Speculative Improvements to Verifiable Bounds Check Elimination

Andreas Gampe Jeffery von Ronne David Niedzielski Kleanthis Psarris
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas, USA
{agampe,vonronne,dniedzie,psarris}@cs.utsa.edu

ABSTRACT

As a safety measure, the Java programming language requires bounds checking of array accesses. This usually translates to dynamic checks each time an array element is accessed. Static analysis can help eliminate some of those checks by proving them to be redundant, reducing the runtime overhead. Compilation of Java programs is usually method-based, and dynamic dispatch complicates interprocedural analysis. The result is a severely restricted static analysis. This paper presents a novel combination of two techniques to alleviate this problem. By assuming constraints that cannot safely be inferred from the program, the amount of provable safe bounds can be greatly extended. These constraints, called speculations, can be derived automatically from the program code by an analyzer, which assumes that there will be no violation of the array bounds. To ensure that the speculations hold at runtime, additional checks have to be injected into the code. Finding good speculations that benefit the runtime performance can be expensive. This paper shows that the speculation technique can be combined with a verifiable annotation framework, allowing most of the work to be shifted to compile-time. Experimental results show that this combination of techniques increases the number of eliminated bounds checks and can result in speedups that approach unconditional bounds check removal.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Code generation*

General Terms

Algorithms, Languages, Performance

Keywords

Java, array bounds check elimination, just-in-time compilation, optimization, verifiable annotations, SafeTSA

PPPJ 2008, September 9–11, 2008, Modena, Italy.

1. INTRODUCTION

The semantics of the Java programming language require that out-of-bounds array accesses be caught at run-time [8]. A simple way to achieve this is by performing a runtime "bounds check" before each access to an array, but this can incur a quite significant overhead [11], which is caused not only by the direct cost of conditional branches implementing the array bounds checks but also by lost opportunities for optimization and parallelization due to Java's precise exception semantics.

One way to reduce this overhead is to apply a static analysis that identifies some of the array element accesses that will never cause an out of bounds exception and to optimize the program by eliminating those unnecessary bounds checks.

Because Java is designed to use a virtual machine providing platform-independence, dynamic class-loading, and safety, most optimizations are done at runtime during just-in-time (JIT) compilation. Although it is possible for JIT compilers to perform bounds check elimination, the most precise analyses (e.g.,[11, 13, 16]) are too expensive to run in JIT compilers. For that reason, faster analyses (e.g., [2],[20]), which eliminate fewer unnecessary array bounds checks, are used instead.

An alternative light-weight bounds-check elimination algorithms is to develop an annotation scheme that allows more expensive static analysis to be performed by an annotator once prior to distribution as part of the compilation from source code. The results of this analysis can be shipped to the code consumer as annotations added to bytecode. The code consumer's JIT can then optimize the bytecode based on these annotations. This is only safe, however, if the annotations can be verified to be correct. Otherwise a malicious annotator could lie causing the code consumer to omit necessary bounds checks.

Any intraprocedural static analysis will be limited in its ability to eliminate bounds checks whose redundancy depend on relationships among variables that are passed as parameters. Interprocedural analysis can expose those relationships, but suffers practical limitations in Java. The potential complications arise from the combination of interprocedural analyses's need to know what method implementations are being called, Java's dynamic class-loading and Java's dynamic dispatch of most method invocations. In general, new classes can be loaded at any time, and if a new class overrides an existing method, this can invalidate assumptions about which method implementations can be invoked at particular call sites. This in turn may invalidate the results of the analysis, and may require discarding machine code that was optimized using that analysis. The overhead in complexity and in performance is such that (except for inlining) most Java optimizations are done only at the intraprocedural level.

Another technique for optimizing code which depends on prop-

erties which are stable but cannot be proven through intraprocedural analysis is to utilize *speculative optimization*. A speculative optimization uses some heuristic to identify some property (such as that a variable has some particular value or that a certain hot branch will be taken) is likely to hold. It then specializes the code under the assumption that the property holds, but also uses some mechanism to revert to an unspecialized version when the property does not hold. Würthinger et al.'s recent work with pattern-based speculations for array bounds check elimination has shown that this technique can achieve considerable speedups [19].

In this paper, we present a novel combination of verifiable annotations and speculative optimization for bounds check elimination This allows both the process of traditional static analysis as well as finding beneficial speculations to be shifted to compile time, opening a path to more complex speculation finding algorithms for JIT compilers.

2. VERIFIABLE ANNOTATION FRAMEWORK

Our approach is based on a verifiable boundscheck elimination framework[18]. This framework uses linear inequality constraints to encode the ranges of integer variables, which are more general then the widely-used difference constraints (e.g. [2],[4],[19]), without imposing a substantial cost in the code consumer. The ultimate goal is then to derive constraints of the form $0 \leq$ *index* and *index* $<$ a. length for an array access `a[`*index*`]`, that is, that no out-of-bounds exception will occur.

The following constraints are allowed in this framework:

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

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

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

Proofs of certain constraints, e.g. the redundancy of a bounds check, are gained by combining program constraints. There are two primitive constraint combinators:

+ (addition) takes two inequalities and combines them by a adding them together to produce a new linear equality

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

2.1 Integer overflow/underflow

Combining linear inequality constraints relies on properties which hold for standard arithmetic in $\mathbb{Z}$. The Java programming language's primitive integer type (`int`) cannot represent any number in the integer domain, but is instead, restricted to integers that can be stored into a 32-bit word using 2's complement representation. When integer arithmetic operations result in a value less than -2^{31} (referred to as `MIN`) or more than $2^{31}-1$ (referred to as `MAX`), the computation "wraps around" so that only the least significant 32-bits are retained.

This difference between the mathematical domain of integers ($\mathbb{Z}$) and Java's primitive `int` type needs to be taken to account during integer range analysis to ensure the soundness of the array bounds check elimination. Therefore, linear inequality constraints must be expressed in the domain of integer numbers. But linear inequality constraints that would seem to fall out naturally from source code statements may not hold when constraints among 2's complement variables are expressed as inequalities of variables in $\mathbb{Z}$. For example, consider the proposition that the Java statement, `x = y + z`, implies that $-[[\text{x}]]+[[\text{y}]]+[[\text{z}]] \leq 0$ (where $[[x]]$ denotes the value in $\mathbb{Z}$ of the Java variable x). This proposition will not hold if $y+z$ overflows.

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

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

Table 1: Axiomatic Linear Inequalities

2.2 Valid constraints

Except for a handful of universally valid axiomatic constraints (see Table 1), a constraint cannot be considered valid unless an annotation claims it to be true. These claims must match one of the claim rules listed in Tables 2 and 3. These may be categorized as:

- the claim that a bounds check is unnecessary (Rule (0) in Table 2)
- a linear inequality constraint anchored to a particular integer or array operation (Rules (1)-(14) in Table 2)
- a constraint that is anchored to the join node in the control flow graph (Rule (15) in Table 2)
- a predicate constraint anchored to an out-going control flow graph edge of a conditional branch instruction (Rules (16)-(17) in Table 3)
- a predicated linear inequalities constraint anchored to an integer comparison operation (Rules (18)-(29) in Table 3)
- a predicated predicate anchored to a boolean logic operation (Rules (30)-(33) in Table 3)

With the exception of rule (15), the constraints are dictated by the rule and the instruction to which it is attached. The validity of a claim depends only on the semantics of the instruction to which the claim is anchored and the satisfaction of any proof obligations included in the claim rule. If the proof obligations are discharged, then the claim is guaranteed to be valid in the region dominated by that claim's anchor.

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

2.3 Merging Constraints at Join nodes

In order to simplify tracking the validity of constraints, our verification system assumes programs are in static single assignment form (SSA) such as SafeTSA [1].Rule (15), which is associated with join nodes (i.e., nodes in the control flow graph that have more than one predecessor and may contain ϕ-functions), requires some explanation. This claim rule is a consequence of the instructions being in SSA form, and exists to allow constraints that are valid in

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

Where x, y, and z are Java variables of type `int`, a is a Java variable containing an array reference, and c is an integer constant
* proof obligation is based on the claimed constraint and all of the ϕ-functions in the join node. See Section 2.3 for details.

Table 2: Claim Rules for Linear Inequality Constraints

Instruction Form	Rule	Claimed Constraint
Conditional Branches		
if b goto ... *[branch taken]*	(16)	b
if b goto ... *[not taken]*	(17)	$\neg b$
Predicated Inequalities		
`b = x == y`	(18)	$b \Rightarrow [[x]] - [[y]] + 0 \leq 0$
	(19)	$b \Rightarrow -[[x]] + [[y]] + 0 \leq 0$
`b = x != y`	(20)	$\neg b \Rightarrow [[x]] - [[y]] + 0 \leq 0$
	(21)	$\neg b \Rightarrow -[[x]] + [[y]] + 0 \leq 0$
`b = x <= y`	(22)	$b \Rightarrow [[x]] - [[y]] + 0 \leq 0$
	(23)	$\neg b \Rightarrow -[[x]] + [[y]] + 1 \leq 0$
`b = x < y`	(24)	$b \Rightarrow [[x]] - [[y]] + 1 \leq 0$
	(25)	$\neg b \Rightarrow -[[x]] + [[y]] + 0 \leq 0$
`b = x >= y`	(26)	$b \Rightarrow -[[x]] + [[y]] + 0 \leq 0$
	(27)	$\neg b \Rightarrow [[x]] - [[y]] + 1 \leq 0$
`b = x > y`	(28)	$b \Rightarrow -[[x]] + [[y]] + 1 \leq 0$
	(29)	$\neg b \Rightarrow [[x]] - [[y]] + 0 \leq 0$
`b = c & d`	(30)	$b \Rightarrow c$
	(31)	$b \Rightarrow d$
`b = c | d`	(32)	$c \Rightarrow b$
	(33)	$d \Rightarrow b$

Table 3: Claim Rules for Conditional Branches & Predicated Inequalities

all of the join node's predecessors to the program region dominated by the join node, even though the join node may not be dominated by any of the join node's predecessors. This can be used to annotate loop invariants, but can also be used to show that some constraint holds after an if statement because it holds at the end of the then part of the if statement and at the end of the else part of the if statement.

Rule (15) is unique in a few ways. First, the claim is anchored to a join node as a whole rather than to any individual ϕ-function contained inside of it. Second, the constraint being claimed can be any constraint (which must be described completely in the annotation) as long as the required proof obligations are discharged. Third, one proof obligation is anchored to each of the join node's predecessors and can only use the claims which are valid at the branch instruction in that predecessor. Fourth, the proof obligations are obtained by taking the claimed constraint and substituting, for each occurrence of variables which are on the left-hand side of ϕ-functions in the join node, the variable on the right-hand side of that ϕ-function that is associated with that proof obligation's anchor.

Note that in the case of loops this does not create circular proofs. The rules above result in proofs by induction. There is one obligation that is bound to the initial value of the ϕ-function. This obligation's proof cannot use the claim itself since the ϕ does not dominate this predecessor. It constitutes the base case of the induction. The other obligation is bound to the end of the loop and thus describes the induction step. It is legal to use the induction claim, and thus the ϕ's claim, here.

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

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

[1] $-i_0 \leq 0$
anchored at end of Init
$\because$ Ⓐ

[2] $-i_2 \leq 0$
anchored at end of Body

$\because$	$-i_2 + i_1 + 1 \leq 0$	Ⓕ
+	$-i_1 \leq 0$	Ⓑ
	$-i_2 + 1 \leq 0$	
+	$-1 \leq 0$	②
	$-i_2 \leq 0$	

[3] $-i_1 \leq 0$
$\because$ Ⓑ

[4] $i_1 - a_0.\text{length} + 1 \leq 0$

$\because$	$t_1 \Rightarrow i_1 - t_0 + 1 \leq 0$	Ⓓ
MP	t_1	Ⓔ
	$i_1 - t_0 + 1 \leq 0$	
+	$t_0 - a_0.\text{length} \leq 0$	Ⓒ
	$i_1 - a_0.\text{length} + 1 \leq 0$	

[5] $i_1 + 1 - \text{MAX} \leq 0$

$\because$	$t_1 \Rightarrow i_1 - t_0 + 1 \leq 0$	Ⓓ
MP	t_1	Ⓔ
	$i_1 - t_0 + 1 \leq 0$	
+	$t_0 - a_0.\text{length} \leq 0$	Ⓒ
	$i_1 - a_0.\text{length} + 1 \leq 0$	
+	$a_0.\text{length} - \text{MAX} \leq 0$	③
	$i_1 + 1 - \text{MAX} \leq 0$	

Figure 2: Proofs for the Claims in the Idiomatic Loop

2.4 An Example: an Idiomatic Java Loop

Consider the Java function:

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

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

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

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

2.5 Verifying Annotations

Verification of these linear constraint annotations is straight forward. The system merely requires a pre-order traversal of the program's dominator tree, during which a list of active claims is maintained. When an annotation is encountered, the claim rule is checked against the instruction it is anchored to. If the type of instruction does not match the claim rule, if the referenced claims are not in the active list, or if the referenced claims do not match the kinds

of constraints required by a combinator, then the annotation will be rejected. Otherwise, the proofs are checked by loading referenced claims from the active list, applying the indicated combinators, and computing a resulting proof. If this discharges the proof obligation for that claim, then the claim is added to the active list until it no longer dominates the current node (claims are added and removed from the active list in LIFO order).

3. SPECULATIVE OPTIMIZATION

Classical static bounds check analysis can be severely hindered by programming patterns that hide dependencies between program variables. An example of this is:

```
void f(int n, int a[]) {
    for (int i = 0; i < n; i++)
        a[i] = a[i] + 1;
}
void g(int a[]) {
    f(a.length, a);
}
```

In the context of method `f` there is no relation between `n` and `a.length`, which results in any system not being able to safely remove the bounds checks.

If `g` is the only method calling `f`, interprocedural analysis will show that there is a relation between the method parameters, thus allowing the removal of the bounds checks. But since the Java Virtual Machine supports dynamic class-loading, this may change at any time. Thus ensuring the validity of the analysis needs additional overhead of the VM.

Inlining the method `f` into `g` will also expose the dependency, but has the same problem with dynamic class-loading as interprocedural analysis. The optimizer has to ensure that the correct method gets inlined into `g`, which may change upon loading new classes.

Specialization is a technique that does not rely on the context a method is called in. Instead, the specializer assumes certain properties of the program state at certain points in the method and may then optimize the code accordingly. This can result in a set of versions of a method body or smaller code fragments, all optimized for different assumptions. The specializer then inserts *dispatch* code to choose at runtime which version to run.

Since the properties the specializer assumes are not necessarily based on any other analysis, we call them speculation in our framework, the resulting optimized bounds check removal speculative optimization.

Speculations form a third group of annotations beside basic rules and axioms in our framework. They can be anchored to any location in a program and claim an arbitrary linear inequality over the SSA variables arising from the source. This is only valid if those sources are defined at the point of usage, that is their definition dominates that location. It is adequate to anchor speculations to instructions and define that they are valid immediately before that location.

Furthermore speculations have to be proven dynamically by inserting checks into the program code. While a scheduling algorithm might be used at JIT compile-time to find a good placement of those checks, it is simpler and more efficient to use the anchor of the speculations. This way the scheduling can be performed while annotating the program, shifting this work to producer-side compile-time.

With this embedding in the verification framework, speculations can be referenced in proofs just like any other axiom, avoiding unnecessary complexity of the architecture.

4. SPECULATION FRAMEWORK

The extension of the verification framework allows speculations to be used in the verification. The implementation is split between code producer and code consumer side. On the producer side, the annotation generator of the verification framework has to be extended to create speculations and make them available for proofs. We call this component *analyzer* from here on. On the consumer side, the verifier has to be extended to allow speculations in proofs. Furthermore a component has to be added that injects the necessary runtime checks into the program code and creates unoptimized fallback versions. We call this component *specializer* from here on.

Separating the components has several advantages. It allows to change each if necessary or better technology becomes available. Furthermore the specializer can work on the consumer side. This results in significant size benefits of the mobile code, since specialization will take place after transmission. With approriate runtime support, it is possible to avoid code duplication altogether (e.g., [19]).

The task of the analyzer is to incorporate the possibilities of speculative optimizations, that is finding good speculations. A general assumption in finding speculative constraints should be that a normal program runs without bounds check exceptions (that is no out-of-bounds array access is attempted in a safe program). Given this definition, the safety constraints of an index of an array access could be used directly as a speculation, albeit a rather nonsensical one. If we simply replace the bounds checks with its safety constraints, nothing is gained since the runtime system has to ensure the speculation by inserting checks again. There are two obvious ways out of this dilemma. One could make use of one speculation in multiple bounds checks, or ensure that a speculation for a certain bounds check would be checked less often than the bounds check itself. An example of the latter occurs if the speculation can be moved through loop-invariant code motion.

This approach can be implemented by a two-stage process. The first stage analyzes every bounds check for itself, trying to find the best speculation to remove that check. The second stage then consolidates the found speculations and proofs, using heuristics to remove those that do not seem to be beneficial.

4.1 Patterns

Our analyzer uses code patterns to find good speculations in the first stage, which cover a range of common program constructs. Note that for simplicity examples in the following paragraphs are given in high-level code, while the framework itself uses SSA form.

Loop-invariant bounds checks Consider the following code fragment derived from the example in section 2:

```
void f(int a[], int x) {
    int sum = 0;
    for (int i = 0; i < x; i++)
        sum = sum + a[i];
}
```

In this code, neither the array `a` nor the upper bound `x` are changed during the execution of the loop. They are loop-invariant. The lower bounds check can be statically proven redundant. The index is a phi function, so both control flows that reach it have to be proven separately. The first one, coming from before the loop, is trivial. The second one requires that $[[\texttt{i}]]+1 \geq 0$. With our assumption of $[[\texttt{i}]] \geq 0$, this is true. It can also be shown that there will be no overflow. When `i`+1 gets computed, it is known that `i`<`x` (rules (16) and (24)) and `x`≤`MAX` (axiom ⑤), which can be combined to yield `i`+1 ≤`MAX`, which is the obligation for no overflow.

There is no similar proof for the upper bounds check. All that

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

Figure 3: Claims and Proof Obligations for an Extended Idiomatic For Loop

[1] $-i_0 \le 0$
anchored at end of Init
$\because$ (B)

[2] $-i_2 \le 0$
anchored at end of Body

$\because$	$-i_2 + i_1 + 1 \le 0$	(F)
+	$-i_1 \le 0$	(C)
	$-i_2 + 1 \le 0$	
+	$-1 \le 0$	(2)
	$-i_2 \le 0$	

[3] $-i_1 \le 0$
$\because$ (C)

[4] $i_1 - a_0.\text{length} + 1 \le 0$

$\because$	$t_1 \Rightarrow i_1 - x + 1 \le 0$	(D)
MP	t_1	(E)
	$i_1 - x + 1 \le 0$	
+	$x - a_0.\text{length} \le 0$	(α)
	$i_1 - a_0.\text{length} + 1 \le 0$	

[5] $i_1 + 1 - \text{MAX} \le 0$

$\because$	$t_1 \Rightarrow i_1 - x + 1 \le 0$	(D)
MP	t_1	(E)
	$i_1 - x + 1 \le 0$	
+	$x - \text{MAX} \le 0$	(3)
	$i_1 + 1 - \text{MAX} \le 0$	

Figure 4: Proofs for the Claims in the Extended Idiomatic Loop

can be derived is that `i<x` at the location of the array access. The bounds check is thus not fully redundant and cannot be removed directly. Since neither the array `a` nor the upper bound `x` change during the loop, the upper bounds check is partially redundant. It can be moved before the loop, where it only gets executed once. Therefore the speculation will be `x≤a.length`, anchored at a point before the loop where both `x` and `a` dominate it. The SSA code and these proofs are shown in figures 3 and 4.

Care has to be taken to account for the precise exception semantics of Java. All exceptions have to be reported faithfully (that is, leaving the same program state as an unoptimized version). If in the above example the loop would be executed n times before throwing an exception, the optimized code has to behave exactly the same. Thus it is not valid to throw an exception before the loop. In our system the specializer takes care of those semantics.

While this simple pattern already applies to a significant number of loops, it can be generalized to apply to more. The general linear inequality constraints of the verification framework are allowing us to potentially annotate loops where the initial value, condition and array index are extended to linear expressions. Note however that in such cases, one speculation will not suffice anymore. To prove that there will be no overflow, further speculations on the values of the variables in the expressions might be necessary. Since the valid combinations of the values with no overflow form a rather large set, the analyzer will have to guess reasonable values, for example guessing that the values are bigger or equal to zero. These speculations are more arbitrary than the original speculation that there will be no out-of-bounds array access, thus it is not clear how they impact runtime performance. Furthermore, adding more speculations means adding more runtime checks. If the loop is not executed enough times, the gain might not offset the JIT compile time and runtime check overhead. Thus it is reasonable to set an upper bound for the complexity of the expressions in the loop. We found that restricting the expressions so that the resulting speculations have at most three variables is a good compromise that catches all significant hot-path loops in our benchmarks without too much overhead.

Coterminous-array loops are an extended case of the above. Consider the following Java method:

```
void f(int a[], int b[]) {
    for (int i = 0; i < a.length; i++)
        a[i]= a[i] + b[i];
}
```

In this code, it is quietly assumed that the array `b` is at least as long as array `a`. So this is a reasonable speculation, which is enabled by allowing array lengths in the speculations of loop-invariant bounds checks. In the example above this results in a speculation of the form `a.length`$\leq$`b.length`.

Rectangular arrays Scientific algorithms often make use of multi-dimensional arrays for data storage. A computation could look like the following fragment.

```
void f(int a[][]) {
    int n = a[0].length
    for(int i = 0; i<n; i++) {
        b = a[i]
        for(int j=0; j<n; j++)
            b[j]...;
    }
}
```

Many applications only use rectangular arrays, though Java allows multi-dimensional arrays to have any form. That suggests that it might be beneficial to speculate that any higher-dimensional array is rectangular. Note however that this requires certain safeguards. If the given array is non-local, concurrent modifications might occur during the execution of the loop. The Java specification allows to circumvent this in well-defined cases through load elimination, which translates to creating a local copy of the outer array.

As a generalization of rectangular arrays, this system can also be used to only establish a lower or upper bound on the length of arrays. This broadens the applicability in cases of "jagged" arrays that still allow safe bounds check removal, without added complexity to the process.

It is noteworthy that the previous techniques described in the first analyzer stage will probably already produce annotations for some instances of rectangular arrays. Their speculations can only be anchored after the sub-array has been retrieved. This forces the runtime system to insert checks in the middle of the program code, which might either increase code in the hot path or not be supported at all. Thus rectangular array speculation involves a trade-off: incurring some overhead in copying the array, but allowing speculation at the beginning of a method.

In our implementation, we focus on an array and its direct subarrays. In that case, the first stage of the analyzer is augmented to recognize the origin of an array. If it is another array, we mark that one as possibly rectangular and add new constraints to our program representation. These involve a new virtual variable that stands for the lengths of all subarrays of the rectangular array as well as a variable describing the actual length of the current subarray. With these added constraints, the analyzer might be able to prove a bounds check safe with or without further speculations. In either case, the added constraints are generalized to form a statement over the possibly rectangular array and are added as speculations. A new rule (34) is introduced to apply those speculations to a certain sub-array, replacing the added constraints in proofs.

Applied to the example above, this yields the speculation $v_a \geq$ `a.length`, where v_a is the virtual variable describing the length of sub-arrays of `a`.

Simple bounds checks For any bounds check that is not caught by the previous techniques and cannot be proven statically, we can introduce speculations that coincide with the safety requirements. These speculations can then be anchored at the earliest point they are valid, that is their variables are defined.

Note that this in itself does not make much sense since we trade the bounds check for its two equivalent sub-checks, adding code and JIT compile-time overhead. It might even trade one check for two since lower and upper bounds check might be combined into one instruction, e.g. using an unsigned comparison instead of two signed ones.

4.2 Consolidation

After a whole method has been analyzed in the first stage, every bounds check will either be statically proven or have a speculative proof. The task of the second stage is to remove those speculative proofs that are unlikely to have a beneficial effect on the runtime. In addition, it combines and replaces speculations to reduce their number and make them more precise.

The replacement follows a simple rule. If a proof uses a speculation $c_1x_1 + c_2x_2 + \cdots + c_0 \leq 0$, and there is another speculation $c_1x_1 + c_2x_2 + \cdots + c_0' \leq 0$ with $c_0' > c_0$, then replace the first one with the second, since it can be combined with axiom ② to prove the first. If the new speculation does not dominate the anchor of the proof, then move it up in the dominator tree until it does both dominate its old position and the proof's anchor. This ensures that any produced proof stays valid when switching the speculations. For an example consider the following code fragment:

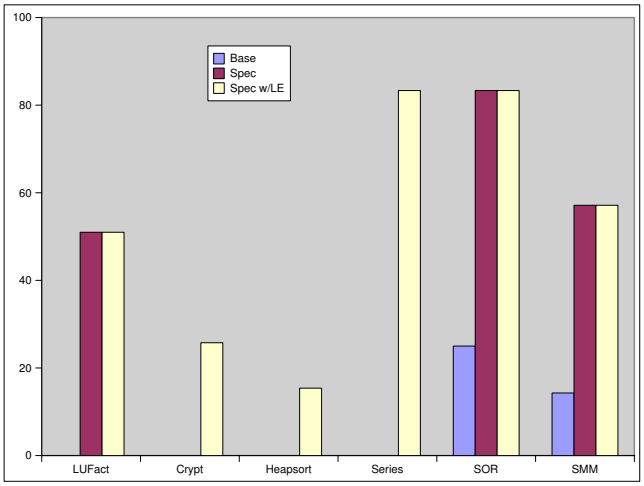
```
a[0] = 0;
a[1] = 1;
a[2] = 2;
```

The first stage of the analyzer will, among others, create three speculations to prove the upper bounds checks. These speculations are $-$`a.length`$+1 \leq 0$, $-$`a.length`$+2 \leq 0$ and $-$`a.length`$+3 \leq 0$, anchored before the first, second and third instruction respectively. The above rule will consolidate these three speculations into the last one, which has to be moved to the head of the fragment to cover all three bounds checks.

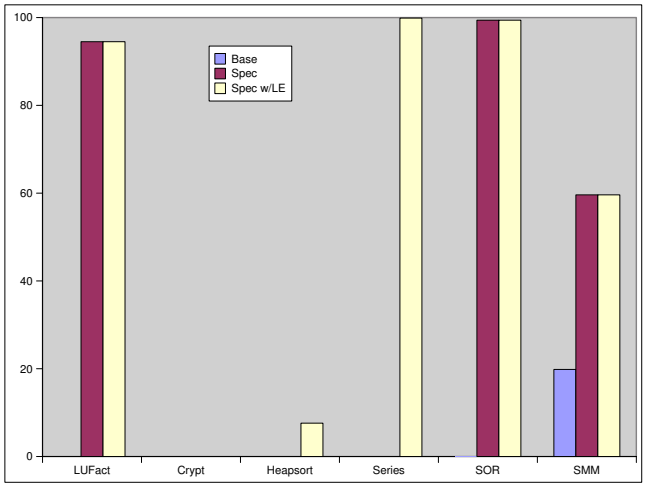
A simple heuristic is used to approximate the runtime impact of a speculation. For every speculation, all bounds checks proofs that depend on that speculation are gathered. If a speculation is used to prove a bounds check inside a loop, but is anchored itself outside the loop, we optimistically assume that it is beneficial. But if instead a speculation is only used in proofs of bounds check of the same nesting depth, we require it to be used in at least three such proofs. This is based on the assumption that a normal bounds check is implemented as one unsigned comparison in a virtual machine. Since there might be two speculations needed to prove upper and lower bounds checks, the speculations can become beneficial with three bounds checks.

4.3 Runtime dispatch

As mentioned in the previous sections, speculation can potentially change the program semantics. If exceptions are thrown immediately when a runtime check fails, it may violate Java's precise exception semantics. Specialization can be used to circumvent this problem. By holding multiple versions of the fragment under the control of the speculation, at least one optimized and one unoptimized, the correct semantics can be ensured by choosing the right fragment at runtime. If no out-of-bounds access will happen as indicated by all speculations holding, choose the fully optimized version. Otherwise fall back to the unoptimized code, which will throw exceptions at the right point. Note that it is possible to choose between more than two fragments, which can be necessary with overlapping speculation ranges.



(a) Static Percentage



(b) Dynamic Percentage

Figure 5: Eliminated Bounds Checks

5. IMPLEMENTATION AND RESULTS

To create a prototype mixed annotation & verification environment, we used our Verifiable Bounds Check Elimination (VBCE) system[18], which is implemented as an annotation generator and annotation verifier within the SafeTSA system. The SafeTSA system is based on Jikes RVM 2.2.0. The static component of the annotation generator creates proofs using a novel Constraint Analysis System (CAS)[14]. This generator was extended to generate speculations in case the CAS was not able to prove a bounds check redundant statically. The implementation followed the patterns and algorithms outlined in the previous section.

The baseline verifier of VBCE was extended to allow speculative constraints in proofs. Our current implementation is limited to a simple form of method versioning with only two version of the method. Every speculation that does not fit this restriction is discarded. This is due to limitations in the JikesRVM platform which makes speculating at other points overly complex.

Both analyzer and specializer support load elimination and code motion in conformance with the Java concurrency and precise exception semantics rules [6]. This allows to keep some of the speculations that would otherwise have to be discarded.

We evaluated our prototype system using section two of the Java Grande Forum benchmarks [3]. These benchmarks were compiled into SafeTSA and optimized using common subexpression elimination, which eliminates duplicate bounds checks using SafeTSA's safe-element-reference type [1].This version of each class is used as the baseline for the evaluation.

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

5.1 Eliminated Bounds Checks

The number of removed bounds checks is an important metric when comparing bounds check elimination algorithms. We extracted the static number of bounds checks in the benchmark code and compare it to the number of removed bounds check. The four different methods shown in figure 5(a) are Baseline bounds check elimination with CAS (Base), Speculative Optimization on top of baseline (Spec) and Speculative Optimization on top of baseline with load elimination and code motion (Spec w/ LE).

In all cases the number of eliminated bounds checks increases significantly when activating speculation. Note that Crypt, Series and Heapsort have to use Load Elimination to remove any bounds checks. Also note that the number of removed bounds checks that our annotator generates proofs for is somewhat higher. As mentioned above, unsupported speculations get rejected by the specializer.

As a second set of results we extracted the dynamic number of bounds check executed. The program code is extended to increment a counter before every bounds check instruction. Furthermore, every check for a speculation is counted as one bounds check, except rectangular array speculations. These count as as many bounds checks as length checks for sub-arrays have to be performed. We used those numbers to compute the percentage of dynamic bounds checks removed and present the results in figure 5(b).

The results differ from the static count. In most cases, they indicate that our speculations captured most of the frequently executed bounds checks. Exceptions are Crypt and Heapsort, where most of the removed bounds checks are outside the hot path. In the case of Crypt, most of the speculations that the annotator produced were incompatible with the specializer and had to be rejected. Heapsort exhibits complex index arithmetics in its hot methods, which could not be successfully analyzed yet.

5.2 JIT Compile-time

The overhead of the verification stage to the JIT compile process can usually be neglected. Our experiments show that verification and program modification takes up less than four percent of the whole JIT compile time in all benchmarks, with an average of 2.1%.

5.3 Runtime Impact

The resulting speedup of array bounds check elimination depends on the type and programming style of application. If the annotator is able to find proofs for bounds checks in hot program paths, the elimination of bounds checks can be very effective, given

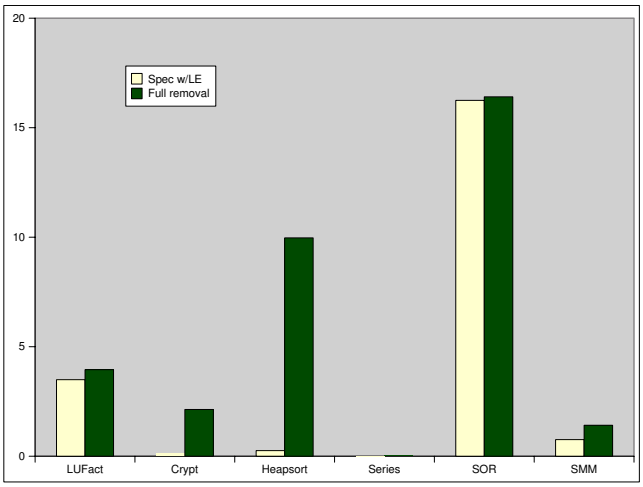


Figure 6: Speedup in percent

that the verifier supports the speculations.

Executing the code without any bounds checks gives an upper bound of the achievable speedup through bounds check elimination. Therefore we compare the speedup achieved by our framework with this value to examine the efficiency of our implementation. Note however, that simply removing all bounds checks does not conform to the Java Language Specification, and can produce unsafe code. A program could freely access memory in the absence of `ArrayOutOfBoundsException`s by accessing an array with an out-of-bounds index. The benchmarks examined do not exhibit this behaviour, since all array accesses are within the correct bounds, so are not affected. The resulting speedups are reported in figure 6.

In the cases of LUFact and SOR, where the execution of bounds checks takes up a considerable amount of execution time and the benchmarks enable speculations, our framework is able to approach the theoretically maximum speedup, as indicated by the removal of 88.4% and 99% of the dynamic bounds check overhead.

Crypt shows that even a significant number of static removed bounds checks does not ensure a resulting speedup, as previously indicated by the dynamic bounds count. A more complex specializer is necessary to achieve significant speedups here.

The Series benchmark displays a rather strange behaviour. While we are able to eliminate nearly all bounds checks, the performance of the benchmark does not improve significantly. In this case, the main factor of the execution time is actually numerical computation, not the array access. This means that bounds check removal is never able to improve performance much, as indicated by the equally low speedup when bounds check are fully removed.

6. RELATED WORK

The concept of annotating programs with proofs of various properties, including safety of bounds checks, that could then be verified was explored as part of Proof-Carrying Code [12], which uses first-order logic. Our framework is more limited: array bounds checks rather than program type safety, and linear constraints of integers rather than first order logic. The tighter focus makes our approach a direct replacement for runtime array bounds-check elimination, as well as should result in shorter, simpler proofs, and faster verification times than those based on a complete proof carrying code framework.

Program specialization is a well-studied topic [17, and references therein]. DyC [7] and others use annotations to guide the specialization process, but rely on programmers to annotate source files. Calpa [10] automates this by profiling a representative input. A key difference from our work is their reliance on constant values, whereas our system speculates on relationships among program variables and symbolic constants, helping in the removal of array bounds checks.

There have been several works addressing the array bounds check problems in Java. Moreira et al. [11] used heavy-weight loop-based transformations and optimizations to optimize bounds checks in scientific applications; their goal was a traditional static compiler for Java programs, so their approach does not support just-in-time compilation and is not a general solution to the Java bounds check problem.

The ABCD algorithm [2] provides global bounds check elimination based on extended-SSA form and difference constraints, it is quite efficient but has some limitations since it can only obtain difference constraints that can be overlayed onto the SSA graph. Menon et al. [9] extended the ABCD algorithm to produce optimized programs augmented with verifiable proof variables, but the verifier would be required to make judgements about facts using integer linear programming instead of checking an explicit proof.

Qian et al. [15] use an iterative dataflow analysis based on difference constraints to annotate bytecode with an indication of which bounds checks are unnecessary, but no mechanism is provided to verify that these annotations are correct. Chen and Kandemir [4] describe a method for annotating the fixed point of an iterative dataflow analysis of integer variable ranges which can then be verified using a single iteration of the same algorithm, but their constraints are limited to a subset of difference constraints.

Würthinger et al. [19] have developed a bounds check elimination for use in the HotSpot JIT compiler, which similarly identifies simple patterns in the source code. Additionally it uses speculation in a similar manner to our approach by recognizing patterns that can be exploited. Their algorithm is intended as a runtime optimization and thus less comprehensive then ours. It is only based on difference constraints, which restricts the complexity of the analysis. There are several bounds checks in our benchmarks that can be eliminated with general linear constraints, but not with difference constraints. Furthermore their pattern set is more restricted than our current implementation. While our system can be easily extended to include any new analysis to find speculations, Würthinger et al.'s algorithm can only use light-weight techniques adapted for runtime use.

The specialization component of their algorithm is implemented by using the on-stack-replacement (OSR) capabilities of the HotSpot compiler. A whole method is compiled in optimzed form only, where speculation checks initiate de-optimization and fallback to interpreted code on mis-speculation. This avoids costly replication of code and allows speculations to be placed at any point. While our framework in general allows any specialization system and could thus work with OSR, our current implementation is not as powerful due to limitations of our version of JikesRVM.

The different virtual machines and specialization algorithms make direct comparisons of the results difficult. The dynamic bounds check removal counts are generally compareable in that they show the algorithms able to eliminate most of the bounds checks in some cases. We expect further work on our analyzer will set us apart in those cases which don't exhibit optimal behaviour yet. The resulting speedups after bounds check elimination differ, as a result of the underlying platforms, but also show the same tendencies.

With the notable exceptions of the proof-carrying code framework for the Special J compiler [5] and the work of Würthinger et

al., prior work, including the ABCD algorithm [2] and Chen and Kandemir's verification system [4], has generally not addressed integer overflow as it exists in Java. The proof-carrying code framework for Special J considers integer overflow as part of rules for 32-bit signed and unsigned comparisons, which can be used to correctly prove that incremented loop induction variables are within array bounds [5], but appear to to be less general than the rules (9)–(14) of our verification framework. Würthinger et al. use a mixture of static analysis and runtime checks to address the overflow problem. This approach is similar to ours, in that our verification framework will try to prove no-overflow statically, if possible, falling back to speculations if necessary.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have described the combination of verifiable annotations with speculation to facilitate bounds check elimination for Java programs. These annotations can be produced by an annotator using thorough static analysis and a speculative analyzer, and then verified efficiently at runtime. Our speculations are based on variable relationships and are enforced by a specializer at runtime.

Experimental results show that this approach is able to significantly increase the number of eliminated bounds checks in several cases, almost completely eliminating the bounds-check overhead in the optimized programs.

In future work, we plan to implement more complex analyzers like weakest precondition computation to exploit the ability to use expensive algorithms at compile-time. We would also like to compare and augment our speculative optimization with the use of interprocedural analysis to guide the speculation-finding process.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] W. Amme, N. Dalton, M. Franz, and J. von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, volume 36, pages 137–147, June 2001.

[2] R. Bodík, R. Gupta, and V. 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, 2000.

[3] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, May 2000.

[4] G. Chen and M. 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, 2005.

[5] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107, 2000.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley Professional, Boston, MA, USA, 3 edition, 2005.

[7] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[9] V. S. Menon, N. Glew, B. R. Murphy, A. McCreight, T. Shpeisman, A.-R. Adl-Tabatabai, and L. 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, 2006.

[10] M. Mock, C. Chambers, and S. J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *International Symposium on Microarchitecture*, pages 291–302, 2000.

[11] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Trans. Program. Lang. Syst.*, 22(2):265–295, 2000.

[12] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.

[13] T. V. N. Nguyen and F. Irigoin. Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst.*, 27(3):527–570, 2005.

[14] D. Niedzielski, A. Gampe, J. von Ronne, and K. Psarris. A verifiable, control flow aware constraint analyzer for bounds check elimination. Technical Report CS-TR-2008-009, Computer Science, The University of Texas at San Antonio, 2008.

[15] F. Qian, L. J. Hendren, and C. 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.

[16] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, 2005.

[17] A. Shankar, S. S. Sastry, R. Bodík, and J. E. Smith. Runtime specialization with optimistic heap analysis. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 327–343, 2005.

[18] J. von Ronne, A. Gampe, D. Niedzielski, and K. Psarris. Safe bounds check annotations. *Concurrency and Computation: Practice and Experience*, (to appear).

[19] T. Würthinger, C. Wimmer, and H. 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.

[20] J. Zhao, I. Rogers, C. Kirkham, and I. 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.