Safe, Multiphase Bounds Check Elimination in Java

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

January 28, 2010

Abstract

As part of its type-safety regime, the Java semantics require precise exceptions at runtime when programs attempt out-of-bound array accesses. This paper describes a Java implementation that utilizes a multiphase approach to identifying safe array accesses. This approach reduces runtime overhead by spreading the out-of-bounds checking effort across three phases of compilation and execution: production of mobile code from source code, JIT compilation in the virtual machine, and application code execution. The code producer uses multiple passes (including common subexpression elimination, load elimination, induction variable substitution, speculation of dynamically-verified invariants, and inequality constraint analysis) to identify and prove redundancy of bounds checks. During class-loading and JIT compilation, the virtual machine verifies the proofs, inserts code to dynamically validate speculated invariants, and generates code specialized under the assumption that the speculated invariants hold. At runtime, the method parameters and other inputs are checked against the speculated invariants, and execution reverts to unoptimized code if the speculated invariants do not hold. The combined effect of the multiple phases is to shift the effort associated with bounds-checking array access to phases that are executed earlier and less frequently, thus, reducing runtime overhead. Experimental results show that this approach is able to eliminate more bounds checks than prior approaches with minimal overhead during JIT compilation. These results also show the contribution of each of the passes to the overall elimination. Furthermore, using our multiphase bounds check elimination method increased the speed at which the benchmarks executed by up to 16%.

1 Introduction

The semantics of the Java programming language require that out-of-bounds array accesses be caught at run-time [32]. This can be achieved by performing a runtime "bounds check" as part of each access to an array, but the overhead of doing so can be quite substantial [36]. This performance overhead is caused not only by the direct cost of conditional branches implementing the array bounds checks but also by lost opportunities for optimization and parallelization due to Java's precise exception semantics.

This overhead can be reduced, however, by applying a static analysis that can identify array element accesses that can never cause an out-of-bounds exception and

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
28 JAN 2010		**00-00-2010 to 00-00-2010**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Safe, Multiphase Bounds Check Elimination in Java	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Texas at San Antonio,Department of Computer Science,One UTSA Circle,San Antonio,TX,78249**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

As part of its type-safety regime, the Java semantics require precise exceptions at runtime when programs attempt out-of-bound array accesses. This paper describes a Java implementation that utilizes a multiphase approach to identifying safe array accesses. This approach reduces runtime overhead by spreading the out-of-bounds checking effort across three phases of compilation and execution production of mobile code from source code, JIT compilation in the virtual machine and application code execution. The code producer uses multiple passes (including common subexpression elimination, load elimination, induction variable substitution, speculation of dynamically-verified invariants, and inequality constraint analysis) to identify and prove redundancy of bounds checks. During class-loading and JIT compilation, the virtual machine verifies the proofs, inserts code to dynamically validate speculated invariants, and generates code specialized under the assumption that the speculated invariants hold. At runtime, the method parameters and other inputs are checked against the speculated invariants and execution reverts to unoptimized code if the speculated invariants do not hold. The combined effect of the multiple phases is to shift the effort associated with bounds-checking array access to phases that are executed earlier and less frequently, thus, reducing runtime overhead. Experimental results show that this approach is able to eliminate more bounds checks than prior approaches with minimal overhead during JIT compilation. These results also show the contribution of each of the passes to the overall elimination. Furthermore, using our multiphase bounds check elimination method increased the speed at which the benchmarks executed by up to 16%.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**42**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

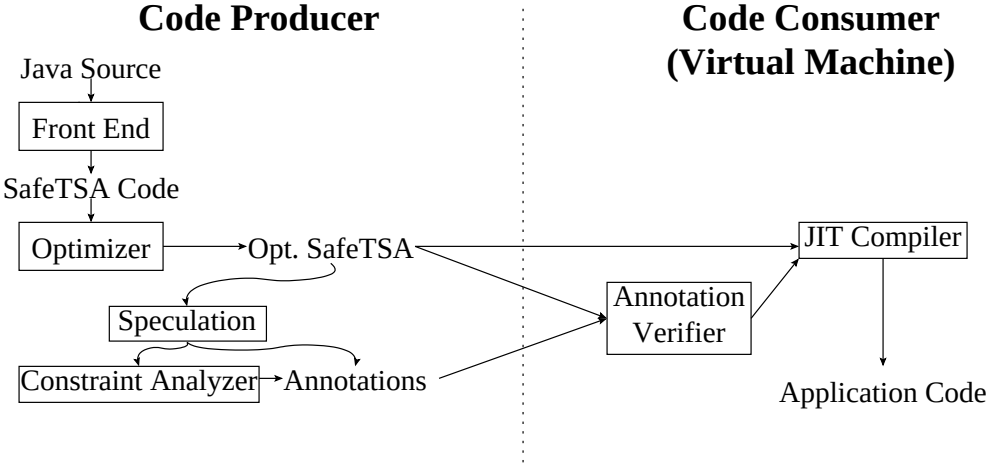

Figure 1: System Architecture

then optimizing the program by eliminating those unnecessary bounds checks. In a language like Java, though, such an approach must be compatible with just-in-time (JIT) compilation and other virtual machine features used to provide platform-independence, dynamic class-loading, and safety. Optimizing during JIT compilation constrains the time and scope available for analysis and thus limits bounds check eliminations to those that can be determined to be redundant through simple intraprocedural analyses.

This paper describes a system that utilizes multiple passes operating at different phases in the application code's life-cycle, including code production (preparation of the mobile code from source), JIT compilation-time, and method invocation-time, to cooperatively provide precise-exception semantics for out-of-bounds array accesses. The intuition behind this approach is that bounds checking effort is shifted to earlier and less performance critical points of time in the software life-cycle compared to pure runtime bounds checking. There are two major techniques that enable this phasing to be done safely. First, a proof-carrying-code-style linear-inequality constraint framework [56] allows the results of relatively expensive static analyses performed by the code producer (including a linear-inequality constraint analyzer and induction variable substitution analyzer) to be used safely by the time-constrained JIT compiler. Second, runtime support for speculative specialization provides a mechanism for including dynamic guards on sections of code and fall back to unoptimized code when those guards fail. This allows code to be aggressively optimized under the assumption that certain relationships hold among variables (which will be checked dynamically). As shall be shown in the experimental results section, this combination allows more bounds checks to be eliminated than was possible with prior methods based solely on verifiable-annotation or JIT-time static analysis.

1.1 System Overview

Our system is built on top of the SafeTSA system [6], which in turn is built on top of the Pizza compiler [44] and Jikes RVM [14]. The major components of the system architecture are shown in Figure 1. These can be divided between the components that process code once at the "code producer" (programmer, software, vendor, dis-

tributor) and those that process code at the time of program execution in the "code consumer" (the virtual machine). Within the virtual machine, we can distinguish between activities that happen statically (when the method is loaded or JIT-compiled) and those that happen dynamically each time a method is executed.

The code producer begins by compiling Java source code into an SSA-based intermediate representation. After compilation from Java source code into an SSA-based intermediate representation, several optimizations, including common subexpression elimination (CSE), are performed and the result is written to a SafeTSA file. Since bounds checks are represented explicitly, CSE is able to eliminate duplicate bounds checks [3, 55]. The optimized SafeTSA file is then processed by the "Speculator" and other preliminary analyses that derive a constraints system from the program code. The speculator searches the code for certain patterns for which it is expected that the redundancy of particular bounds checks can only be established through insertion of dynamic checks on program or method inputs. It then annotates these conditions as speculative invariants of the code. The Constraint Analysis System examines systems of linear inequalities derived from program code and speculated dynamic invariants, identifies whether these constraints are sufficient to entail the safety of particular bounds checks, and for those bounds checks, produces a proof tree showing that this is the case. These proof trees are used to produce verifiable bounds check elimination annotations proving those bounds checks are unnecessary. These annotations are transported along with the SafeTSA code to the code consumer.

In the code consumer, a bounds check elimination annotation verifier checks the proofs against the source code and speculated dynamic invariants. If the proofs verify, the JIT compiler inserts code for dynamic guards at method heads to check any speculative invariants and omits the code for the individual bounds checks which are proved unnecessary. The program will then run without executing the individual bounds checks. If a dynamic guard fails at runtime, however, an unoptimized version of the method will be executed instead.

1.2 Outline

In the following sections, we discuss the functioning of three major system components that contribute to array bounds check elimination: preliminary optimizations and analyses, the speculator, and the constraint analysis system. Issues related to consumer-side runtime system support are included with the discussion of the corresponding producer-side components. In Section 2, we discuss certain optimizations (specifically, common subexpression elimination, load elimination and induction variable substitution) that are performed to enable subsequent analysis and optimization. In Section 3 we describe the functioning of the Speculator component.[1] Section 4 describes the constraint analyzer[2] and how it is used to generate verifiable proofs of array access safety. Then in Section 5, we describe our experiments, which examine the contribution of different components of our system and compare our system's effectiveness with Array Bounds-Checks on Demand (ABCD) [11] and Chen and Kandemir's verifiable dataflow analysis [15] in terms of number of bounds checks eliminated, runtime overhead, and execution speedups for the Java Grande Forum Benchmarks.

[1] A preliminary description of this component was presented at PPPJ'08 [24].
[2] A preliminary description of this component was presented at SAS'09 [43].

2 Preliminary Optimizations and Analysis

Program code is not always in optimal form for bounds check elimination analysis. This can be, for example, because of indirection caused by design patterns or attempts at code optimization by the developer. The result is that relationships between variables may be hard to find and exploit. The preliminary optimization and analysis steps may modify the code into a semantically equivalent form, or perform an analysis that is hard to integrate in later steps. The goal in both cases is to give more information to the constraint analysis phase, so that more bounds checks can be potentially eliminated. Three such techniques are used in our system.

CSE. Common Subexpression Elimination removes redundant computations. The SafeTSA system includes both a global and local variant, implemented as a depth-first search on the program structure. The SafeTSA type system allows not only the removal of redundant arithmetic computation, but also of redundant null and bounds checks. The elimination incurs no cost at runtime, since the class loader uses the type system to verify the code while loading a class.

Although our system—as well as the other bounds check elimination techniques with which we will compare—include a more generalized form of bounds check elimination that can subsume the bounds checks CSE can eliminate, CSE simplifies the code at no cost for the consumer, so we adopted CSE-optimized code as the baseline for our comparisons.

Load Elimination. Object-oriented programming might lead to storing arrays and other values in fields. Using such values, thus, involves field access instructions. A naive, conservative analysis must assume a change of the content between two accesses, since Java is a multi-threaded language. This can often inhibit bounds check eliminations when the array is not accessed directly from a local variable (e.g., it is accessed from an Object field or an array-of-arrays) .

The memory model described in the Java language specification [26] allows optimized code to cache field accesses in the absence of explicit synchronization. This technique is called load elimination (LE).

Our multiphase bounds check elimination system includes a simple load elimination scheme on both the producer and consumer sides. The actual program code is not changed until execution. Instead, a flag is embedded in the annotations whenever load elimination is necessary for the verification to succeed. A verifier that is resource- or time-constrained may thus elect not to perform load elimination and discard any proofs that depend on it.

IVS. Hand-optimized code often transforms loops to be more efficient on the machine level. Instead of using a complex expression in the loop counter to index an array, helper variables are introduced that are independently updated in the loop. Those variables are called induction variables.

Induction variables do not have a direct dependency to the loop counter. Thus a simple analysis will not be able to compute the range of the variable and determine if the bounds check is redundant.

Induction Variable Substitution (IVS) is a technique to counter this problem. It automatically analyzes array index expressions, finds induction variables and computes a closed form in the loop counter for them, if possible. The closed form can than be tested for bounds check redundancy or other optimizations.

Many IVS algorithms with varying degrees of complexity (e.g., [2, 57, 10, 54]) have been invented with the more complex algorithms being able to express induction variables as polynomials [57, 28, 9]. While this power could be useful, our system currently restricts IVS to linear induction variables. This is the result of a conscious design decision: unlike typical induction variable substitution optimization passes, the actual program code is not transformed so that the induction variables are replaced by their closed form. Instead, the closed forms of induction variable are captured as linear inequality constraints on the induction variables, which can then be processed by the constraint analysis system that will be described in Section 4 and verified [56]. This is done for a couple reasons. First, the induction variables might have arisen as the result of intentional optimization (i.e., strength reduction). In such cases, it will often be desirable for the analysis to be able to identify and eliminate as many unnecessary bounds checks as possible without interfering with other optimization. A second concern is code size, replacing induction variables with complex index expressions will likely increase code size, which may be undesirable because of increased size increases network transmission time and memory usage. As a consequence of this design decision, all variable substitutions that are used to establish in-bounds array accesses must be able to be established based on linear inequality constraints, so IVS methods based on more general polynomials cannot be used.[3] Thus the actual implementation of IVS analysis in our system, is actually the result of cooperation between a traditional IVS-based analysis and the constraint analyzer that will be described in Section 4.

A simplified traditional analysis (derived from [57]) is used to find possible induction variables and suggest closed forms. In SSA form, induction variables have to be ϕ instructions, thus join points of loops are examined for candidates. In SafeTSA, loop headers generally have two in-edges: from the initialization before the loop and from the end of the loop. This simplifies the analysis. For each ϕ, the initial parameter and the loop parameter are evaluated symbolically to inequalities. The loop expression is then modified to take into account the start value, so that it is equivalent to the update of the induction variable over one loop iteration:

$V = I + k \times L \qquad V \ldots \text{variable}, I \ldots \text{initialization}, L \ldots \text{loop update}, k \ldots \text{loop counter}$

Simple IVS-like proposals are generated by pairwise comparison of the initial values and loop updates. Assume V_1 and V_2 are loop variables, I_1 and I_2 the initial expressions and L_1 and L_2 the loop updates. The following inequalities can be produced:

$$I_1 \leq I_2 \wedge L_1 \leq L_2 \quad \Rightarrow \quad V_1 - V_2 \leq 0$$
$$I_2 \leq I_1 \wedge L_2 \leq L_1 \quad \Rightarrow \quad V_2 - V_1 \leq 0$$

Traditional IVS is attempted when very simple variables are found: the loop updates are a multiple of each other. In this case, the two equations can be combined to

[3] On the other hand, though, bounding induction variables using inequality constraints does potentially provide some additional flexibility compared to substitution (which requires strict equalities in the closed form).

```
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);
}
```

Figure 2: Array bounds check with context-dependent redundancy.

eliminate the loop counter and still stay linear:

$$\begin{aligned} D_2 \equiv 0(D_1): &\quad V_2 - I_2 + (D_2/D_1) \times (I_1 - V_1) &= 0 \\ D_1 \equiv 0(D_2): &\quad V_1 - I_1 + (D_1/D_2) \times (I_2 - V_2) &= 0 \end{aligned}$$

After proposals are generated, they are passed to the constraint analysis system (Section 4). This may fail, if it cannot be established that there is no arithmetic underflow/overflow. If, however, the constraint analyzer is able to verify the proposed constraint, it will be recorded as an additional constraint that can be used in later phases.

Some additional techniques (including those in [57, 58, 51]) could be adapted to allow for a more precise analysis, detecting and wrapping more kinds of induction variables. In general, though, there is no way to record the results of those techniques precisely as linear constraints. It would, however, be possible to bound the values using linear constraints.

Flip-flop and periodic variables take their values from a ring of values. If all the possible values can be compared to each other, the flip-flop or periodic variables can be bounded by the smallest and the largest value in the ring, yielding two simple linear constraints of the form:

$$\begin{aligned} p - l &\leq 0 \\ s - p &\leq 0 \end{aligned} \qquad p \ldots \text{per. variable}, \; l \ldots \text{largest value}, \; s \ldots \text{smallest value}$$

Conditionally updated variables are not recognized as induction variables by traditional linear IV techniques. However, they often exhibit monotonic behavior, that is they are either always non-decreasing or non-increasing. Advanced techniques (e.g., [58, 51]) have been developed to exploit this, either by only establishing the general direction of a variable or by developing bounding functions. In our system, however, only a simplified version could be used, because the bounding results have to be expressible with linear inequalities.

3 Speculator

3.1 Speculative Optimization

Classical static bounds check analysis can be severely hindered by programming patterns that hide dependencies between program variables. An example of this can be seen in Figure 2. In the context of just method `f`, there is no relation between `n`

and `a.length`, which means that any system using only conservative, intraprocedural static analysis will be to unable safely remove the bounds checks.

There are a few approaches that could be used to optimize such bounds checks. If `g` is the only method calling `f`, interprocedural analysis could be used to 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, additional callers may be added at any time. Thus ensuring the validity of the analysis needs additional runtime overhead. (Inlining the method `f` into `g` would also expose the dependency, but still has complications associated with dynamic class loading and dynamic dispatch, since the inlining would only be valid if `f` is the only method that could be called by `g`.)

The approach used by our system, however, relies on a specialization of the code that does not rely on the context the method is called in, but rather 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 fragment, each specialized for different assumptions. A dynamic test can then be used to select the correct version at runtime.

Since these properties need not be based on a conservative analysis, the optimization based on them can be considered 'speculative' in the sense that the properties may not hold, and if the properties do not hold, then the specialized code may be invalid. Verifying that the speculative properties hold and falling back to the unspecialized version when they do not may have introduced overhead beyond what would have been incurred by simply using the unoptimized code. Hence, we will refer to these properties as 'speculations.'

Conceptually, such speculative specialization could be implemented as a pure producer-side (or even source-to-source) transformation that introduces an if statement condition that tests the speculation and performs the same code in both the 'then' and the 'else' arms allowing the 'then' arm to be optimized based on the knowledge that the speculation is true. In order to improve space efficiency, however, we introduced support in the consumer-side of our system for runtime-specialization.

This is accomplished by having the producer-side *speculator* augment the program representation with annotations that speculate that arbitrary linear inequalities hold at specific program locations. (The location at which a speculation is considered to be 'anchored' should be one that is dominated by all of the program variables that occur in the speculated linear inequality. The linear inequality can be assumed to hold at program locations dominated by the anchor point.) A runtime *specializer* pass occurs during JIT compilation, which inserts the dynamic check that ensure that code optimized under the speculative assumptions will only be executed when those speculations hold. The separation between speculator and specializer has several advantages. It allows each to evolve and be improved independently. In addition, since the specializer can work on the consumer side, after transmission, the bandwidth required for mobile code transmission is reduced. Furthermore, with appropriate runtime support, it is possible to avoid code duplication altogether (e.g., [59]).

The task of the producer-side speculator is to find and annotate profitable speculations. A general assumption underlying the heuristics used in our search for speculative constraints is that, in most programs, bounds check exceptions should be rare. Given this assumption, the safety constraints of an index of an array access could be used directly as a speculation, albeit a rather trivial one. If we simply replace the

```
1 void f(int a[], int x) {
2     int sum = 0;
3     for (int i = 0; i < x; i++)
4         sum = sum + a[i];
5 }
```

Figure 3: Java Method With A Loop-Invariant Constraint.

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 amortize the cost of one speculation across multiple static bounds check locations or one could amortize the cost of the speculation across multiple dynamic executions of the same bounds check. An example of the latter occurs if the speculation can be moved through loop-invariant code motion.

These consideration are realized using a two-stage process. The first stage analyzes every bounds check in isolation, trying to find the best speculation to remove that check. The second stage occurs after static analysis has shown how bounds checks can be eliminated, and consolidates combinable speculations and removes those speculations that are not likely to be profitable.

3.2 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, the examples in the following paragraphs are given in high-level code, while the framework itself uses SSA form.

Loop-invariant bounds checks. Consider the code fragment in Figure 3. 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 ϕ-function, so the constraint needs to be proven separately for each of the control flow edges. The proof corresponding to the first edge (when execution initially enters the loop and $i = 0$) is trivial. The second one requires that `i`$+1 \geq 0$. With the inductive assumption of `i`≥ 0, this is true. It can also be shown that there will be no arithmetic overflow. When `i`$+1$ gets computed, it is known that `i`$<$`x` (line 3) and `x`$\leq$`MAX_INT` (since x is an `int` variable), it follows that `i`$+1 \leq$`MAX_INT`, and there is no arithmetic overflow.

There is no similar proof for the upper bounds check. All that 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`$\leq$`a.length`, anchored at a point before the loop dominated by both `x` and `a`.

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

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

Figure 4: A method with coterminous arrays.

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

Figure 5: A method with rectangular arrays.

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 general linear inequality constraints built from linear expressions used for the induction variable's initial value, loop condition, and the array index. Note, however, that in such cases, one speculation will no longer suffice to prove that there will be no arithmetic overflow, so 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 speculator 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 less obvious that they should hold during program execution. 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. Loops commonly operate over multiple arrays, which are implicitly assumed to be of the same length. An example of such a loop operating on coterminous arrays is found in Figure 4. In this code, it is 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. Such a computation might look like the code shown in Figure 5. Many applications only use rectangular arrays, though Java allows multi-dimensional arrays to have any form.[4] 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 memory model, however, often allows memory reads to be moved as long as there is no intervening synchronizing 'acquire' event. In these cases, we may perform a variant of load elimination, which circumvents the problem by making a thread local copy of the outer array, and speculate on the lengths of the subarrays of this copy (which is now known to be invariant over the region of specialization). (A similar technique was used by Midkiff et al. [34].)

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.

It is noteworthy that the techniques previously described in the first speculator stage will probably already produce annotations for some instances of rectangular arrays. Unfortunately, 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.[5] 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 length of the shortest subarray 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.

When applied to the example above, this yields the speculation $v_a \geq$ `a[0].length`, where v_a is the virtual variable describing the length of the shortest sub-array 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. Each of these speculations can then be anchored at the earliest point it is valid (i.e., where all of its variables have been 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

[4] Cf. Moreira's [37] proposal for true multidimensional rectangular arrays.

[5] In fact, our prototype system only inserts speculations at the beginning of methods, and leaves the non-optimized code in a separate method that will remain uncompiled until and unless it is needed.

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

Figure 6: Bounds checks that could be consolidated

combined into one instruction, e.g. using an unsigned comparison instead of two signed ones.

3.3 Consolidation

After a whole method has been analyzed in the first stage, it is analyzed by the constraint analysis system, which will be described in Section 4. At that point every bounds check will either be statically proven to be redundant or will have a speculative proof. The task of the second speculator stage is to remove those speculations that are unlikely to be profitable 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 $c_1x_1 + c_2x_2 + \ldots + c_0 \leq 0$, and there is another speculation $c_1x_1 + c_2x_2 + \ldots + c'_0 \leq 0$ with $c'_0 > c_0$, then only the second speculation is needed, since it entails the first. This speculation needs to be scheduled at a program location that dominates both of the prior speculations. This ensures that any produced proof stays valid when switching the speculations. As an example consider the code fragment in Figure 6. The first stage of the speculator 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.

3.4 Precise Exceptions and Runtime dispatch

If implemented naively, speculation could potentially change program semantics. Specifically, Java's precise exception semantics require that the program state (or at least, the visible behavior of the program after the exception) appears as if all of the code up to the exception—but none of the code in the try block after the exception—was executed. So, implementing speculation by throwing an exception immediately whenever a speculation fails, would violate Java's precise exception semantics. Spe-

cialization 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 may be necessary to optimize for overlapping speculation ranges.

4 Static Analysis

After the program has been optimized, and the set of dynamically checked speculative constraints, IVS-constraints, and the set of load eliminations have been obtained, the program code is analyzed to develop a set of linear inequality constraints over extended SSA (eSSA) variables in each procedure that hold when the variables are in scope. These are then passed to the Constraint Analysis System (CAS), to determine whether these constraints are sufficient to prove that the program bounds checks are redundant.

CAS builds a *Constraint System (CS)* out of general linear relationships among program variables. These have the form:

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

The CS is populated with linear constraints known as *program constraints* that are known to hold because they are derived directly from the statements in the program. CAS then determines whether a *proposed inequality constraint* (a conjectured relationship among program variables) is consistent with those program constraints. That is, CAS attempts to determine if the proposed constraint creates a contradiction. CAS does so using negative logic: it tries to find sequences of program constraint combinations that are consistent with valid program control flow and which produce an inconsistent result (i.e., $c \leq 0$, where c is a positive constant) when combined with the proposed constraint. Constraints are combined via elementary row operations to produce an equivalent constraint with fewer variables. If CAS can determine that, regardless of the control path taken to reach the variables in a constraint, there is a corresponding set of program constraint combinations that can be combined with the proposed constraint to produce an inconsistency, then the proposed constraint can be rejected. Specifically, in the case where the rejected constraint states that an index is out of bounds, the array access using that index will be safe and its runtime bounds check can be eliminated. CAS records the sequence of constraint combinations from which an inconsistency was derived, enabling its reasoning to be later checked by verification systems.

4.1 eSSA Representation

Our system is able to simultaneously consider program control and data flow making meaningful conclusions about how program variables relate to one another, because the CS used by our system is derived from the program's eSSA [11] representation.

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

Figure 7: Example

```
read(a)
read(b)
if (a < b) then
   a1 = π(a)
   b1 = π(b)          /* a1 + 1 <= b1 */
   {Block A}          /* a,b replaced with a1, b1 */
else
   a2 = π(a)
   b2 = π(b)          /* b2 <= a2 */
   {Block B}          /* a,b replaced with a2, b2 */
end
a3 = φ(a1, a2)
b3 = φ(b1, b2)
{Rest of program} /* a,b replaced with a3, b3 */
```

Figure 8: eSSA Example

eSSA extends the traditional SSA [18] representation with "π-assignments", which reflect constraints resulting from taking particular control flow graph edges subsequent to conditional statements. These π-assignments create new variables (aliases) along control flow paths that are dominated by the outcome of a conditional expression. The π-variables referenced in constraints implicitly identify a particular set of control flow paths within the program, and can later be combined via an SSA ϕ-function at a control flow merge point. The code fragment in Figure 7 would be transformed into the equivalent eSSA fragment shown in Figure 8

4.2 Elementary Row Operations

CAS reduces the problem of determining whether a proposed constraint holds in the context of a program to deciding whether a system of linear inequalities is consistent. Several of the techniques used to solve this and related problems (such as Gaussian, Gauss-Jordan, and Fourier-Motzkin elimination) operate on the principle of iteratively reducing the original system to simpler but equivalent forms until it is able to determine consistency (or solutions). One of the fundamental concepts underlying such techniques is that of *elementary row operations*, which are transformations to the set of linear inequalities which do not change the solution set of the system. Techniques such as Gaussian and Gauss-Jordan elimination reduce a matrix representing a system of linear equations into an equivalent but simpler form by performing only the following row operations:

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

Row Multiplication A row can be multiplied by a positive constant

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

A related technique to determine the consistency of a system of linear inequalities via elementary row operations is Fourier-Motzkin Elimination (FME) [39, 20, 49]. FME applies elementary row operations to eliminate variables from the system until its consistency is readily decidable. It eliminates a variable by combining all upper bounds on a variable x with all lower bounds on x. Whenever a lower bound on x is paired with an upper bound on x, a new inequality constraint is produced in which x does not appear. After all variables have been eliminated, the system contains constraints of the form $c \leq 0$. If all such constraints are valid (that is, c is negative or zero), then the original system is consistent. Otherwise, the original system is inconsistent.

4.3 The CAS Constraint System

CAS's approach is based on FME: it eliminates a variable from all constraints in a single step. CAS constructs a *Constraint System* (CS) and combines constraints within it to reason about relationships among program variables. We now describe these two facets of CAS: representation via the CS and reasoning via constraint combination.

CAS builds and manipulates a *Constraint System* ($\mathbf{CS} = (\mathbf{V},\mathbf{C})$) to represent relationships among program variables. $\mathbf{V}$ contains representations of the variables

in the program, while **C** contains linear inequalities that represent constraints over those variables. These elements of **CS** are derived from the program's assignment and conditional statements. CAS continually reduces **CS** by eliminating variables in **V** until it is able to determine consistency. To eliminate a variable x_n in **V**, CAS combines each lower bound (LB) on x_n in **C** (i.e., constraints in which x_n's coefficient is negative) with every upper bound (UB) on x_n (i.e., constraints in which x_n's coefficient is positive) in **C** via elementary row operations that produce a zero coefficient for x_n in the result. At the conclusion of this step, x_n is removed from **V**, all new constraints formed by these combinations are added to **C**, and all former bounds on x_n are removed from **C**. Thus, as processing continues, the set **V** becomes progressively smaller, while **C** (potentially) becomes larger. Note that **C** does not increase if all constraints are difference constraints, whereas the elimination of variables involved in general linear constrains causes the number of constraints to increase.

4.3.1 Vertices in CS and their Properties

CS contains a vertex for each variable in the eSSA representation of the program (including π- and ϕ-result variables). Each vertex $v \in \mathbf{V}$ has several properties, including:

v.LB the set of constraints $e \in \mathbf{C}$ representing lower bounds on v.

v.UB the set of constraints $e \in \mathbf{C}$ representing upper bounds on v.

v.PHI a boolean indicating whether v is a ϕ-result variable

4.3.2 Constraints in CS and their Properties

The constraints in **C** represent linear constraints over the program's variables. CAS deals generally with two kinds of constraints: the constraints in **C** that are derived from program statements are called *program constraints*, and are known to be mutually consistent if they all lie on a non-cyclic feasible control flow path, whereas a *proposed constraint* represents a linear constraint over variables that CAS attempts to disprove. In our work, a proposed constraint represents an unsafe condition that would lead to an array-out-of-bounds exception (either upper or lower bound). Since the program constraints are self-consistent (arising directly from program logic), and since a proposed constraint states an access is unsafe (exceeds array bounds), an inconsistent system means that the array access is actually safe. In CAS all constraints represent less-than-or-equal relationships, so assignments and equalities are represented by a pair of inverted constraints. That is, if the assignment statement $x = y$ or equality statement $x == y$ occurs in the program, then two constraints are added to **C**: $x - y \leq 0$ and $y - x \leq 0$. In order to ensure combined constraints lie on consistent control-flow paths, constraints in **C** have a "direction" flag associated with them to distinguish whether the assigned variable has a negative coefficient ("forward" direction) or a positive coefficient ("reverse" direction). For example, the constraint $-x + y \leq 0$ resulting from the assignment statement $x = y$ is assigned a "forward" direction, since the variable being assigned to (i.e., x) is given a lower bound (has a negative coefficient). The other constraint resulting from this assignment ($x - y \leq 0$) is assigned a "reverse" direction since the assigned variable carries a

```
    int A[] = new int[y0];
    /* y0 == A.length */
    x0 = 0
L:  x1 = φ(x0, x3)
    if (x1 >= y0) goto E:
     x2 = π(x1)
     y1 = π(y0)
     /* x2 < y2 */
     A[x2] = ...
     x3 = x2 + 1
     goto L:
E:  x4 = π(x1)
    y2 = π(y0)
    /* y2 <= x4 */
          ...
```

Figure 9: eSSA Version

Constraint	Direction
$x_0 + 0 \leq 0$	Rev
$-x_0 + 0 \leq 0$	Fwd
$A.length - y_0 + 0 \leq 0$	Ind
$-A.length + y_0 + 0 \leq 0$	Ind
$x_0 - x_1 + 0 \leq 0$	Fwd
$-x_0 + x_1 + 0 \leq 0$	Rev
$x_2 - x_1 + 0 \leq 0$	Rev
$-x_2 + x_1 + 0 \leq 0$	Fwd
$y_1 - y_0 + 0 \leq 0$	Rev
$-y_1 + y_0 + 0 \leq 0$	Fwd
$x_2 - y_1 + 1 \leq 0$	Ind
$x_2 - x_3 + 1 + 0 \leq 0$	Fwd
$-x_2 + x_3 - 1 - 0 \leq 0$	Rev
$x_3 - x_1 + 0 \leq 0$	Fwd
$-x_3 + x_1 + 0 \leq 0$	Rev
$x_4 - x_1 + 0 \leq 0$	Rev
$-x_4 + x_1 + 0 \leq 0$	Fwd
$y_2 - y_0 + 0 \leq 0$	Rev
$-y_2 + y_0 + 0 \leq 0$	Fwd
$y_2 - x_4 + 0 \leq 0$	Ind

Figure 10: Constraint System (CS)

positive coefficient. Constraints arising from equality and inequality test operations are assigned an "independent" direction, but as we shall see later, if an "independent" constraint is combined with a constraint from an assignment, the new constraint will carry the direction of the assignment. Additionally, proposed constraints are initially assigned an "independent" direction as well. Another flag associated with a constraint is the *proposed* flag, which is a binary indication of whether a constraint is a program or proposed constraint. If a proposed constraint is combined with a non-proposed constraint, the *proposed* attribute is set to true on the result.

Any constraint in which x_n's coefficient is negative is a lower bound on x_n, and is added to the set $x_n.LB$, while those with positive coefficients for x_n are upper bounds and are added to $x_n.UB$. Since a single constraint can contain terms for multiple variables, the same constraint can simultaneously be in multiple UB or LB sets.

As an example, consider a simple program that allocates an integer array of y elements, and then assigns some value to each element in ascending order. The eSSA listing is shown in Figure 9, and the resulting **CS** is shown in Figure 10.

4.4 Constraint Combination in CAS

Once all program constraints and the proposed constraint are added to **CS**, CAS determines the plausibility of the proposed constraint by eliminating variables. Unlike conventional FME where each upper bound on a variable is indiscriminately combined with each lower bound on a variable to eliminate it from the system, CAS restricts

which constraints can be combined in order to ensure the resulting constraint corresponds to a valid control flow path in the program.

4.4.1 Direction Compatibility and Coupling Constraints

Some constraints, in particular, the 'coupling' of a ϕ-result variable with each of its operand variables, are only valid when certain control flow graph edges are traversed. An example of this can be seen in the case of the coupling constraints derived from the instruction $x_1 = \phi(x_0, x_3)$ in Figure 9: $x_1 - x_0 + 0 \leq 0$, $-x_1 + x_0 \leq 0$, $-x_1 + x_3 \leq 0$, and $x_1 - x_3 \leq 0$. If the lower and upper bounds on x_1 in these constraints were allowed to be combined arbitrarily, it would yield $0 \leq 0$, $0 \leq 0$, $x_0 - x_3 \leq 0$ and $-x_0 + x_3 \leq 0$. The first two of these constraints are consistent but redundant, whereas the second two are together equivalent to $x_0 == x_3$. If this were true, then the value of x_4 at the loop exit in Figure 9 would always be 0, but this relationship does not follow from the code, since the coupling between x_0 and x_1 and between x_1 and x_3 exist on different loop iterations.

CAS avoids this by flagging these 'ϕ-coupling' constraints (like $-x_1 + x_0 + 0 \leq 0$) where the ϕ-variable is negative as "forward" and those where the ϕ-variable is positive as "reverse." CAS then prohibits the combination of bounds with opposing directions. That is, CAS will never combine a forward constraint with a reverse constraint. If a forward or reverse constraint is combined with another constraint, then the resulting constraint will carry the forward or reverse direction, respectively. In this way, combined constraints derived from at least one non-independent constraint inherit the direction of the non-independent constraint, while a combined constraint is "independent" only if both "parent" constraints were "independent."

In the case of the ϕ-function in Figure 9, the constraint $-x_1 + x_0 \leq 0$ has forward direction and the constraint $x_1 - x_3 \leq 0$ has reverse direction. Combining them to eliminate x_1 is therefore not allowed. This will always be true for ϕ-coupling constraints. Since the ϕ-variable (x_1, in the example) is the variable being assigned, it will always have a coefficient of -1 in the forward ϕ-coupling constraints. Conversely, it will always have a coefficient of $+1$ in reverse ϕ-coupling constraints. For two constraints to be combined, they need to have opposite signs on the coefficient of the variable being eliminated, so because coupling constraints with opposite signs will also have opposite directions, the direction compatibility test effectively prevents CAS from using the ϕ-coupling constraints to inappropriately combine the constraints on the different ϕ operands. (They can, however, still be combined in the case of loops as described in the section of Sub-Cycle Elimination below).

In addition, direction flags on "π-coupling" constraints (such as $x_2 - x_1 + 0 \leq 0$) play a role in ensuring that constraints derived from conditional branches are only applied on the control flow path dependent on that conditional branch. An example of this is the constraint $x_2 - y_1 + 1 \leq 0$ in Figure 10 which is derived from the conditional branch in Figure 9. In general, constraints are presumed to hold wherever all of the variables involved in the constraint are in scope; eSSA π-variables are introduced for variables involved in conditionals, so that constraints for control-conditions are implicitly scoped to the control-dependent region. The direction rules ensure that a constraint obtained by combining condition constraints with constraints that are neither ϕ-coupling constraints nor proposed constraints will always contain at least one variable that is control dependent on the condition. As a result, those

variables can only be eliminated from the constraint if it is combined with a constraint derived from the coupling of a ϕ-function operand that is control-dependent on the condition or from an original proposed constraint that contained a variable that is control dependent on the condition; in either case, it is appropriate to include the condition-derived constraint. Otherwise, if it is inappropriate, the variable will be left unbounded and will not be able to be used to show an inconsistency.

4.4.2 Sub-Cycle Elimination

CAS detects (in)consistencies through the formation of inequalities in which all variables have been eliminated, leaving only the constant term (we refer to these as 'reduced' constraints). This can be viewed as a 'cycle' in which each term with a positive coefficient is paired with a term with a negative coefficient. In the example program (Figure 9 and 10), this would occur if the proposed constraint for the upper-bound being violated $-x_2 + A.length \leq 0$ is combined with $x_2 - y_1 + 1 \leq 0$ by eliminating x_2 to produce $A.length - y_1 + 1 \leq 0$ which is combined with $y_1 - y_0 + 0 \leq 0$ to produce $A.length - y_0 + 1 \leq 0$, which is combined with $-A.length + y_0 + 0 \leq 0$ to produce the inconsistency $1 \leq 0$. (Note how the final constraint completed the cycle $A.length \leftarrow x_2 \leftarrow y_1 \leftarrow A.length$ eliminating all of the variables.)

Smaller sub-cycles can also be formed if there are constraints derived from equality relationships (i.e., $x - y \leq 0$ and $y - x \leq 0$) which are not qualified with direction flags or in the case of loops. To handle the former situation, CAS keeps track of variables that were eliminated to derive each constraint and does not allow constraints to be combined if the same variable was eliminated from both sides of the inequality. The latter case is handled by eliminating ϕ-variables last and detecting cycles during ϕ-variable elimination. In such cases, CAS summarizes the loop under the assumption that the back edge is traversed either an infinite number of times, or else not taken at all, depending on which gives the most conservative (safest) summary for that constraint.

4.4.3 ϕ-nodes

CAS uses a map to compare multiple constraints with the same set of terms and propagates only the "best" one. The definition of "best" affects precision as well as safety decisions and depends upon whether the variable being eliminated is a ϕ-node. As described in [11], ϕ-nodes are "maximum" nodes, whereas other nodes are "minimum" nodes. We ensure that only the weakest constraints are propagated when a ϕ-node is eliminated, and the strongest constraint otherwise. To achieve this, CAS maintains a map effectively keyed by a set of terms (excluding a constant), combined with a direction. The value of the entry is the "best" constraint with those terms and direction. This map initially is populated by program and proposed constraints as they are added to the system. Thereafter, when a new constraint is produced via LB-UB combinations, the map is consulted to see if a 'better' value (depending on the type of eliminated variable) has already been produced. This is done by comparing the constant in the new constraint with the constant in the map entry corresponding to the new constraints terms and direction. If the new constraint and mapped constraint are of equal strength, a constraint carrying the "proposed" indicator takes preference. If not, the map is updated and the old constraint is deleted.

Mapping a reduced result requires explanation. Consider the combination of $z - y \leq 0$ (Forward) and $y - z \leq 0$ (Forward) to eliminate y. The reduced result ($0 \leq 0$) (Forward) has no terms, and the map key cannot directly be derived. In this case, CAS constructs a *key constraint* to be used as a map key. The key constraint is of the the form (α, *direction*), where α is the set of all terms in the LB, excluding the variable being eliminated, and *direction* is the direction of the reduced constraint. For example, suppose we wish to combine $3x + 4y - 2z \leq 0$ (Forward) with $-3x - 4y + 2z \leq 0$ (Forward) to eliminate z. Since the result is reduced, our key constraint is ($3x+4y$, Forward). Furthermore, if the constant in the reduced constraint is negative and composed from the combination of exclusively assignment statements, then the constant in the key constraint is set to $-\infty$ to reflect the fact that the loop is unsafe when taken in this direction and must be assumed to be taken an arbitrary number of times (since each iteration weakens the constraint). Alternatively, since taking a safe (incrementing) loop in a particular direction makes the constraint stronger (by incrementing the constant), we assume the loop is *never* executed by setting the constant in the the key constraint to the constant in the reduced constraint.

4.4.4 Unbound Variables

When a proposed LB is selected for pairing, a check is done to see if the variable being eliminated is unbound. If so, CAS conservatively reports that the proposed constraint holds. Equivalently, the unbound variable will remain after all other variables are eliminated, but this approach allows us to terminate the algorithm quickly.

4.4.5 Initialization

CAS processing begins by initializing an empty **CS**, and program constraints are added as the program is being parsed. CAS creates the internal representations of the constraint and referenced variables, adds new variables to **V**, sets the direction in the new constraint, and adds the new constraint to the appropriate variable's LB and UB sets, and adds the constraint to the map. Additionally, if the constraint arises from an assignment or equality, the constraint is copied, the coefficients and directions inverted in the copy, and the copy is added to UB and LB sets as well as the map. Once all the program constraints are added, a proposed constraint is formulated and passed to the `propose()` procedure. This routine creates and initializes a new constraint, sets its proposed flag, and adds it to appropriate UB and LB sets and the map. Next, assignments in the program are examined. For each assignment constraint where none of the variables are ϕ- or π-result variables, the direction of the constraint is changed to "independent." This enables CAS to infer relationships between x and y in code such as

```
x = 5;
y = 6;
```

where there is no intermediate variable to eliminate and direction incompatibility would otherwise prevent the necessary constraint combinations.

Procedure `propose(`*pc*`)`

```
/* Check if the proposed constraint pc is provably inconsistent
   with the program constraints in this CS                        */
proposed_inconsistencies = 0 ;
/* Add the proposed constraints to the UB and LB collections of
   the variables it references                                    */
foreach v in pc's terms do
   if v's coefficient is negative then
      Add pc to v.LB;
   else
      Add pc to v.UB;
   end
end
update_map(pc, null);
/* Convert assignments into inequalities where safe to do so    */
foreach non-φ v in V do
   if v is assigned in constraint c, c has no φs, and c is not a Pi-Assignment
   then
      c.direction = Independent;
   end
end
/* Eliminate all variables in the system.  We'll return early if
   we see a proposed consistency preventing us from disproving
   the proposed constraint                                        */
foreach non-φ v in CS do
   eliminate(v);
end
foreach φ v in CS do
   eliminate(v);
end
/* No proposed inconsistencies -- check if proposed
   inconsistencies detected                                       */
if proposed_inconsistencies > 0 then
   return FALSE;
else
   return TRUE;
end
```

Procedure `eliminate(`v`)`

```
foreach LB in v.LB do
    if LB.proposed AND v has no upper bounds then
        /* Unbound variable detected                                          */
        EXIT(TRUE);
    end
    foreach UB in v.UB do
        if LB and UB have compatible directions AND do not form a sub-cycle then
            /* Combine the constraints to eliminate v                         */
            new_con = combine(LB,v,UB);
            if new_con is reduced then
                /* All variables have been eliminated, leaving only a constant term.  If
                   this is a proposed constraint or is not a harmless cycle, then create
                   a key constraint from the eliminated terms (excluding v) so we can
                   consult the map                                            */
                if (new_con.constant < 0 AND new_constant.isAssignmentCycle) OR
                new_con.proposed then
                    calculate key_constraint from LB and UB;
                    if new_con.proposed then
                        /* Assume a proposed constraint is not strengthened by safe loop
                           traversals                                         */
                        key_constraint.constant = new_con.constant;
                    else
                        /* Assume unsafe loops are taken indefinitely         */
                        key_constraint.constant = −∞;
                    end
                    key_constraint.direction = new_con.direction;
                    update_map(key_constraint, v);
                end
            else
                /* There are more variables to eliminate -- see if this new constraint is
                   ``better'' than one we've already seen                     */
                if (update_map(new_con, v) == true then
                    Add new_con to the UB and LB sets of remaining variables;
                else
                    Discard new_con;
                end
            end
        end
    end
end
delete all bounds in v.UB and v.LB;
if map[v, *].constant ≤ 0 and is proposed then
    /* A consistency exists -- indicate we cannot disprove proposed constraint */
    EXIT(TRUE)
end
if map[v, *].constant > 0 and is proposed then
    /* Inconsistent proposed cycle -- increment counter and keep checking     */
    proposed_constraints += 1
end
```

Procedure update_map(*con, v*)

```
/* Manage a map of the ''best'' constraints seen thus far.  The
   definition of ''best'' depends on whether the term just
   eliminated to create a new constraint is a φ node or not     */
if map[(con.terms,con.direction)] exists then
   /* A constraint with the same terms and direction as the new
      constraint has already been processed.  See if the new
      constraint is ''better''                                  */
   old_con = map[(con.terms,con.direction)];
   /* If old_con is proposed and con is a program constraint (or
      vice versa) verify that the proposed constraint is not the
      original proposed constraint and equal to or weaker than the
      program constraint.  If so, return ''true''               */
   if v is non-φ then
      /* Test the constant in the old constraint to see if the new
         constraint is stronger.  Replace and delete the old
         constraint if it is weaker, or if it is not proposed
         while the new constraint is proposed (to be
         conservative).  Return ''true'' if we updated the map,
         ''false'' otherwise                                    */
   else
      /* Test the constant in the old constraint to see if the new
         constraint is weaker, or if the new constraint is
         proposed and is the same strength as the old constraint.
         If so, replace and delete the old constraint.  Return
         ''true'' if we updated the map, ''false'' otherwise    */
   end
else
   /* This is a new map entry                                   */
   map[(con.terms,con.direction)] = con;
   return "true";
end
```

4.4.6 Processing

After initializing the constraint system, `propose()` calls `eliminate()` to remove each variable from the constraint system. This method combines each LB on a variable with all compatible UBs on the variable, checks for unbound variables, and invokes the method `update_map()` to manage the map of "best" constraints. As part of the combination, a new direction is computed, and the parent UB, parent LB, and the eliminated variable are recorded in the result, so they can be used to generate a proof. Importantly, `update_map()` checks that the proposed constraint is not equal to or weaker than a program constraint that arises either through combinations or from the original set of constraints from the program. If that is the case, then `update_map()` immediately returns "true" to indicate the proposed constraint is consistent with the program constraints. If a cycle is detected, this is recorded by calling `update_map()` after first generating the appropriate key constraint as described previously.

CAS first eliminates all non-ϕ-result variables, leaving only the strongest constraint on each unique linear combination of ϕ-result variables. That is, if during elimination, there are two constraints $av_1 + bv_2 + cv_3 + \ldots + c_1 \leq 0$ and $av_1 + bv_2 + cv_3 + \ldots + c_2 \leq 0$ that arise during the elimination of non-ϕ-result variables, only the constraint with the largest constant value (c_1 or c_2) will be retained. Thereafter, the ϕ-result variables are eliminated retaining only the weakest constraint (i.e., the constraint with the smallest constant) for a given linear combination at each intermediate stage. In either case, the retained constraint is the strongest conservative constraint that CAS can derive for a given linear combination of variables. After CAS eliminates a variable, it checks the map with keys (v, Independent), (v, Forward), and (v, Reverse), which will yield the strongest conservative constraint of the form $c \leq 0$ that resulted from eliminating v. If c is non-positive, `eliminate()` immediately returns 'true' meaning that it might be possible for an out of bounds access to occur. Otherwise, if c is a positive constant, the bounds check violation is not possible on the corresponding control flow paths. In this case, a counter is updated to reflect the number of proposed inconsistencies encountered.

4.4.7 Termination

Once all variables have been eliminated without discovering a proposed consistency, `propose()` returns 'false' if at least one proposed inconsistency was discovered. Otherwise, `propose()` conservatively returns 'true.'

4.5 Generating Proofs

The core algorithm described above was extended to produce proofs while finding inconsistencies. As CAS eliminates variables to obtain derived constraints, it maintains a record of which source constraints were combined to produce the derived constraint. These records can be combined to produce a proof tree, rooted at the $c \leq 0$ inconsistency. For example, consider the Java fragment in Figure 11a. For this program, determining whether the upper bounds check is unnecessary involves finding an inconsistency involving a system of inequalities entailed by instruction semantics plus the proposed constraint that the upper bound is violated ($-y + a.length \leq 0$). While deriving the inconsistency $1 \leq 0$ for this system of inequalities, CAS would also build a tree of constraints leading to the inconsistency similar to the one show

```
1  x = 5;
2  a = new int[x];
3  y = 4;
4  a[y];
```

(a) source

$$\dfrac{\dfrac{\dfrac{}{-x+5\leq 0}\ ASGN_{LB}(1)\quad \dfrac{}{-a.length+x\leq 0}\ NEW(2)}{-a.length+5\leq 0}\ ADD \qquad \dfrac{\dfrac{}{y-4\leq 0}\ ASGN_{UB}(3)\quad \dfrac{}{-y+a.length\leq 0}\ PROPOSED}{a.length-4\leq 0}\ ADD}{1\leq 0}\ ADD$$

(b) CAS constraint inconsistency tree

$$\dfrac{\dfrac{\dfrac{}{-x+5\leq 0}\ ASGN_{LB}(1)\quad \dfrac{}{-a.length+x\leq 0}\ NEW(2)}{-a.length+5\leq 0}\ ADD \qquad \dfrac{}{y-4\leq 0}\ ASGN_{UB}(3)}{y-a.length+1\leq 0}\ ADD$$

(c) positive logic proof tree

Figure 11: Program with Proof Tree

in Figure 11b. Each inequality in the proof tree is justified by adding to inequalities together (ADD) or is derived from program source statements according to a rule specific to those source statements. After CAS returns its result, the constraint for the proposed "assumed violation" is removed, leaving a positive deductive proof that the upper bound of the array is respected by program execution similar to the one shown in Figure 11c. This proof tree can be used as a certificate that the check of the index does not exceed the array's upper bound is unnecessary.

In general, the rules for constraints derived from instructions may have premises that need to be satisfied. In particular, the constraints derived from arithmetic operations (e.g., the constraint $x - y - 1 \leq 0$ from the operation `x = y + 1`) require the establishment of the lack of arithmetic overflow (e.g., $y + 1 \leq$`MAX_INT`). In this case, CAS may be invoked iteratively to create additional proof trees to satisfy such premises. These trees can then be composed to produce a complete proof tree of the desired property.

These proofs are then encoded using the verifiable bounds check elimination annotation scheme that was described in von Ronne et al. [56]. In this scheme the annotations consist of *claims* and *proofs*. There are a fixed set of rules for instantiating claims for different types for operations. Claims are considered anchored to the instruction which they were instantiated from. For example, one rule allows the claim $x - y - 1 \leq 0$ at the location of the instruction implementing `x = y + 1`. Some of the rules require that proof obligations be discharged for the claim to be considered valid. In fact, the rule allowing $x - y - 1 \leq 0$ from `x = y + 1` includes an obligation for a proof that shows $y + 1 \leq$`MAX_INT`. Similarly, bounds checks that are determined by the producer-side system to be redundant must be claimed redundant by a rule. That rule imposes proof obligations requiring it to be shown that the index is less than the array length and greater than 0. Proofs are formed by combining claims (that are in scope) using addition and modus ponens. A complete description of this scheme, including more details on the rules and claim combination operators, can be found in von Ronne et al. [56].

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

4.6 Arithmetic Overflow

The algorithm described above finds symbolic solutions in the integer domain $\mathbb{Z}$. Java's integer type `int`, however, is restricted to integers representable in 2's complement 32-bit words and "wrap around" when operations underflow/overflow. Thus, the verification system requires supplementary proofs that the arithmetic operations from which constraints are drawn do not invalidate those constraints through arithmetic underflow or overflow. These are generated by examining the eliminations used by CAS, identifying the constraints derived from arithmetic operations, and invoking the CAS algorithm on a proposed constraint representing the underflow or overflow condition. Proving these constraints may, in turn, require further invocations to check for underflow or overflow in the arithmetic instructions providing constraints used in those proofs and so on.

5 Experimental Results

The components of our Multiphase Bounds Check Elimination method (MBCE) were implemented in the SafeTSA compiler and the SafeTSA virtual machine (VM). The SafeTSA VM is derived from Jikes RVM 2.2.0.

For the purpose of comparison, we also implemented Array Bounds Checks on Demand (ABCD) [11], a well-known fast JIT-time bounds check elimination algorithm, and Chen and Kandemir's [15] bounds check elimination method based on a verifiable data flow analysis (which we will henceforth refer to as VDF). Both methods had to be slightly extended to work with the SafeTSA data structures and features, most notably, the extended type system and its explicit type coercion instructions.

To evaluate the three approaches, we used the Java Grande Forum benchmarks [13]. The benchmarks were modified so that more array bounds could be eliminated by intraprocedural analysis[6]. They were then compiled into SafeTSA. During compilation to SafeTSA, common subexpression elimination was applied. Thus, the baseline to which all three methods were compared in the following experiments includes only the bounds checks that could not be eliminated by the SafeTSA CSE algorithm.

All of the runtime measurements were made while executing the benchmarks on a 1.5GHz G4 PowerMac with 1GB of RAM running a Linux 2.6.15 kernel. All bench-

[6]Symbolic constants were used for array bounds limits instead of passed in parameters where possible

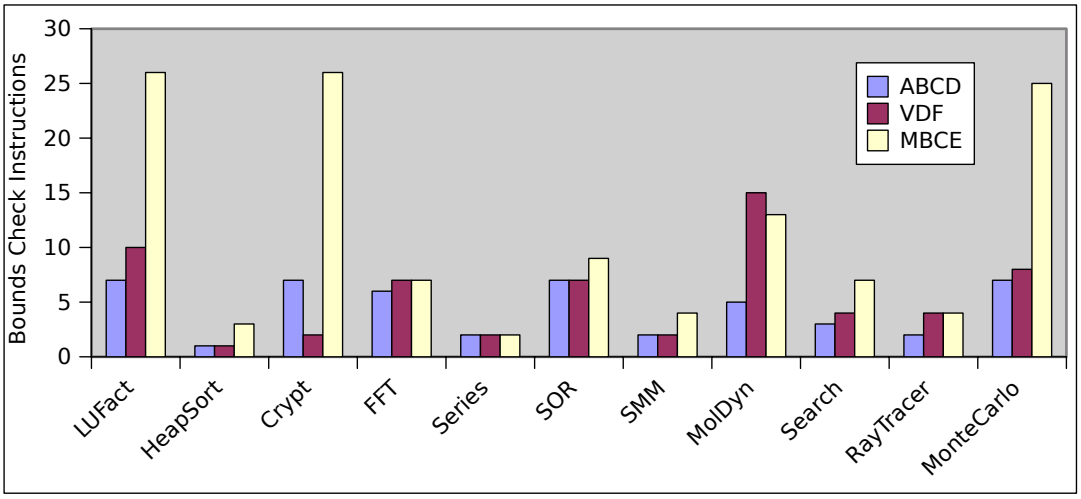

Figure 12: Eliminated Bounds Checks (Static)

marks were run with the large data size where possible. They were repeated at least 75 times, each iteration with a fresh virtual machine. The first 15 results were discarded, and any analysis was performed over the remaining runs.

5.1 Removed bounds checks

The number of bounds checks removed is an important metric when comparing bounds check elimination algorithms. Figure 12 shows, for each benchmark, the number of bounds checks removed by each elimination method

In all but one benchmark, our MBCE method removes all of the bounds checks removed by ABCD and VDF. In seven out of eleven of the benchmarks, it removes more bounds checks.

In the MolDyn (particle simulation) benchmark, both ABCD and VDF incorrectly remove the bounds check from code that looks similar to the following:

```
for (int i=0; i < a.length; i += 2) {
  a[i] = ...
}
```

In this code, both ABCD and VDF eliminate the bounds check for the store to `a[i]`. This is unsound, however, because if `a.length` is the maximum integer value (2,147,483,647), an arithmetic overflow would occur when two is add to `i`, changing it from `a.length-1` (2,147,483,646) to `a.length+1` (-2,147,483,648). When this occurs, the next store to a[i] will violate the lower bound of the array.[7] Thus, ABCD and VDF's failure to account for arithmetic overflow makes them unsound, and MBCE would subsume a sound variant of ABCD or VDF.

The better performance of MBCE can be explained with the successful mix of speculation, load elimination and rectangular array analysis, which enables less conservative bounds check analysis, and the more powerful analyzer, e.g., non-difference constraints in LUFact.

[7] In the actual benchmark execution, of course, this never happens, but it would require a much more sophisticated analysis than MBCE, ABCD, or VDF to determine this soundly.

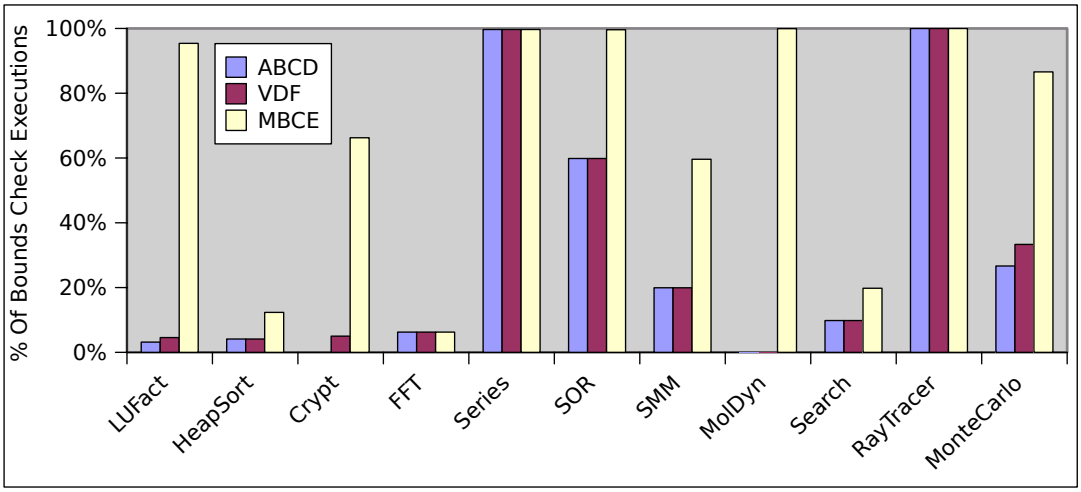

Figure 13: Eliminated Bounds Checks (Dynamic)

As a second set of results we extracted the dynamic number of bounds check executed during a benchmark run. This number better reflects the effect of the bounds check removal: the number of removed bounds check instructions weighs every instruction equally, whereas the dynamic count gives more importance to often-executed ones.

The SafeTSA class loader modifies the program code so that a counter is incremented 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. The difference between the dynamic count for the baseline and the count for each of the three methods is used as the number of bounds check eliminated. This number of bounds checks eliminated was then used to compute the percentage of dynamic bounds checks removed relative to the baseline. The results are shown in Figure 13.

The results differ from the static count. In most cases, they indicate that our speculations captured most or at least a significant number of the frequently executed bounds checks. Of special note are LUFact and MolDyn, where MBCE removes almost all checks and ABCD and VDF fail to make a significant impact.

Exceptions are Heapsort, FFT and Search, where most of the removed bounds checks are outside the hot path. HeapSort has a main loop that repeatedly doubles the loop counter. This would require a more powerful analysis than linear constraints allow. However, even then the code is not safe because of overflow issues and would need very involved speculations. FFT has a similar loop, as well as more complex index expressions. Search uses array elements as indices to other arrays. Since MBCE is purely intraprocedural, an analysis is impossible.

5.2 Influence of Different Optimization and Analysis Passes

The MBCE system consists of several different passes, including common subexpression elimination (CSE), our constraint analysis system (CAS), load elimination (LE), speculation (SPEC), and induction variable substitution (IVS). These are employed

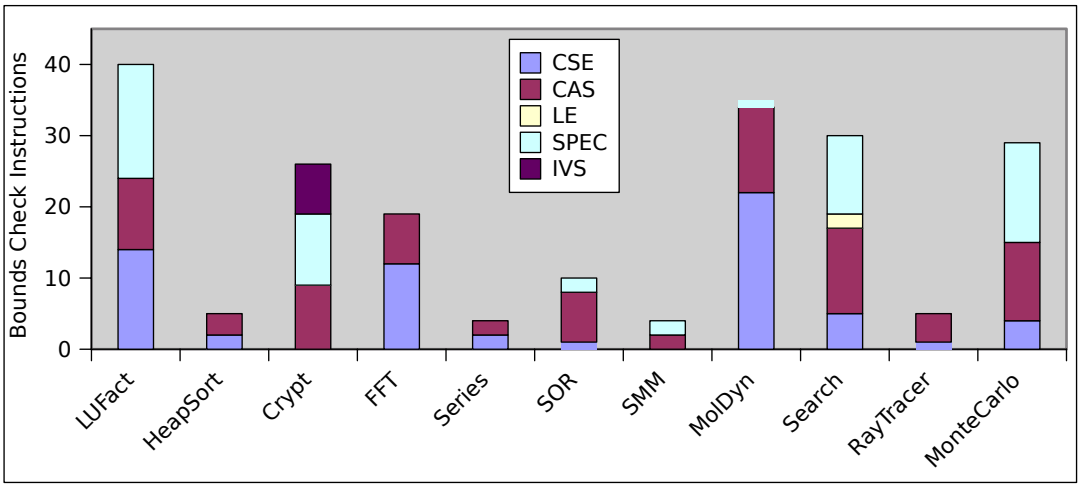

Figure 14: Static contributions

together to remove more bounds checks than any one could individually. It is interesting to evaluate which phases contribute how much to the final result. The cumulative contribution to the number of bounds checks eliminated from enabling each pass is shown in in Figure 14.[8] It should be noted that which phase bounds checks are attributable is somewhat order-dependent, because some bounds checks require a combination of phases before they can be eliminated, and others may be able to be independently eliminated by multiple phases.)

Note that common subexpression elimination is able to eliminate a significant number of bounds checks in several benchmarks, without any price at the consumer side. This is the case when an element of an array is repeatedly accessed. A good example is the particle simulation MolDyn. Each particle is an element of a large array. A particle update is a loop over all other particles, computing force influences on it. To compute a single force, the other particle is accessed nine times in a row. Common subexpression elimination is able to reduce this to only one access check per particle.

However, in most cases, the majority of the checks is removed by the rest of the system. In all cases, MBCE itself removes a significant part of the bounds checks, usually about a third to a half. In four benchmarks, namely LUFact, Crypt, Search and MonteCarlo, load elimination and speculation add another significant number. Note, that load elimination is usually an enabling factor for speculation and does not show up well by itself. In Crypt, induction variable substitution finally adds another significant number of eliminated bounds checks.

Note that IVS can be beneficial on its own. However, in our benchmarks the loop variable ranges over one array, while the induction variable indexes another. Coterminous loop speculation is thus necessary to remove the checks even after the induction variable has been analyzed.

In most cases, common subexpression elimination simplifies the program, but does not have a significant effect on the dynamic bounds check elimination numbers (as shown in Figure 15). Only FFT and MolDyn have the above-mentioned structure,

[8]The counts in Figure 14 are higher than those in Figure 12 because the counts in Figure 14 include the bounds checks due to CSE, which were already eliminated in the baseline of Figure 12.

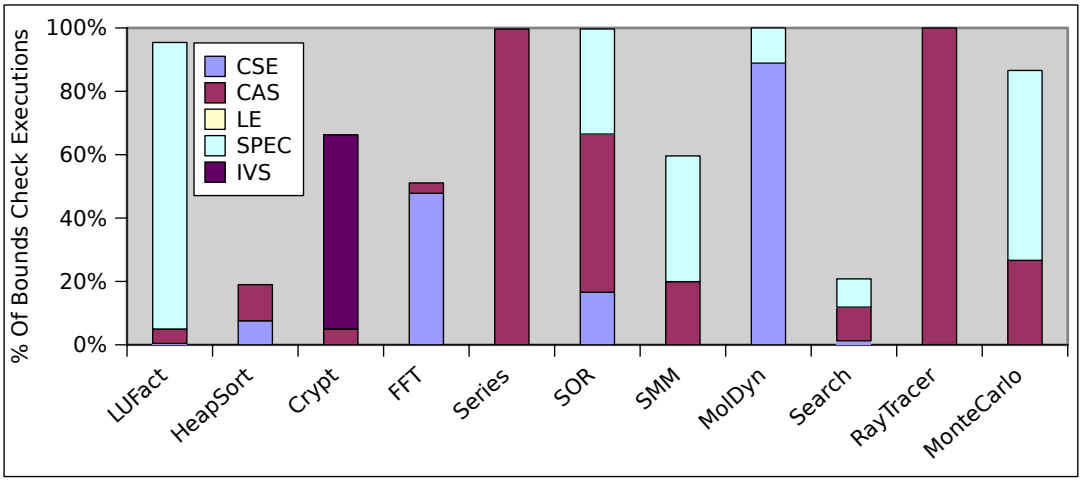

Figure 15: Dynamic contributions

and CSE is able to remove many bounds checks in the hot program path. The bounds checks removed in the other benchmarks typically reside in the initialization routine or outside of loops, so are not executed often enough to influence the statistics.

CAS and speculation are accountable for the majority of the eliminated bounds checks. In three benchmarks, namely Series, SOR and RayTracer, CAS adds the most significant amount, whereas speculation is dominant in LUFact, SparseMatMult and MonteCarlo.

IVS only plays a significant role in one of our benchmarks, Crypt. However, it is the single-most important factor, pushing the percentage of removed bounds checks from less than five percent to nearly seventy.

5.3 Just-in-time compile time

All three approaches (ABCD, VDF, and MBCE) perform some computation in the runtime system. In the case of ABCD, the whole analysis takes place at runtime. VDF performs a dataflow-based verification, while MBCE both verifies annotations and specializes program code at runtime. These actions result in some JIT compilation-time overhead. We examined the compile times reported by JikesRVM. The percentage of time spent in the bounds check elimination against the overall compile time (of the benchmark classes) is shown in figure 16.

In all cases, the elimination effort presents a minor impact on compile time, with a peak of 7.4% in LUFact, and an average of 2.4%. ABCD and VDF perform slightly better overall, with averages of 1.6% and 1%, respectively. Additionally, note that in all of our benchmarks, compile time makes up an insignificant part of the overall runtime. So 7.4% of compilation-time is less than one millionth of the entire execution time.

The overall impact of bounds check elimination for our system is higher than that of ABCD and VDF. However, this is compensated by the fact that MBCE removes a higher number of bounds checks in most cases. As the next section will show, this extra effort always at least pays for itself, and allows a significant speedup in several cases.

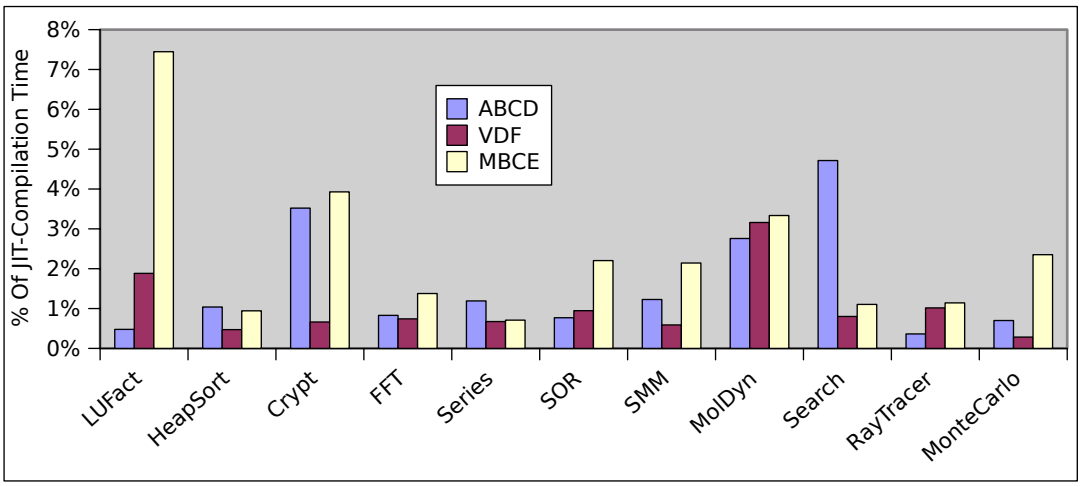

Figure 16: JIT-time Cost as Percentage of JIT compile time

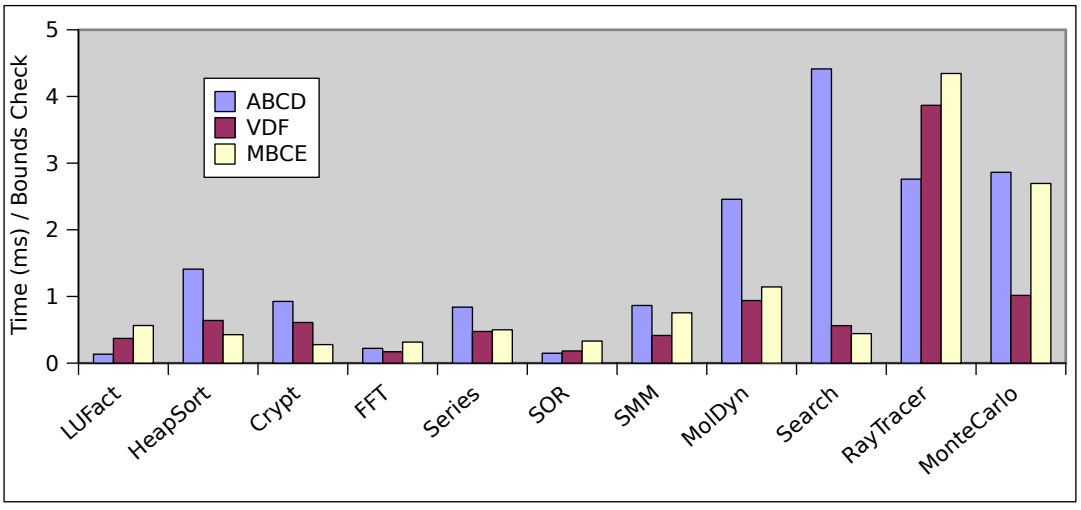

Figure 17: JIT-time overhead per eliminated bounds check

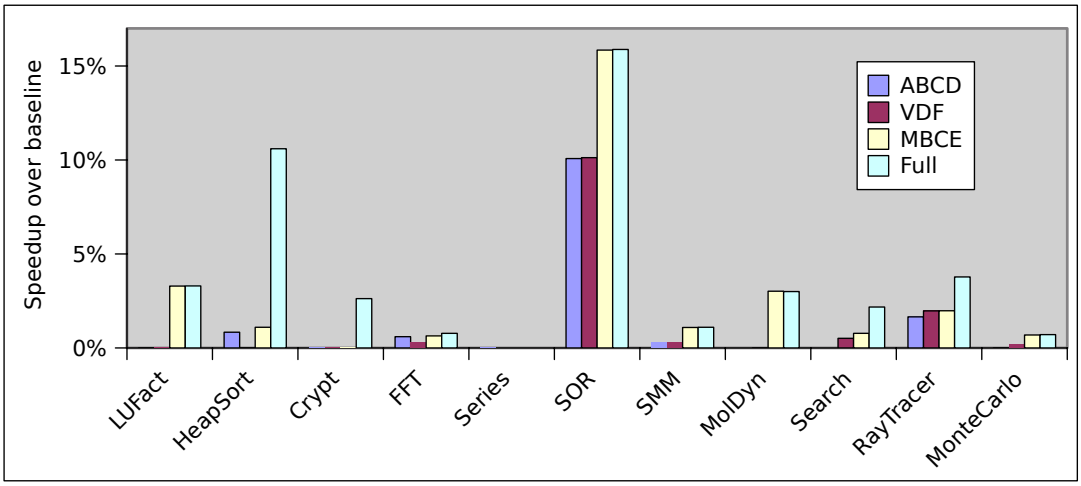

Figure 18: Runtime Speedup

As an alternative perspective, it is interesting to examine the average time per eliminated bounds check. This metric normalizes the overall time over the number of eliminated bounds checks, which equalizes time differences because of higher workload. With that, it turns out that ABCD is the most expensive with 1.35ms per bounds check, MBCE needs 1.07ms, and VDF 0.8ms (per-benchmark information is available in Figure 17). While ABCD is a very fast algorithm, it suffers from having to check every bounds check in the program code. VDF and our system can annotate which bounds checks can be removed and do not have to attempt an unsuccessful analysis. A prime example is the Search benchmark. Out of the 82 bounds checks, only three can be eliminated by ABCD (MBCE removes seven), but still has to analyze all of them, generating a significant overhead.

5.4 Runtime results

The resulting speedup of array bounds check elimination depends on the type and programming style of an 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, assuming that the runtime system supports the necessary 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. In general, of course, simply removing all bounds checks does not conform to the Java Language Specification, and can produce unsafe code. (In the absence of `ArrayOutOfBoundsException`'s, a program could freely access memory by accessing an array with an out-of-bounds index.) The benchmarks examined do not exhibit this behavior, since all array accesses are within the correct bounds, so are not affected. The resulting speedups are reported in Figure 18.

As expected from the elimination statistics, MBCE performs better than ABCD and VDF. In all cases, MBCE achieves at least as much of a speedup as the other systems. In eight out of eleven cases, MBCE performs better. In three cases, namely LUFact, SOR and MolDyn, MBCE performs significantly better, with a peak 16%

speedup in SOR.

Note that in almost all cases a good result in the dynamic bounds check removal statistics results in a near-optimal speedup. This is especially visible in the LUFact, SOR and MolDyn benchmarks.

Crypt shows that even a significant number of static removed bounds checks does not ensure a resulting speedup. A more complex specializer is necessary to achieve significant speedups here, since some analysis-enabling speculations had to be rejected by our simple implementation.

The Series benchmark has somewhat anomalous behavior. Despite MBCE being 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 as can be seen by the equally low speedup when all bounds checks are removed.

6 Related Work

Array bounds analysis has a long history. In 1977, Suzuki and Ishihata described a theorem prover that was able to derive the necessary loop invariants to prove certain assertions (such as, an index being within array bounds) [53]. The next year, Cousot and Halbwachs described a method for analyzing integer ranges using properties of convex polyhedrons to solve linear constraints; one of the intended applications of the analysis is array bounds checks [17]. More recent work on array bounds check analysis includes a symbolic bounds analysis for C programs using linear programming to solve a systems of linear inequality constraints [48], a bounds checker for Fortran programs [42], an abstract interpretation framework for certifying correct safe array accesses [8] and speculative bounds check elimination for Java [59].

Given the breadth of existing work on bounds check elimination, we will focus our discussion on those works related to our speculative analysis, and those related to our verification system, and existing systems for elimination of bounds checks in Java applications.

6.1 Program Specialization

Program specialization is a well-studied topic [50, and references therein]. DyC [27] and others use annotations to guide the specialization process, but rely on programmers to annotate source files. Calpa [35] 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.

Würthinger et al. [59] have developed a bounds check elimination mechanism for use in the HotSpot JIT compiler. It works by identifying simple patterns in the source code for which bound checks are unnecessary or for which speculation can be used. Their speculation operates in a manner similar to our approach, but their algorithm is intended as a runtime optimization and is thus less comprehensive then ours. It is only based on difference constraints, which reduces 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.

6.2 Analysis of Linear Inequality Constraints

Our constraint analysis system (CAS) algorithm can be described as a variant of the Fourier-Motzkin Elimination method [39, 20] for determining the satisfiability of systems of linear inequality. CAS extends Fourier-Motzkin elimination by (1) recording elimination steps to create a proof, and (2) incorporating special ϕ- and π-coupling constraints that allow constraints corresponding to different control flow paths and values variables take in different loop iterations to be combined into a single system of constraints over extended-SSA variables.

Fourier-Motzkin Elimination has previously been used in the context of array bounds check elimination in Xi and Pfenning's [60] work on dependent type systems. In their system, the type checking process generates a set of linear inequality constraints that are solved using Pugh and Wonnacott's variant of Fourier-Motzkin Elimination [45]. The type system requires program annotations (primarily, the signature of recursive functions, which would be the equivalent of loop invariants for imperative programs). This contrasts with our system, where cycle-reduction involving ϕ-coupling constraints serves to automatically identify appropriate loop invariants.

Systems of inequalities can also be framed as a linear programming and solved using other standard linear programming algorithms such as Simplex [19] or Karmarkar's interior points algorithm [30]. Cousot and Halbwachs [17] used a variant of the Simplex algorithm in an abstract interpretation to update a convex polyhedron (represented as linear equality and inequality constraints and as a frame) that denotes the set of combinations of values variables can take simultaneously at different program points. This analysis was used by Dor et al. [22] in the C String Static Verifier (CSSV) to detected buffer overflows in C programs. In contrast to this abstract interpretation approach which requires the solving of many linear programs to recalculate the polyhedra at different program points until a fixed point is reached, Ganapathy et al. [25] reduce buffer overflows detection to a single linear programming problem but, they do so by sacrificing soundness and flow-sensitivity. CAS, in contrast, is able to solve a single problem in a single constraint system that includes flow-sensitive information in the form of ϕ- and π-constraint directions. Rugina and Rinard [48]'s intraprocedural algorithm is also able to reduce the problem of bounding variable ranges to a singe system of constraints, specifically a linear programming problem over the coefficients of polynomials describing the bounds (rather than over the program variables), but this reduction requires that the variables be positive (or at least bound a priori by a known constant–an interprocedural positivity analysis is used for this). CAS does not have this restriction.

None of these systems using linear programming solvers have been applied to Java or optimization within JIT compilers. In this context, JIT-optimization-time is important, and the ability to efficiently verify CAS's proofs is important.

In the last few years, Satisfiability Modulo Theories (SMT) solvers have built on Satisfiability [21]. These extend techniques used for Boolean Satisfiability (SAT) solvers to automatically solve formulas consisting of conjunctions of terms from different theories. Two commonly supported theories that could be applied to bounds

check elimination are difference arithmetic and linear arithmetic. Most SMT solvers do not output a verifiable proof, but a few, including CVC3 [7], Fx7 [38], and veriT [12] can. In these systems, proof complexity and verification cost can be an issue sometimes even exceeding the cost of initially determining if the formula is satisfiable [38]. Even after applying rewriting techniques to reduce overhead, the verification times are measured in seconds [38] compared to milliseconds for verification in our system. Even with more efficient, verification, using SMT solvers formulating the bounds check problem as a a single formula would require finding the appropriate invariants externally (e.g., [52, 31]). In contrast, CAS is able to automatically find and prove appropriate loop invariants through its handling of cycle-reduction for ϕ-functions constraints.

6.3 Verifiable Bounds Check Elimination

The concept of annotating programs with proofs of various properties that could then be verified was explored as part of Necula's dissertation on Proof-Carrying Code (PCC) [41, 40]. Proof-carrying code was based on first-order logic, and is thus more general than the linear inequality framework used in our work [56]. The certifying compiler used with proof carrying code was capable of performing and proving the safety of several compiler optimizations including bounds check elimination [40]. The Special J compiler extended proof-carrying code to support the translation of Java programs into x86 assembly language [16].

We have taken some inspiration from proof-carrying code, but our work is more focused: array bounds checks rather than program type safety, and linear constraints of integers rather than first order logic. Our framework allows us to integrate with an existing Java virtual machine and continue to benefit from its machine-independence and dynamic class loading capability. The tighter focus makes our approach a direct replacement for runtime array bounds-check elimination. In addition, like the proposed virtual machine for proof-carrying code from Franz et al. [23], our approach should result in shorter, simpler proofs, and faster verification times than those based on a complete proof carrying code framework. Another difference is that our annotations are an optional addition to a type-safe mobile code format, and so unlike proof-carrying code, can be safely ignored by non-supporting virtual machines.

Other PCC approaches to verifying program analyses have also been designed based on iterative data flow analysis and abstract interpretation. Rose's work on Lightweight bytecode verification [47] transported the fixed point of the prior iterative bytecode verification algorithm as additional information in the bytecode file. Variations of that work were later adopted, first, as *stack maps* in the Sun's embedded system offering (the Kilobyte Virtual Machine of the Java 2 Micro edition) and, later, *split verification* in the Java 5 virtual machine. A generalization of this concept to other data flow analyses was described by Haldar [29], and Amme et al. [5] built a framework for transporting verifiable dataflow analysis results with SafeTSA programs. Chen and Kandemir [15] developed a method (referred to in the experimental results section as VDF), which utilizes the fixed point of an iterative dataflow analysis to split bounds check elimination between a producer and consumer phases, providing the same benefits of reduced overhead during JIT compilation as our system. Unlike our system, however, their system relies on a limited form of difference constraints (rather than general linear inequalities) and does not consider arithmetic overflow.

In a similar vein, Albert et al. [1] introduced Abstraction Carrying Code based on the CiaoPP abstract interpretation system for the Ciao Constraint Logic Programming System. The abstract interpreter takes a program and an abstract domain and produces a certificate containing an 'answer table' that approximates the abstract semantics of the program and analysis results as the fixed point of the abstract interpretation process. The code consumer is then able to check the correctness of this analysis by using a single-pass version of the abstract interpreter. The system then uses a trusted VCGen module to extract a verification condition that is then checked to see if the execution of the code does not violate the safety property. A foundational approach to proof-carrying code from certified abstract interpretation is taken by Besson et al. [8], who use the Coq Proof Assistant system to validate specialized certificate checkers against the formal operational semantics and safety policy of the program. This reduces the amount of code that needs to be trusted (and standardized) while still allowing for efficient verification of abstract interpretation over specialized abstract domains. Their system is also different in that it transports sparse "strategies" for reconstructing the analysis solution. Beson et al. [8] applied their framework to an interval analysis for a bytecode language. The bytecode language contains only a subset of the features found in standard Java bytecode, and so their analysis does not need to handle the complexities of identifying which array bounds checks can be optimized (it shows either that they are all safe or reports the program as unsafe), precise exception semantics, parameters, fields, or arithmetic overflow.

6.4 Static Bounds Check Elimination for Java

There have been several additional works addressing the array bounds check problems in Java.

Moreira et al. [36] used heavy-weight loop-based transformations and optimizations to optimize bounds checks in scientific applications; their goal was to provide a traditional static compiler for Java programs that provides performance approaching that of traditional optimizing compilers for Fortran, so their approach does not support just-in-time compilation and is not a general solution to the Java bounds check problem.

The ABCD algorithm [11] 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. The ABCD paper also discusses how the ABCD algorithm could be extended to partially redundant bounds checks by identifying sufficient conditions for the bounds check to be safe. This would be needed to be coupled with something like our speculation mechanism in order to correctly support Java's precise exception semantics. The ABCD algorithm implicitly assumes that variables are unbounded integers. Extending ABCD to check for arithmetic overflow would be possible, but may significantly impact the algorithm's efficiency. Menon et al. [33] extended the ABCD algorithm to produce optimized programs augmented with verifiable proof variables. The result is quite similar to our claims and proof obligations, but the verifier would be required to make judgements about facts (similar to our claims) using integer linear programming instead of checking an explicit proof; although verification performance was not reported by Menon et al. [33], we expect that integer linear programming would be slower than our approach.

Qian et al. [46] use an iterative dataflow analysis based on difference constraints to annotate bytecode with an indication of which bounds checks are unnecessary. Their framework also included an interprocedural array field and rectangular array analysis that would enable it to eliminate some of the same bounds that are enabled in our system through load elimination and rectangular array speculation. In contrast to our work, however, there is no mechanism to verify that the annotations are correct and remain correct during dynamic class loading, and thus the method is not compatible with standard Java Virtual Machine safety guarantees and functionality.

The elimination of bounds checks with producer-side CSE was possible because the base SafeTSA representation provides special types to facilitate the producer-side removal of duplicate bounds checks [55]. As seen in the experiment results of this article, a multiphase bounds check elimination method can improve precision significantly over those bounds checks that can be eliminated through CSE.

Several bounds check elimination techniques are based on identifying simple patterns of loops (that cover, e.g., the common case of a loop in which the loop variable is used as the array index and the loop condition compares that variable to the array's length). An example of such an algorithm is Zhao et al.'s [62], which is quite efficient during JIT compilation. Yessick and Jones [61] also proposed a method of bounds check elimination for restricted loop forms but it relies on annotations to reduce runtime overhead. Similarly, Amme and Gampe [4] developed a method for bounds check elimination on simple loop forms that can safely be applied during the production of an extended form SafeTSA. These techniques are subsumed by more general techniques including ABCD, VDF, or MBCE.

7 Conclusions

In this article, we presented a multiphase bounds check elimination method (MBCE) for the removal of array bounds check in Java. Our method consists of an annotator on the producer side that analyzes a program for redundant checks and annotates the program code with proofs certifying that fact, and a verifier on the consumer side, which uses the annotated proofs to verify the redundancy of bounds checks and finally removes them.

This architecture makes a shift of analysis time from the consumer to the producer possible, allowing us to implement multiple analysis steps that complement each other. The system applies five analysis and transformation passes during code production: Common subexpression elimination (CSE) simplifies the code at no cost for the consumer. Load elimination reduces the precision lost due to indirection in object-oriented designs. Induction variable substitution (IVS) analyzes optimized mathematical code. A speculator identifies potential runtime checks that increase the number bounds checks that can be eliminated. Finally, the Constraint Analysis Systems (CAS) uses the information derived in the previous steps, evaluates bounds checks for redundancy, and generates proofs that those checks are unnecessary. During loading and JIT compilation, the verifier checks the proofs generated by CAS and IVS, a code transformer performs load elimination, and the specializer instantiates multiple versions of methods guarded by different speculations. At runtime, specialized versions of methods are selected by dynamically evaluating the appropriate guards.

Each pass provides distinct benefits. CSE is cheap, but does not necessarily have a direct impact on runtime. The basic CAS algorithm, and the proofs it generates, form a baseline that performs as well as or better than ABCD [11] and VDF [15]. Load elimination, speculations, and IVS eliminate additional bounds checks at the cost of JIT-time overhead (from additional proofs and code transformations) and execution overhead (from the dynamic guards).

Experimental evaluation using the Java Grande Forum benchmark suite shows that the MBCE approach our can translate to significant speedups. (The SOR benchmark runs 16% faster than the baseline.) Despite a slightly higher JIT-time overhead, benchmark execution of the MBCE optimized code performed as well as or better than ABCD and VDF. For six benchmarks, optimization with the MBCE method system resulted in execution times significantly better than ABCD or VDF.

8 Acknowledgements

This research was supported in part by the Air Force Research Laboratory under grant F30602-02-1-0001 and the National Science Foundation under grants CCF-0846010, EIA-0117255, CCF-0702527, and CNS-0855247.

References

[1] Elvira Albert, Germán Puebla, and Manuel Hermenegildo. Abstraction-carrying code. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, volume 3452 of *Lecture Notes in Computer Science*, pages 380–397, 2005.

[2] Zahira Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 283–295, New York, NY, USA, 1990. ACM.

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

[4] Wolfram Amme and Andreas Gampe. Eliminating bound checks through ranges. Vordiplom Project, Informatik, Friedrich-Schiller-Universität Jena, 2005.

[5] Wolfram Amme, Marc-André Möller, and Philipp Adler. Data flow analysis as a general concept for the transport of verifiable program annotations. In *Proceedings of the 5th International Workshop on Compiler Optimization meets Compiler Verification (COCV 2006)*, volume 176(3) of *Electronic Notes in Theoretical Computer Science*, pages 97–108, 2007.

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

[7] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[8] Frédéric Besson, Thomas Jensen, and David Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theoretical Computer Science*, 364(3):273 – 291, 2006.

[9] Johnnie Birch, Robert van Engelen, and Kyle Gallivan. Value range analysis of conditionally updated variables and pointers. In *In proceedings of Compilers for Parallel Computing (CPC) (2004*, pages 265–276, 2004.

[10] William Blume and Rudolf Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *Proceedings of Supercomputing '94, Washington D.C*, pages 528–537, 1994.

[11] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 321–333, New York, NY, USA, 2000. ACM Press.

[12] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In *Automated Deduction CADE-22*, Lecture Notes in Computer Science, pages 151–156, 2009.

[13] 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.

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

[15] Guangyu Chen and Mahmut Kandemir. Verifiable annotations for embedded java environments. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 105–114, New York, NY, USA, 2005. ACM Press.

[16] Cristopher Colby, Peter Lee, George Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107, New York, NY, USA, 2000. ACM.

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

[18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[19] GB Dantzig. *Linear Programming and extensions*. Landmarks in Mathematics and Physics. Princeton University Presss, 1998.

[20] George B. Dantzig and B. Curtis Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A*, 14(3):288–297, 1973.

[21] Leonardo de Moura, Bruno Dutertre, and Natarajan Shanka. A tutorial on satisfiability modulo theories. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36, 2007.

[22] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, New York, NY, USA, 2003. ACM.

[23] Michael Franz, Deepak Chandra, Andreas Gal, Vivek Haldar, Fermín Reig, and Ning Wang. A portable virtual machine target for proof-carrying code. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 24–31, New York, NY, USA, 2003. ACM.

[24] Andreas Gampe, Jeffery von Ronne, David Niedzielski, and Kleanthis Psarris. Speculative improvements to verifiable bounds check elimination. In *Proceedings of the International Conference on Principles and Practice of Programming In Java (PPPJ 2008)*. ACM Press, 2008.

[25] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 345–354, New York, NY, USA, 2003. ACM.

[26] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition, 2005.

[27] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, October 2000.

[28] Mohammad Reza Haghighat. *Symbolic analysis for parallelizing compilers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1995.

[29] Vivek Haldar. Verifying data flow optimizations for just-in-time compilation. Technical Report 2002-118, Sun Labs, October 2002.

[30] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[31] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 119–134, 2005.

[32] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[33] Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, and Leaf Petersen. A verifiable ssa program representation for aggressive compiler optimization. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 397–408, New York, NY, USA, 2006. ACM Press.

[34] Samuel P. Midkiff, José E. Moreira, and Marc Snir. Optimizing array reference checking in java programs. *IBM Systems Journal*, 37(3):409–453, 1998.

[35] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 291–302, New York, NY, USA, 2000. ACM.

[36] José; E. Moreira, Samuel P. Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. *ACM Trans. Program. Lang. Syst.*, 22(2):265–295, 2000.

[37] José E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Marc Snir, and Richard D. Lawrence. Java programming for high-performance numerical computing. *IBM Sysmtems Journal*, 39(1):21–56, 2000.

[38] Michal Moskal. Rocket-fast proof checking for smt solvers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecutre Notes in Computer Science*, pages 486–500, 2008.

[39] Theodore S. Motzkin. *Beiträge zur Theorie der Linearen Ungleichungen*. Inaugural dissertation, University of Basel, 1936. Azriel: Jerusalem.

[40] George Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[41] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[42] Thi Viet Nga Nguyen and Francois Irigoin. Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst.*, 27(3):527–570, 2005.

[43] David Niedzielski, Jeffery von Ronne, Andreas Gampe, and Kleanthis Psarris. A verifiable, control flow aware constraint analyzer for bounds check elimination. In *Statis Analysis: Proceedings of the 16th International Static Analysis Symposium, SAS 2009*, volume 5673 of *Lecture Notes in Computer Science*, pages 137–153, August 2009.

[44] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'1997)*, ACM SIGPLAN Notices, pages 146–159, Paris, France, January 1997. ACM SIGACT and SIGPLAN, ACM Press.

[45] William Pugh and David Wonnacott. Eliminating false data dependences using the omega test. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 140–151, New York, NY, USA, 1992. ACM.

[46] Feng Qian, Laurie J. Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for java. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 325–342, London, UK, 2002. Springer-Verlag.

[47] Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

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

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

[50] Ajeet Shankar, S. Subramanya Sastry, Rastislav Bodík, and James 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, New York, NY, USA, 2005. ACM.

[51] Yixin Shou, Robert A. van Engelen, Johnnie Birch, and Kyle A. Gallivan. Toward efficient flow-sensitive induction variable analysis and dependence testing for loop optimization. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 1–6, New York, NY, USA, 2006. ACM.

[52] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Vs3: Smt solvers for program verification. In *21st International Conference on Computer Aided Verification (CAV 2009)*, 2009.

[53] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 132–143, New York, NY, USA, 1977. ACM Press.

[54] Robert van Engelen. Efficient symbolic analysis for optimizing compilers. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 118–132, London, UK, 2001. Springer-Verlag.

[55] Jeffery von Ronne, Wolfram Amme, and Michael Franz. An inherently type-safe ssa-based code format. Technical Report CS-TR-2006-004, Computer Science, The University of Texas at San Antonio, 2006.

[56] Jeffery von Ronne, Andreas Gampe, David Niedzielski, and Kleanthis Psarris. Safe bounds check annotations. *Concurrency and Computations: Practice and Experience*, 21(1), 2009. DOI: 10.1002/cpe.1341.

[57] Michael Wolfe. Beyond induction variables. *SIGPLAN Not.*, 27(7):162–174, 1992.

[58] Peng Wu, Albert Cohen, Jay Hoeflinger, and David Padua. Monotonic evolution: an alternative to induction variable substitution for dependence analysis. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 78–91, New York, NY, USA, 2001. ACM.

[59] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the java hotspot client compiler. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 125–133, New York, NY, USA, 2007. ACM.

[60] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, New York, NY, USA, 1998. ACM Press.

[61] Donald E. Yessick and Joel Jones. Removal of bounds checks in an annotation aware jvm. In *Proceedings of the IEEE SoutheastCon, 2002*, pages 226–228. IEEE, 2002.

[62] Jisheng Zhao, Ian Rogers, Chris Kirkham, and Ian Watson. Loop parallelisation for the jikes rvm. In *Proceedings of the Sixth International Conference on Parallel and Distributed Computing (PDCAT'05)*, pages 35–39. IEEE Computer Society, 2005.