the shortest-path problem in a weighted directed graph. Before we outline the generalization, we note that, for upper-bound check elimination, the distance in G_I corresponds to the longest path, *i.e.,* to the shortest path in the *reverse* problem, in which $+\infty < -\infty$ and hence the *max* operator selects the shorter path.

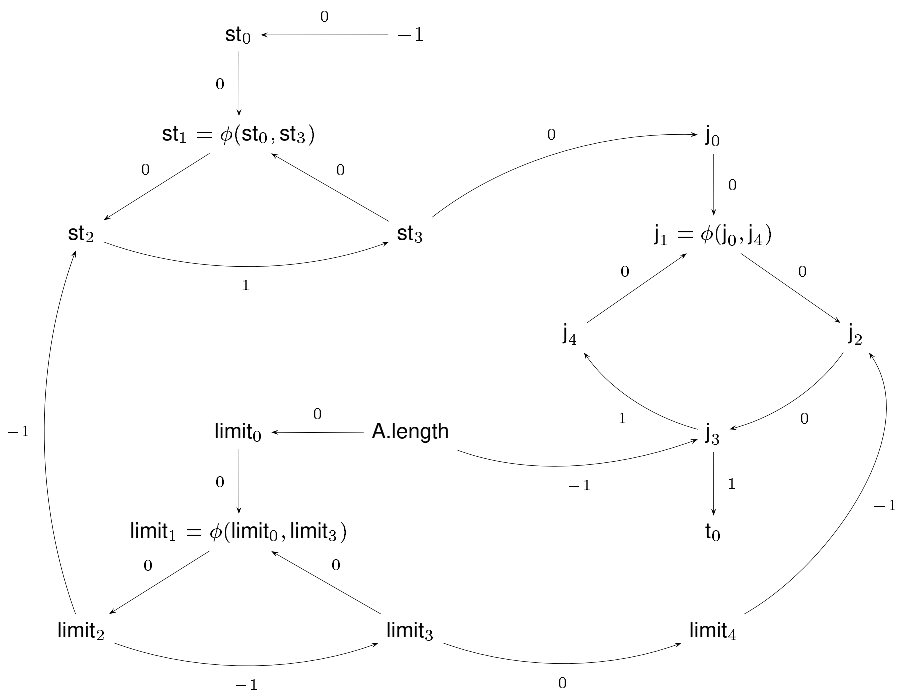

Figure 4: **The inequality graph G_I of the running example.** The meaning of an edge $u \xrightarrow{c} v$ is $v \leq u + c$. Each (upper-bound) check is represented in G_I with two vertices: the *array-length* vertex and the *array-index* vertex. For example, the bounds check "**check** $A[j_2]$," is represeneted with vertices $A.length$ and j_2. A bounds check is redundant if the distance between the two vertices is less than 0. To compute the distance from a node u, we "propagate" from u the sum of edge weights in the following way: at each ϕ-node, the distance equals the greatest incoming distance; at each remaining node, the distance equals the smallest incoming distance. The distance between $A.length$ and j_2 is -2. The corresponding "shortest" path for this distance is $\langle A.length, limit_0, limit_1, limit_2, limit_3, limit_4, j_2\rangle$.

For the rest of the paper, the terms "shorter/longer" refer to the reverse problem (in which 3 is shorter than 2).

An intuitive way to describe how distance generalizes the shortest path is to give an instance of G_I in which the distance and the shortest path are defined identically. This is the case when only vertices from V_ϕ have multiple predecessors. Under such a restriction, each path in G_I corresponds to a control flow path, and the distance in G_I corresponds to the shortest path in G_I. The shortest path thus finds a constraint that is valid along all control flow paths. In contrast, when vertices in $V - V_\phi$ are allowed to have multiple predecessors, G_I may contain paths that corresponds to *different* constraints that hold along the *same* control flow path. Therefore, when computing the distance along a given control flow path, we pick the longest of these different paths in G_I.

An elegant formalism for dealing with two kinds of paths is the *hypergraph* [Ber73, GLPN93, RR96]. A directed hypergraph consists of a set of nodes and a set of hyperarcs, where each hyperarc connects a set of nodes to a single target node. The concept of *hyperpaths* is defined recursively. There exists an empty hyperpath from from a set S of nodes to a node t if $t \in S$. A non-empty hyperpath from a set S of nodes to a node t consists of an hyperarc from a set S' to t and a hyperpath from S to s for every node s in S'. A hyperpath from S to t is thus a set of *component* paths, which are traditional paths (*i.e.,* sequences of vertices).

To turn the inequality graph into a hypergraph, we group all in-edges of a vertex $v \in V - V_\phi$ into a single hyperarc; every in-edge of a vertex in V_ϕ represent a separate hyperarc. The distance between u and v is defined as follows: The length of a hyperpath P from u to v equals the length of the longest of the component paths of P. The distance from u to v equals the shortest of all hyperpaths from u to v.

Example 3 To determine whether the bounds check "**check** $A[j_2]$" is redundant using the inequality graph in Figure 4, we compute the distance between the vertex $A.length$ and the vertex j_2. If the distance is less than 0, then the array index j_2 is at most -1 greater than the array length, and hence is within its bounds.

The distance between $A.length$ and j_2 is -2. Hence the check "**check** $A[j_2]$" is redundant. The distance is equal to the longest component path of the shortest hyperpath between the two vertices, which is $\langle A.length, limit_0, limit_1, limit_2, limit_3, limit_4, j_2\rangle$. □

5 The Constraint Solver

This section presents the details of the solver that ABCD uses to identify *fully* redundant checks. The extensions for *partially* redundant checks are described in the next section.

The constraint system represented by the inequality graph can be solved in various ways. First, as it has already been mentioned, the inequality graph G_I can be viewed as a hypergraph [Ber73, GLPN93, RR96]; the redundancy of a check is then reduced to computing the shortest hyperpath between two vertices (the array-length vertex and the array-index vertex). Second, when the hypergraph is viewed as an *abstract grammar problem* [Ram96], the shortest hyperpath can be found using a fixed-point computation. Third, the shortest hyperpath can be computed efficiently using the dataflow analysis solver in [GW75], in $O(E^2)$ time (even if G_I is irreducible).

ABCD uses an algorithm with worse asymptotic time complexity but with very good practical running time, which is mainly due to the typically small size of the sparse inequality graph. In contrast to the above three *exhaustive* analysis approaches, our solver works on *demand*. An exhaustive algorithm analyzes all bounds checks in the program, which in the context of shortest paths means computing the single-source shortest-path problem for each array-length vertex. A demand-driven approach analyzes a single bounds check, which amounts to computing the shortest path between the array-length vertex and the array-index vertex.

Our solver is demand-driven in yet another sense: Because we expect that a dynamic optimizer will optimize only a small fraction of all bounds checks, our design favors performance of a single-check analysis over a batch analysis of all bounds checks. Therefore, our solver does not actually return the *length* of the shortest path, but only a *boolean* information whether the shortest path is below a limit sufficient to prove that a given bounds check is redundant. This less strict question sometimes allows the solver to examine fewer paths than would be necessary to compute the precise value of the shortest path.

To explain how the solver works, let us consider first G_I that is acyclic (but contains both *min* and *max* nodes). An efficient way of computing the shortest path between a source s (an array-length vertex) and a target t (an array-index vertex) is to perform a topological traversal from s to t, performing the min/max operations in the process. However, to determine the topological order, one must first traverse the entire graph, which is what demand analysis seeks to avoid in the first place. Therefore, instead of a topological traversal, the solver performs a "brute-force" depth-first exploration of the graph, in which a node may be visited multiple times. Each successive visit of a node corresponds to a stronger question about the distance of that node from the array-length vertex.

The algorithm, shown in Figure 5, performs a depth-first traversal of G_I starting at the array-index vertex b. Intuitively, the recursive exploration of the graph proceeds *forward* (against the direction of G_I edges) until either the source vertex a is reached, or until a cycle is detected. The goal is to determine, for each traversed path from b to a, whether it is longer than c. To this end, the forward pass propagates the value c and adjusts it as it crosses each edge, so that when the source vertex is reached, the traversed path is longer than c iff the propagated value is greater than zero. Depending on the outcome of this comparison, the recursion will return the value of either *True* (the path is shorter) or *False* (the path is longer or of the same length).

Let us now consider cyclic inequality graphs. As was mentioned in the previous section, although arbitrary inequality graphs may contain negative-weight cycles, these cycles have non-decreasing effect on the path distance, because each cycle is broken by a *max* node. Cycles with positive weight, however, may impact path distance, if they are not broken by a *min* vertex. Positive cycles in G_I correspond to program loops in which the program variable is incremented in the loop body. We call these G_I cycles *amplifying* cycles. In Figure 4, the cycles involving st_1 and the cycle involving j_1 are amplifying. In contrast, the cycle of $limit_1$ is not amplifying. Consider the cycle on variable st in the example in Figure 4. If the edge $limit_2 \rightarrow st_2$ was removed from Figure 4, the cycle on st would cause the distance from st_0 to j_0 to be of positive infinite weight.

Conceptually, our algorithm works by identifying and reducing amplifying cycles. After such a cycle is broken, two situations may occur. If the cycle was an articulation point between the source vertex and the bounds check vertex, the distance will be (correctly) computed to have infinite weight, which means that the check cannot be proven to be redundant. If, after breaking the cycle, another path leads to the source (via a *min* vertex), the value of distance may still be small enough to prove the check redundant. Consider j_2 in Figure 4. When its amplifying cycle is broken (*i.e.,* by removing edge $j_1 \rightarrow j_2$), an alternative path to $A.length$ remains, via $limit_4$, with weight of -2, which is sufficient to prove the bounds check redundant.

To detect positive-weight cycles, the forward exploration keeps in *active*$[v]$ the value of the propagated value c for each vertex v that is on the active depth-first traversal path. When a back-edge is traversed, the positive weight cycle is detected by comparing the current value of c and its value "one cycle ago." When a positive-weight cycle is detected, the value *False* is returned. When a "harmless" cycle of weight zero or less is detected, we consider the path reduced, returning the value *Reduced*. The cycle is reduced in the sense that it does not influence the distance from a to b.

In summary, when the forward exploration stops at a node, we know for each path originating at b whether its distance is smaller than or equal to c (result *True*), greater than c (result *False*), or is reduced (result *Reduced*). When the recursion is returning *backwards*, the algorithm merges these results according to the min-max semantics of G_I vertices, using the following lattice L:

$$\textit{True} > \textit{Reduced} > \textit{False},$$

where *True* is the top element and *False* is the bottom element. A *max* node $v \in V_\phi$ merges these values using a meet operator $\sqcap$, $x \sqcap y = x \Leftrightarrow x \leq y$. A *min* node $v \in V - V_\phi$ merges these values using a join operator $\sqcup$, $x \sqcup y = x \Leftrightarrow x \geq y$.

6 Removing Partially Redundant Checks

The previous section identified checks that *never* fail. For some checks, however, such a strong guarantee cannot be proven. A typical example is a loop-invariant bounds check—all we can prove is that it either fails in the first iteration of the loop or it never fails. In general, such checks are called *partially redundant:* they are guaranteed not to fail along some (but not all) control flow paths leading to them.

Some partially redundant checks can be eliminated with an optimization called Partial Redundancy Elimination (PRE), which generalizes common subexpression removal and loop invariant code motion [MR79]. PRE works by inserting compensating checks in such program points that make the partially redundant check fully redundant (*i.e.,* after the insertions, the check will have been performed along all control flow paths) and hence it can be removed. We present here PRE optimization of array-bounds checks, as a natural extension of the ABCD algorithm for full redundancies.

Before we proceed to describe the extensions to ABCD, let us introduce into the running example a partially redundant check. This task is easily accomplished by removing the assignment "$limit_0 := A.length$" from Figure 3. The effect on the inequality graph (Figure 4) is that the vertex $limit_0$ becomes disconnected from vertex $A.length$, which breaks the shortest path that was used to prove that "**check** $A[j_2]$" is fully redundant. The check is now only partially redundant; namely, it is loop-invariant.

6.1 The Analysis

In order to turn partially redundant bounds checks into fully redundant checks, the analysis must determine:

- *Where to insert the compensating checks.* Our goal is to find, for each partially redundant check b, a set of CFG edges into which checks must be inserted to make b fully redundant. For "**check** $A[j_2]$" in Figure 3, it is sufficient to insert a check

function *demandProve*(G_I, t) **return boolean**

- $G_I = (V, E, d)$ is the inequality graph with *max* vertices $V_\phi \subseteq V$.
- $t = \langle b - a \leq c \rangle$ is the check to be proven, where $b \in V$ is the check's index variable, and $a \in V$ is the array-length literal. For example, when analyzing "**check** $A[x]$," $t = \langle x - A.\textit{length} \leq -1 \rangle$. Vertices a and b are the source and the target of the shortest-path computation, respectively.
- C memoizes the result of proving $v - a \leq c$, where a the array-length literal. $C : V \times \mathbb{N} \rightarrow L$, *i.e.,* C maps $\langle v - a \leq c \rangle$ into $\{$*True*, *False*, *Reduced*$\}$.
- *active* detects cycles: if *active*$[v] \neq$ **null**, then *active*$[v]$ is the distance of v from b, where b is the check's index variable. *active* maintains the distance for each vertex v that is on the path on the current DFS stack.

begin

    C $\leftarrow$ *active* $\leftarrow \emptyset$
    if *prove*$(a, b, c) \in \{$*True*, *Reduced*$\}$ **then return true** ; **else return false**

    function *prove*(vertex a, vertex v, int c) **return** $\{$*True*, *Reduced*, *False*$\}$
        // same or stronger difference was already proven
        if $C[v - a \leq e] =$ *True* for some $e \leq c$ **then return** *True*
        // same or weaker difference was already disproved
        if $C[v - a \leq e] =$ *False* for some $e \geq c$ **then return** *False*
        // v *is on a cycle that was reduced for same or stronger difference*
        if $C[v - a \leq e] =$ *Reduced* for some $e \leq c$ **then return** *Reduced*
        // traversal reached the source vertex, success if $a - a \leq c$
        if $v = a$ **and** $c \geq 0$ **then return** *True*
        // if no constraint exist on the value of v*, we fail*
        if v has no predecessor in G_I **then return** *False*
        // a cycle was encountered
        if *active*$[v] \neq$ **null then**
            if $c >$ *active*$[v]$ **then return** *False*   *// an amplifying cycle*
            else return *Reduced*   *// a "harmless" cycle*
        end if
        active$[v] \leftarrow c$
        if $v \in V_\phi$ **then**
            for each edge $u \rightarrow v \in E$ **do** $C[v - a \leq c] \leftarrow C[v - a \leq c] \sqcap$ *prove*$(a, u, c - d(u \rightarrow v))$
        else
            for each edge $u \rightarrow v \in E$ **do** $C[v - a \leq c] \leftarrow C[v - a \leq c] \sqcup$ *prove*$(a, u, c - d(u \rightarrow v))$
        end if
        active$[v] \leftarrow$ **null**
        return $C[v - a \leq c]$
    end *prove*
end *demandProve*

Figure 5: **The algorithm for proving the redundancy of a bounds check.** The procedure *demandProve*(G_I, t) returns true if the check $t = \langle b - a \leq c \rangle$ is proven to be redundant on the inequality graph G_I, which is equivalent to showing that the distance between vertex a and vertex b in G_I is no greater than c.

into the edge that corresponds to the first argument of $limit_1$'s ϕ-node, which is the entry of the *while* loop.

The ABCD algorithm extended for PRE computes the set of insertion edges during the backtracking from the recursive exploration of the inequality graph. A check is inserted into a ϕ-node's in-edge exactly when some of ϕ-node's arguments were proven (*i.e.,* procedure *prove()* returned *True*) and some were not able to be proven (*i.e., prove()* returned *False*). The *False* arguments are then collected during the backtracking into the insertion set.

- *What compensating checks should be inserted.* The compensating check may be different than the check being optimized. Specifically, the checks may differ in their index expression: the compensation of a check "**check** $A[v_i]$" may require insertion of a check "**check** $A[v_j + c_j]$." In our example, the check "**check**$A[j_2]$" is compensated with a check "**check** $A[limit_0 + 2]$."

  Identifying the index expression in ABCD is trivial. Due to the reliance on simple difference constraints, the index expression is always of the form $v_i + d$ (we assume that the index expression of the optimized bounds check is of the form $v_j + 0$): The variable v_i corresponds to the ϕ-node argument into which the insertion is being performed. The constant d equals the distance from the insertion point to the array-index vertex; in our example, the distance between $limit_0$ and j_2 equals 2; therefore, the compensating check in our example is "**check** $A[limit_0 + 2]$." This distance can easily be computed from the value c propagated by the recursive procedure *prove()*.

- *Profitability of removing the partial redundancies.* To ensure that inserted checks do not increase the dynamic number of checks in the program, traditional PRE techniques insert a check only when its cost can be amortized by removing a check on each control flow path emanating from the insertion point. The "amortization" condition is computed as backward dataflow problem of *anticipability* [MR79].

  ABCD estimates profitability using run-time profiling. We do not require that the check be removed on each emanating path. Instead, we allow *control-speculative* insertion, in which we speculate that the program will follow a path on which the insertion will lead to a removal of a check. To determine whether speculative insertion is profitable, ABCD compares the cumulative execution frequency of the insertion points with the frequency of the partially redundant check. If the impairment due to insertions is lower than the benefit due to removal of the check, we carry out the transformation. Such a profile-based PRE has been shown to be nearly as powerful as a complete removal of redundancies based on program restructuring [BGS99].

The algorithm in Figure 5 can be easily extended to remove partial redundancies. The recursive function *prove* is extended to return not only the three lattice values but, when the value is *False*, also the list of insertion edges that will make the optimized check fully redundant. The meet and join operators now also manipulate the insertion sets. At a min vertex, ABCD selects the set that has the lower execution frequency. At a *max* vertex, the propagated sets are merged.

6.2 The Transformation

Although ABCD allows an elegant *analysis* of where compensating checks should be inserted, it does not address the problem of *transforming* the program. The broader problem is how to maintain exception semantics in the presence of code motion: When a *hoisted* (*i.e.,* inserted) check fails, the exception cannot be raised immediately; it must be delayed, and raised at the location of the *original* (*i.e.,* partially redundant) check. This broader problem is beyond the scope of the paper. In this subsection we describe the solution that we currently use in our implementation of ABCD. We also sketch the solution that we are investigating.

In the current solution, each bounds check is split into two instructions: the *compare* instruction, which sets a register if the index is out of bounds, and the *trap* instruction that raises the exception if the flag is set. As we discuss below, traps require a different transformation than compares do. Therefore, our current approach optimizes only the compare instructions. While leaving traps in the original locations prevents us from moving code freely around array accesses, which was the primary motivation for bounds-check optimization, note that only partially redundant traps are left in the code; all fully redundant traps—a majority of all traps—are removed.

Removing traps is similar to removing conditional branches. Traditionally, conditional branches are optimized with program restructuring (*i.e.,* code duplication) [MW95, BGS97]. We see restructuring as too expensive for a dynamic compiler, and hence our current work is exploring transformation techniques unique to the dynamic-optimization setting.

Our approach for run-time removal of traps is as follows. The optimized version of the loop executes without a trap. When a compare instruction fails, the code for the unoptimized version of the loop (with the trap) is generated and the execution is transferred to it. Now, it is possible that the hoisted check failed spuriously, *i.e.,* it was executed speculatively (see Section 6.1). If this is the case, the unoptimized loop finishes without encountering the trap and we proceed with the execution of the optimized version of the code.

7 Extensions

7.1 Global Value Numbering

The inequality graph G_I can represent not only *local* constraints, *i.e.,* those generated by individual program statements shown in Table 1, but also constraints deduced by a *global* program analysis, for example, by *global value numbering* [AWZ88]. When two SSA variables v_i and v_j are found to be value congruent, their equivalence can be reflected in G_I by a constraint edge $v_i \rightarrow v_j$ with weight 0. Following the restrictions of the value-numbering algorithm, this edge can be inserted only if the definition of v_i dominates the definition of v_j, which serves to guarantee that any constraint that holds on v_i will hold whenever v_j is executed.

Our current implementation of ABCD exploits global value numbering in a more restricted, but also more economical, fashion than described above. We do not encode the results of value-numbering analysis on G_I. Instead, we consult the congruence information on demand, in the following common scenario: When attempting to eliminate a check "**check** $A[x]$," we were able to establish that $x < B.length$ but not that $x < A.length$. In that case, we consulted the value-numbering analysis and if A and B were congruent, we obtained the desired proof that $x < A.length$.

7.2 Elimination of both lower- and upper-bounds checks

The ABCD algorithm presented in the paper eliminates upper-bound checks. To detect redundant lower-bound checks, two changes to the algorithm are needed. First, we reverse the relational operator of the constraints $C1$–$C3$ from $\leq$ to $\geq$ (see Table 1). Second, the source vertex for the shortest-path computation is not $A_i.length$, but the lower bound, which in Java is the constant vertex 0.

It is interesting to note in this context that ABCD performs (an implicit) subsumption of bound checks. For example, the upper-bound check $A[i-1]$ is redundant with respect to the upper-bound check for $A[i]$. A equivalent subsumption will be performed for lower bound checks: a lower-bound check for $A[i]$ is redundant with respect to a lower-bound check $A[i-1]$.

Although ABCD treats the *analysis* of upper- and lower-bound checks as two independent problems, these two problems are treated together in the *transformation* stage of the optimization. Next, we describe a trick that can merge an upper- and a lower-bound check into a single check instruction. This trick applies to arrays that have zero as their lower bound (as in Java). The merged check is performed as an unsigned comparison, thanks to which a negative value of the array index is thus transformed into a large positive value that is guaranteed to exceed the size of the largest array allowed in a Java virtual machine. Therefore, the upper-bound check on the unsigned value is equivalent to performing a (lower-bound) check for a negative value as well as the upper-bound check on the signed value.

7.3 Pointer Aliasing and Array SSA Form

To convince the reader that ABCD correctly handles pointer aliasing, it is useful to highlight how aliasing in a strongly typed language like Java differs from that in a language like C. We then explain how ABCD deals with the two kinds of aliases.

In Java, local variables cannot be subject to pointer aliasing because their address cannot be taken. Furthermore, no statement can change the size of an array referenced by a local Java variable. Consider the program below.

```
x = new int[10];
y = x;
y = new int[1];
x[2];          passes bounds check
```

The two middle statements do not affect the value of `x`, and hence the array access `x[2]` remains a valid (in-bounds) reference to the array created in the `x = new int[10]` statement. SSA form correctly accounts for the fact that `x` is not modified, by placing a def-use edge between the definition of `x` and the reference to `x[2]`.

```
x.f = new int[10];
y = x;
y.f = new int[1];
x.f[2];          fails bounds check!
```

Now, consider the second example. Because of the `y = x` copy statement, array access `x.f[2]` refers to the array created by the `y.f = new int[1]` statement, and hence the array access is out of bounds. Again, SSA form will correctly capture this effect, because the value of `x.f` will be the result of a memory load which is assumed to return an unknown array. Therefore, there will be no edge between the definition of `x.f` and the use of `x.f` in the array access, and hence the ABCD algorithm will conclude that `x.f[2]` refers to an unknown array.

In future work, we plan to use Array SSA form [SK98, KS98] to perform a more precise def-use analysis in the presence of pointers that will enable the ABCD algorithm to conclude that `x.f[2]` refers to the array created in the `y.f = new int[2]` statement.

8 Preliminary Experiments

We present an initial experimental evaluation of the ABCD algorithm. Our experimental results were obtained by using the Jalapeño optimizing compiler infrastructure [BCF+99] on a 166MHz PowerPC 604e processor running AIX v4.3. For these experiments, the Jalapeño optimizing compiler performed a basic set of standard optimizations including copy propagation, type propagation, null check elimination, constant folding, devirtualization, local common subexpression elimination, load/store elimination, dead code elimination, and linear-scan register allocation. Previous work [BCF+99] has demonstrated that Jalapeño performance with these optimizations is roughly equivalent to that of the industry-leading IBM product JVM and JIT compiler for the AIX/PowerPC platform.

For our experiments, we used five Java benchmarks (db, mpeg, jack, compress, jess) from the SPECjvm98 suite, seven microbenchmarks (bubbleSort, biDirBubbleSort, Qsort, Sieve, Hanoi, Dhrystone, Array) from the Symantec suite [Sym], and three other Java programs (toba, bytemark and jolt). For the SPEC codes, we use the medium-size (-s10) inputs. The focus of our measurements was on dynamic counts of bounds check operations. When we report timing information, we report the best wall-clock time from three runs.

Our preliminary implementation has several limitations. We do not use any interprocedural summary information, as the Jalapeño optimizing compiler assumes on "open-world" due to dynamic class loading. We do not perform any code duplication, such as generation of multiple versions of a loop or partitioning a loop iteration space into safe and unsafe regions [MMS98]. Most importantly, the Jalapeño optimizing compiler still lacks many optimizations (*e.g.,* global code motion) that can benefit from removal of array bounds checks. For these reasons, these experimental results should be considered a lower bound on the potential gains due to array bounds check elimination, and we expect the results to improve as Jalapeño matures.

Figure 6 shows the dynamic number of bounds checks removed by ABCD. The baseline represents all the bounds checks that were analyzed by ABCD; these are upper-bound checks, measured in dynamic terms. For the five SPEC benchmarks, the number is broken down into local checks (those from the same basic block, and checks whose redundancy required global analysis). This division is not made for the remaining programs. Manual examination of the Symantec benchmarks showed that ABCD achieved near-optimal performance, in the sense that checks that were left unoptimized were not optimizable with intraprocedural analysis (Hanoi), or requiring a complex pointer analysis (Dhrystone). In static terms, the average number of checks that were found fully redundant was about 31%. Only *bytemark* had a significant number of static checks that were partially redundant (26%).

The average number of analysis steps (*i.e.,* invocations of the recursive procedure *prove*) was less than 10 per analyzed check. This low number confirms the benefit of the sparse approach. The time to analyze one bounds check ranged from 0 to 35 milliseconds, and averaged around 4 milliseconds. This time does not include the time to construct the *e*-SSA form.

We measured run-time speedup on the Symantec benchmarks. We observed about 10% improvement. The number was lower than we expected, mainly due to the limitations of the infrastructure outlined above.

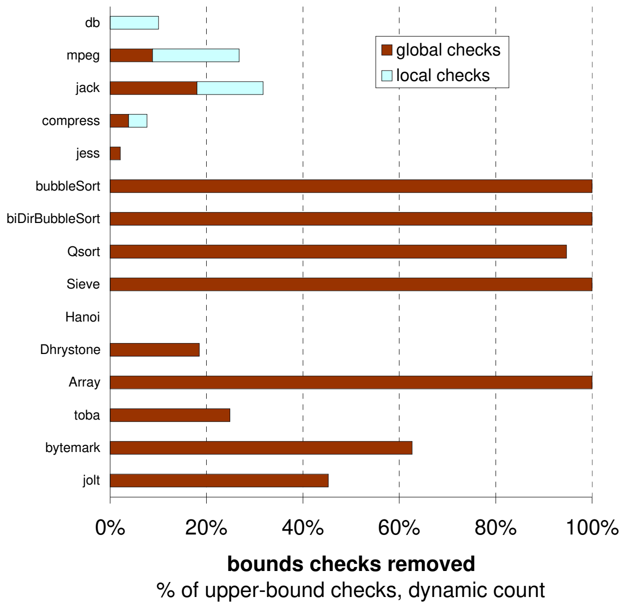

Figure 6: **The amount of bounds checks removed.** The amount shown for each benchmark represents the fraction of upper-bound checks that were removed, measured in terms of dynamic instruction counts. For the benchmarks from SPECjvm98 (the top five bars), this fraction is divided between local and global checks.

9 Related Work

Elimination of Array Bounds Checks. A number of approaches have been taken for performing elimination of array bounds checks. *Theorem-proving*-style algorithms have been used by Suzuki and Ishihata [SI77], Necula and Lee [Nec98, NL98], as well as Xu *et al.* [XMR00]. Although more powerful than ABCD, theorem proving is expensive and therefore unsuitable for a dynamic optimization setting. *Value-range analysis* has been used to compute bounds on the values of index expressions for the purpose of eliminating *full redundancy* [Har77, Pat95, RR99]. One of our goals was to handle non-scientific programs, which often have complex control flow. Therefore, in addition to full redundancy, it was important for us to handle elimination of partial redundancy. Several conventional iterative data-flow-style bounds check elimination algorithms have been developed that eliminate partial redundancy [MCM82, Gup90, Gup94, Asu92, KW95]. However, none of these algorithms exhibit the efficiency of our algorithm, which is based upon demand-driven analysis over a sparse representation. While all of the above research focuses on imperative programs, work by Xi and Pfenning considers functional programs [XP98].

Midkiff *et al.* [MMS98] have recently focused on elimination of bounds checks from scientific code written in Java. Their work is complementary to ours since our focus is on non-scientific code with complex control flow.

Techniques for eliminating partially redundant conditional branches [BGS97, MW95] determine through compile-time analysis the outcomes of branch conditionals, which are similar in nature to bounds check conditions. However, these algorithms are limited in their power as, unlike our algorithm, they do not interpret increments of variables, which is critical for bounds check elimination because index expressions in loops nearly always refer to loop-induction variables. While the algorithm by Bodik *et al.* [BGS97] uses demand-driven analysis, it is still quite expensive for use in a dynamic-optimization setting. It does not employ a sparse representation and uses expensive restructuring transformations for elimination of partial redundancy, while we perform hoisting.

Chow *et al.* [CCK+97] have developed a PRE algorithm that also operates on the SSA graph. However, their algorithm does not employ demand-driven analysis. Our algorithm is also simpler in that it does not require an explicit distinction between speculative and nonspeculative redundancy removal. Finally and most importantly, the focus of their work, like many other works on redundancy elimination [KRS92, BGS98], is on PRE of expressions and not array-bounds checks. As we have demonstrated in this paper, elimination of bounds checks has a significantly different character since assertions generated from different sources (loop exit conditions, increments of induction variables, etc.) must be analyzed in concert to prove that a bounds check is redundant. Also array-bounds check elimination exhibits *min-max* hypergraph behavior.

Demand-driven Data-Flow Analysis. Duesterwald *et al.* [DGS97] and Horwitz *et al.* [HRS95, SRH96] have developed demand-driven data flow analysis frameworks for iterative approaches that only allow lattices of finite size or height. For the purpose of bounds check elimination we need to handle infinite-height lattices and require elimination-style analysis to handle loops. Bodik *et al.* present an elimination-style demand-driven analyzer that can handle lattices of infinite height in [BGS98]. However, it operates on a dense program representation (control flow graph) and thus does not satisfy the efficiency goals of dynamic optimization. In contrast, the algorithm developed in this paper achieves further efficiency by performing demand-driven analysis over a sparse program representation.

Acknowledgements

We are indebted to Susan Horwitz, G. Ramalingam, Tom Reps, Mark Wegman, and Zhichen Xu for discussions and their suggestions on the presentation of the paper. We also thank the anonymous referees for their suggestions. Thanks to Tom Reps for the discussions on the relationship of between the inequality graph, hypergraphs and grammar problems.

References

[Asu92] J.M. Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2):109–118, June 1992.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equalities of variables in programs. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[BCF+99] M.G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

[Ber73] Claude Berge. *Graphs and Hypergraphs (transl: E. Minieka)*. North-Holland, Amsterdam, 1973.

[BGS97] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 146–158, June 1997.

[BGS98] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 1–14, June 1998.

[BGS99] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Load-reuse analysis: Design and evaluation. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999.

[CCK+97] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 273–286, June 1997.

[CFR+91] 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*, 13(4):451–490, October 1991.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms (Chapter 25.5, pages 539-543)*. MIT Press and McGraw-Hill Book Company, 1992.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[GLPN93] G Gallo, G Longo, S Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42:177–201, 1993.

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. In Mark Scott Johnson, editor, *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (SIGPLAN '90)*, pages 272–282, White Plains, NY, USA, June 1990. ACM Press.

[Gup94] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 1('-4):135–150, March-December 1994.

[GW75] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Communications of the ACM*, 18(12):716–716, December 1975.

[Har77] W.H. Harrison. Compiler analysis for the value ranges of variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[HH98] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 97–105, Montreal, Canada, 17–19 June 1998.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand Interprocedural Dataflow Analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, October 1995.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, pages 78–89, June 1993.

[KRS92] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

[KS98] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 107–120, San Diego, California, 19–21 January 1998.

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. *ACM SIGPLAN Notices*, 30(6):270–278, June 1995. Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[MCM82] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In Proceedings of a Symposium on Compiler Optimization, pages 114–119, June 1982.

[MMS98] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.

[MR79] E. Morel and C. Renviose. Global optimization by supression of partial redundancies. *CACM*, 22(2):96–103, 1979.

[MW95] Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 30 of *ACM SIGPLAN Notices*, pages 56–66. ACM SIGPLAN, ACM Press, June 1995.

[Nec98] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[NL98] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[Pat95] Jason R.C. Patterson. Accurate static branch prediction by value range propagation. pages 67–78, June 1995. Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[Pra77] V.R. Pratt. Two easy theories whose combinantion is hard. Technical report, Massachusetts Institute of Technology, 1977.

[Ram96] G. Ramalingam. *Bounded incremental computation*, volume 1089 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1996.

[RR96] G. Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, September 1996.

[RR99] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the ACM SIGPLAN 1999 Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999. ACM SIGPLAN.

[Sho81] Robert Shostak. Deciding Linear Inequalities by Computing Loop Residues. *Journal of the Association for Computing Machinery*, 28(4), 1981.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 132–143, Los Angeles, California, January 17–19, 1977. ACM SIGACT-SIGPLAN.

[SK98] V. Sarkar and K. Knobe. Enabling sparse constant propagation of array elements via array SSA form. *Lecture Notes in Computer Science*, 1503:33–40, 1998.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

[Sym] Just-In-Time Compilation. http://www.symantec.com/cafe/analysis1.html#jitcomp.

[XMR00] Zhichen Xu, Barton Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of the ACM SIGPLAN '00 Conf. on Progr. Language Design and Implementation*, page to appear, jun 2000.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices*, 33(5):249–257, May 1998. Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).