A COMPREHENSIVE APPROACH TO ARRAY BOUNDS CHECK ELIMINATION FOR JAVA

by
Feng Qian

School of Computer Science
McGill University, Montreal

March 2001

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

The Java programming language requires array reference range checks at run time to guarantee a program's safe execution. If the array index exceeds the range, the run-time environment must throw an `IndexOutOfBoundsException` at the precise program point where the array reference occurs. Compilers generate conditional branch instructions for implementing array bounds checks. A branch instruction has great performance penalties in modern pipelined architectures. Also, it makes many other optimizations difficult. For array-intensive applications, array bounds checks may cause a heavy run-time overhead, and thus it is beneficial to eliminate all checks which a static analysis can prove to be unneeded. Array bounds checks are required by some other languages such as Ada and Fortran, and some bounds check elimination algorithms have been developed for these kinds of languages. However, these algorithms are not directly applicable for Java applications because of the precise-exception requirement of the language.

We present a new approach to eliminate array bounds checks in Java by using static analyses. Our approach is based upon a flow-sensitive intraprocedural analysis called *variable constraint analysis* (VCA). VCA collects constraints between locals related to array references. The array bounds check problem is formulated as solving a system of difference constraints. The analysis builds a small constraint graph for each important point in a method, and then computes the *shortest-path* weight of the graph. The shortest-path weights from upper bound to array index and from the index to lower bound indicates the safety of checks. Using VCA as the base analysis, we also show how two further analyses can improve the results of VCA. *Array field analysis* is applied on each class and provides information about some arrays stored in fields, while *rectangular array analysis* is an interprocedural analysis to approximate the shape of arrays, and is useful for finding rectangular (non-ragged) arrays.

We have implemented all three analyses using the Soot bytecode optimization/annotation framework and we transmit the results of the analysis to virtual machines using class file attributes. We have modified the Kaffe JIT, and IBM's High Performance Compiler for Java (HPCJ) [1] to make use of these attributes, and we demonstrate significant speed-ups.

[1] The experiment on HPCJ was conducted by Clark Verbrugge.

Résumé

Le langage Java vérifie les valeurs des indices de tableaux durant l'exécution pour garantir une exécution sûre. Si l'indice est supérieur à la taille du tableau, l'environnement d'exécution produit une exception `IndexOutOfBoundsException` à l'endroit précis du programme où l'indice de tableau fautif apparaît. Les compilateurs génèrent des instructions de branchements conditionnels pour implémenter cette vérification. Une instruction de branchement est très pénalisante dans les architectures en pipeline modernes, et rend difficiles beaucoup d'autres optimisations. Pour les applications qui utilisent beaucoup de tableaux, la vérification des limites de tableaux peut causer une importante augmentation du temps d'exécution, et il serait donc bénéfique d'éliminer toutes les vérifications qu'une analyse statique révélerait inutiles. Les vérifications de limites de tableaux sont nécessaires pour certains langages comme Ada et Fortran, et des algorithmes d'élimination ont été développés pour ceux-ci. Or ces algorithmes ne sont pas directement applicables à Java de par la présence du mécanisme d'exceptions du langage.

Nous présentons une nouvelle approche pour éliminer les vérifications de limites de tableaux en Java par des analyses statiques. Notre approche est basée sur une analyse intraprocédurale et "flow-sensitive" appellée *analyse à contraintes variables* (VCA). La VCA collecte les contraintes entre variables locales liées aux indices de tableaux. Le problème des vérifications de limites de tableaux est formulé comme un système de différence de potentiels. L'analyse construit un petit graphe de contraintes pour chaque point important de la méthode et calcule la valeur du plus court chemin du graphe. Les valeurs des plus courts chemins de la limite supérieure à la valeur de l'indice et de l'indice à la limite inférieure indiquent l'utilité de la vérification. En utilisant VCA comme analyse de base, nous montrons aussi comment deux analyses plus poussées peuvent améliorer les résultats. L'*analyse des champs tableaux* est appliquée sur chaque classe et fournit des informations sur certains tableaux utilisés

dans les champs, tandis que l'*analyse de tableaux rectangulaires* est une analyse interprocédurale d'approximation de la forme des tableaux multi-dimensionnels, qui est utile pour trouver les tableaux rectangulaires.

Ces trois analyses ont été implémentées avec la structure d'optimisation et d'annotation Soot grâce à laquelle nous transmettons les résultats de nos analyses aux machines virtuelles Java par le biais des attributs des fichiers classes. Nous avons modifié le JIT de Kaffe, ainsi que le High Performance Compiler for Java (HPCJ) d'IBM [2] pour utiliser ces attributs et nous montrons les améliorations significatives qui en résultent.

[2]L'expérience sur HPCJ a été réalisée par Clark Verbrugge.

Acknowledgements

This thesis would not be written without the help from others. First of all, I would like to give special thanks to my advisor, Laurie Hendren, who led me to the fantastic world of compilers. Not only she gave great help in academic research, but also provided personal support and consistant encouragement. Her cheerful nature and enthusiasm make the group full of fun. I would also like to thank Karel Driesen for teaching me many things about computer architectures and object-oriented langauges.

Special thanks go to the authors of Soot on which the experimental part of thesis is based. Raja Vallée-Rai designed and implemented beautiful JIMPLE IR and APIs which make the implementation much easier. Many useful tool classes designed by Patrick Lam are bases of my analysis. I also learned many OO design patterns from their code. Patrice Pominville's work on class file annotation makes the experiment on real Java virtual machine possible. I would also like to thank other members in Sable Research Group. Etienne Gagnon, his advise on career is invaluable. Jerome Miecznikowski, Fabien Deschodt, and Felix Kwok, we got so much fun from many talks we had. I also appreciated Jerome's proofreading of the thesis, and Fabien's help on translating the abstract to the french version.

Finally, I would like to thank my family members, my parents and my uncle, who support me at all time. This thesis is also a special gift to my wife, Beibei, who shared my happiness and sadness every day.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Java programming language is becoming increasingly popular for the implementation of a wide variety of application programs, including loop-intensive programs that use arrays. Java offers many desirable features such as object-oriented software design, cross-platform portability, safe execution, and many support class libraries. By programming in Java, a programmer can increase productivity while writing safe code. Also the program can be written once and run everywhere. These attractive features, however, cause performance penalties. The object-oriented feature relies on virtual method calls; the cross-platform portability is accomplished by interpreting and/or just-in-time compiling bytecode; and the safety is secured by various compiler and run-time checks, e.g, class file verification, array bounds checks, null pointer checks, and type checks. Because of these expensive operations, a Java program usually is much slower than its counterpart in C/C++.

A Java program is compiled to a class file in bytecode format. The bytecode class file is executed by a virtual machine (VM). The Java programming language has its own specification [10], which defines the syntax and semantics of the language. The Java virtual machine specification [17] defines the bytecode format and the run-time support environment. The bytecode class file can be executed in several ways. In an internet environment, the class file is loaded and executed by a virtual machine. The VM can interpret the bytecode, or use a Just-In-Time compiler to translate the bytecode to native code and execute it by hardware directly. In this case, the interpretation and/or compilation time contributes to the total execution time of the program. In other fields, such as scientific computations and real-time applications,

all the class files of the application can be compiled to native code by an Ahead-Of-Time (AOT) compiler before execution. In this case, the compilation time can usually be ignored.

To speed up the execution of Java programs, a general approach is to build a sophisticated virtual machine, which includes a class file loader and verifier, an interpreter and/or JIT compiler(s), and a garbage collector. A naive JIT compiler [15] translates bytecode to native code without performing many optimizations (it may perform some simple optimizations within basic blocks). Sophisticated JIT compilers [5, 30, 1, 11] apply traditional and adaptive optimizations on the process of translation. It has been proved that Just-In-Time compilation makes the execution of Java programs much faster than interpretation. Because the compilation time accounts a part of the program execution time, a JIT compiler can not afford many advanced optimizations which are usually expensive.

Another approach to improve the performance of Java programs is to optimize the bytecode and perform relatively expensive analyses statically. The optimizations can target either space reduction, which removes unused fields and methods from class files, or performance improvement. Many traditional analyses can be applied to bytecode and produce good-quality bytecode class files. Such optimizations include common subexpression elimination, deadcode removal, static inlining, and so on. Another group of analysis results cannot be reflected by transforming bytecode directly, for example, array bounds check elimination, type check removal, and stack object allocation. But these analysis results can be used by a virtual machine or an Ahead-Of-Time compiler (the optimizations can be built in AOT compilers). The analyses are not limited in compilation from bytecode to native code, they can also improve memory management, task organization, and so on. This approach moves the performance burden from running time to static compilation time, and allows us to optimize the class file once for reuse by many VMs at any time.

The focus of this thesis is on reducing the run-time overhead caused by array bounds check instructions (and partially null pointer check instructions). We are using static analyses to analyze Java applications at the bytecode level. The results are encoded in the class file as attributes. A JIT or AOT compiler understands the attributes and removes the bounds check instructions which are marked as unnecessary. The algorithm can also be implemented in an AOT compiler. Although the algorithm was developed for Java, it also can be implemented in compilers for other imperative languages which require array bounds checks.

The rest of this chapter is organized as follows. Section 1.1 introduces the problem of array bounds check elimination in Java. Section 1.2 describes the framework on which our analyses are implemented. Thesis contribution and organization are presented in Sections 1.3 and 1.4.

1.1 Array bounds checks in Java: the problem

In languages like C, a major source of potential errors is illegal memory accesses. For example, writing to the region outside of an array can cause unanticipated consequences. Java provides secure and safe execution of programs. As part of the safety system, array bounds checks are used to detect memory violations due to illegal array accesses. The Java language specification requires that an exception has to be raised for any array access in which the array index expression evaluates to be out of bounds. Figure 1.1 gives several examples that raise `IndexOutOfBoundsExceptions`. In addition to the `IndexOutOfBoundsException` exception, an array reference will throw a `NullPointerException` if the array object is `null`, and the virtual machine will not check the array bounds. The Java language specification also requires that the exception has to be thrown at the precise point where the exception happens because user code can catch such exceptions or dump stack traces for debugging purposes. Execution of an array reference bytecode (e.g, `iaload, istore`) needs a null pointer check first, and then checks of both lower and upper bounds. The lower bound of an array reference is fixed to the constant `0`, and the upper bound is 1 less than the array length stored in the array object. Both lower and upper bounds checks must be satisfied. The exceptions for lower and upper bounds checks are the same.

```
     int a = new int[10];

(1)  a[-1] = ...;  // lower bound out of range

(2)  a[10] = ...;  // upper bound out of range

(3)  for (i=0; i<=a.length; i++)
       a[i] ... ;   // upper bound out of range
```

Figure 1.1: IndexOutOfBoundsException examples

A direct implementation of checks for one array reference adds three conditional branch instructions: 1) if the address of the array object equals zero, branch to a routine raising a null pointer exception, 2) if the index is less than zero, raise an array bounds out of range exception, and 3) after reading in the array length, if the index is greater than the array length minus 1, raise an array bounds out of range exception. Some well-known techniques can reduce three branch instructions to one in most of modern architectures (e.g, x86, PPC). The null pointer check does not need an explicit check instruction when the hardware is capable of catching memory accesses to the first page (page address starting from zero). Usually the array length field is located near the object head. Thus, reading in the field from a null object would cause a hardware trap and the trap handler would raise a null pointer exception. Lower and upper bounds checks can be implemented by one unsigned comparison instruction because any negative integer is greater than any positive one when it is treated as an unsigned integer.

Although we can use the above techniques to reduce the cost of checks, at least one conditional branch instruction is still needed for each array access. A naive JIT or AOT compiler inserts checks for each array access, which is clearly inefficient. These checks cause a program to execute slower due to both direct and indirect effects of the bounds check. The direct effect is that the bounds check is usually implemented via comparison and branch instructions, and thus each array access has this additional overhead. The indirect effect is that these checks also limit further optimizations because the Java virtual machine specification requires precise exception handling. This limits code movement and also limits many effective loop transformations which are commonly used in high-performance C and Fortran compilers [21]. Furthermore, this same precise exception requirement limits program transformations that optimize the run-time checks. For example, checks cannot be moved to earlier program points if this changes the exception behavior of the program.

The problem of eliminating array bounds checks has been studied for other languages and static analyses have been shown to be quite successful[12, 13, 16]. However, array bounds check analysis in Java presents several special challenges. Firstly, the length of an array is determined dynamically, when the array is allocated, and thus the length (or upper bound) of the array may not be a known constant. Secondly, arrays in Java are objects, and these objects may be passed as references through method calls, or may be stored as a field of some objects. Thus, there may be a non-obvious correspondence between the allocation site of an array and the accesses

to the array. Thirdly, multidimensional arrays in Java are not necessarily rectangular, and so reasoning about the lengths of higher dimensions is not simple. Finally, techniques that require transforming the program or inserting checks at other earlier program points are not as applicable in Java as in other languages with less strict semantics about exceptions.

Figure 1.2(a) shows a piece of code which needs two checks for two array references. Some well-known algorithms[12, 13, 16] can merge two checks to one as in 1.2(b). Although the change reduces two checks to one, the new code does not have same exception behavior as original one. Consider that the length of the array is 4. In (a) the exception is raised before the second array access `a[5]`, and in (b) the exception happens before the first reference. The problem is that a user may write a try-catch clause to catch the exception and do some recover work. The catch statement would get different value of `i` for the two different cases. The second treatment violates the precise exception requirement of the Java language.

```
int a = new int[k];

; if a.length <= i
;    raise exception
a[i]... ;

; if a.length <= i+1
;    raise exception
a[i+1]... ;
```

(a) original checks

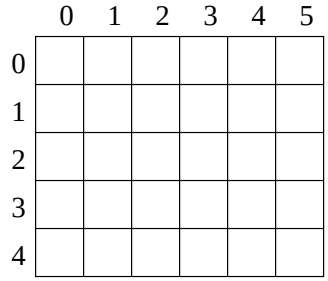
```
int a = new int[k];

; if a.length <= i+1
;    raise exception
a[i]... ;

a[i+1]... ;
```

(b) merged check

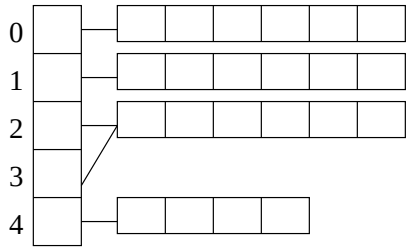
Figure 1.2: A precise-exception example

Multidimensional arrays are the most common data structures in scientific computation. Vectors and matrices in linear algebra are represented as one- and two-dimensional arrays (we have used a few in our benchmarks). To make Java as competitive as C and Fortran, operations on multidimensional arrays must be performed efficiently. In C or other languages, a two-dimensional array is allocated in a contiguous memory block as in Figure 1.3(a). However, Java defines a multidimensional array as an array of arrays. See Figure 1.3(b) which is a legal array shape in

Java. Sub-arrays are independent and can have different lengths. To deal with this, a reference to the second dimension in source code is implemented in bytecode by two array references, as in Figure 1.4. The bytecode instruction set provides only one-dimensional array access and accesses to multidimensional arrays are performed one dimension at a time. This definition makes the multidimensional array be a very loose structure, and the sub-arrays may not all be the same length, or sub-arrays may be references to the same array object (aliased), or they could even be null.



(a) A two-dimensional array in C



(b) A two-dimensional array in Java

Figure 1.3: Compare multidimensional array shapes

1.1.1 Eliminating unnecessary array bounds checks in Java

This thesis describes a flow-sensitive, intraprocedural algorithm called *variable constraint analysis* (`VCA` for short) that can prove that many array references are safe, without transforming the original program. The algorithm collects differences constraints, and builds a constraint graph for each array reference. Then it uses the graph to infer the relationship between the index of the array reference and the array's length. The algorithm was designed carefully to take advantage of the fact that

```
int[][] I = new int[10][10];

I[2][3] = 10;
```

a) Java source code

```
bipush            10
bipush            10
multianewarray    <[[I>   2 (2)
astore_1
aload_1
iconst_2
aaload
iconst_3
bipush            10
iastore
```

b) Bytecode

```
$r1 = multinewarray int[10][10];
$r2 = $r1[2];
$r2[3] = 10;
```

c) More readable JIMPLE code

Figure 1.4: A reference to two-dimensional array in Java

variables used in index expressions often have very short lifetimes, and thus building graphs for only live variables of interest leads to very small graphs. Further, we tuned the worklist algorithm to reduce the number of iterations. As a result, the actual running time is linear in the size of the method being analyzed.

We have improved the base VCA algorithm using two additional analyses: *array field analysis* is applied to each class and provides information about some arrays stored in fields, while *rectangular array analysis* is an interprocedural analysis based on call graphs to approximate the shapes of arrays.

Java is a class-based object-oriented language. Each class can declare fields, and each field has a modifier which defines the access privilege to it. *Array field analysis*

takes advantage of the fact that updates to `final` or `private` fields are limited. A `final` type field can only be assigned once in its declaring class. A `private` field can only be assigned in the declaring class. By analyzing assignments to such fields, we can often identify fields which always hold some constant length array objects. Such information can pass the bound of methods and be utilized by all methods of the class.

For programs using multidimensional arrays, VCA does not know any information about sub-arrays. Even if the programmer knows all sub-arrays have the same length, a conservative approach must assume that sub-arrays may have different lengths. *Rectangular array analysis* aims to determine if an array is guaranteed to be rectangular, i.e. all sub-arrays have the same length. Rectangular array information can be used to make VCA more powerful by allowing VCA to include sub-array accesses.

All three analyses have been implemented using the Soot bytecode optimization framework[34, 33], but could be easily implemented in other compilers with good intermediate representations. The Soot framework converts bytecode from class files into a typed 3-address representation called JIMPLE, and the analysis is implemented on this representation. In order to convey the results of the analysis to virtual machines we use the tagging/attributing capabilities of Soot to tag each array access instruction to indicate if the lower bound and/or upper bound checks can be eliminated. Moreover, a simple intraprocedural null pointer analysis generates null pointer check attributes about array references. The Soot framework then produces bytecode output, with the tag information stored in the attributes section of the class files. Virtual machines or ahead-of-time bytecode-to-nativecode compilers can then use these attributes to avoid emitting bounds checks based on the attributes. We have instrumented both the Kaffe JIT and IBM HPCJ ahead-of-time compiler to read these attributes.

We have experimented with 10 benchmark programs, including 5 specJVM benchmarks, 3 kernels from the `scimark2` suite[1] and 2 array-based benchmarks we implemented according to standard algorithms. First, we measured the complexity of our base VCA analysis, measuring both the maximum and average sizes of the constraint graphs, and the average number of times each block was analyzed. These results show that the analysis is practical, with small graph sizes (maximum size 13) and a low number of iterations (average always less than 3). We then measured the dynamic behavior of array bounds checks and compared the synthetic case when all bounds

[1] Available at `http://math.nist.gov/scimark2`.

checks are removed (an upper bound of what could be achieved with static analysis) and the results of our analysis. Not surprisingly, we found that it was much harder to eliminate upper array bounds checks than lower array bounds checks. We showed that the base VCA algorithm could eliminate from 3% to 60% of both the lower and upper bounds checks for array references, while adding the array field analysis and rectangular array analysis improved these results. In five of the benchmarks we could eliminate 60% or more checks and in three of those cases we eliminate more than 99% of the checks. We also provide run-time speed-ups, and we showed significant speed-ups for both the Kaffe VM and IBM's HPCJ.

1.2 Soot: background

We implemented algorithms on the Soot framework because it provides a stackless, typed, 3-address intermediate representation. All analyses work on this IR. Some Soot utility classes alleviate the work of development. Furthermore, the analysis results are passed to class files using Soot's attribute annotation functionality.

Soot is a Java bytecode optimization and annotation framework[28, 34]. Soot reads in a bytecode class file, converts it to an intermediate representation form called JIMPLE, which is a typed 3-address code. Static analyses and transformations are performed on the JIMPLE IR. After that, the JIMPLE IR is written back to the class file bytecode format.

In Soot, a bytecode class is represented with a `SootClass` object. Fields and methods are represented as `SootField` and `SootMethod` objects, respectively. A `SootMethod` object may have a method body, which consists of a chain of JIMPLE statements. Analyses can either directly optimize the JIMPLE statements by changing instructions (e.g. peephole optimizations, CSE, and static inlining), or encode results in class file attributes which can be used by a Java virtual machine (e.g. bounds checks and null pointer checks).

1.2.1 JIMPLE: a typed 3-address IR

JIMPLE is a 3-address (stackless) intermediate representation of bytecode. It simplifies the representation of more than two hundred types of bytecode instructions to about seventeen types of JIMPLE statements. A JIMPLE statement is a typical

3-address code, which is suitable for many analyses and optimizations. Readers can get detailed description from [33]. Here I would like to describe some features used for the analyses presented in this thesis.

Locals in JIMPLE code are typed by a static type inference system[8]. The operands of a statement have declared types. Based on these types we can determine if a method involves arrays by examining the types of its locals.

A static analysis on JIMPLE is simplified since each JIMPLE statement has only one complex feature. Figure 1.5 shows an example. An assignment from a field reference to an array reference is achieved by using a local variable. The focus of the first statement is the field reference, and the second statement emphasizes the array reference.

```
a[i] = o.f;            $r1 = o.f;
                       a[i] = $r1;

a) Java code           b) Jimple code
```

Figure 1.5: Example of JIMPLE representation

To further improve the results of analysis, local variables are split using def-use/use-def webs, which is a simple alternative to SSA form. Figure 1.6 shows an example of the original Java code and the resulting JIMPLE code. It should be clear that two assignments to variable `a` are split to two unrelated variables `r1` and `r2`.

```
a = new int[10];        r1 = new int[10];
a[i] = ... ;            r1[i1] = ...;
  ...                     ...
a = o.f;                r2 = o.f;
a[i] = ... ;            r2[i1] = ...;


a) Java code            b) Jimple code
```

Figure 1.6: Example of DU-UD webs

1.2.2 Intraprocedural analysis tool classes

For a method with bytecode, the Soot framework provides various control graphs, with or without exception edges, on the unit base or basic blocks, and so on. A set of well-implemented tool classes makes data-flow analyses (flow-sensitive or flow-insensitive) easy (see the package `soot.jimple.toolkits`).

Here, I describe a few classes used by VCA:

BlockGraph implements a control-flow graph (CFG) for a method body where the nodes of the graph are basic blocks.

BackwardFlowAnalysis provides the fixed point iteration functionality required by all backward flow analyses. VCA extends the BackwardFlowAnalysis to compute live locals related to array references.

ForwardBranchedFlowAnalysis provides functionality for branched forward flow analysis. A branched flow analysis can propagate different information to the successors/predecessors of a node (e.g., a conditional branch instruction). VCA uses a customized version of this class, which has special operations such as ordering graph nodes and widening edge weights.

1.2.3 Call graphs

Virtual method calls are resolved at run time, which means the exact type of a receiver may not be known at compilation time. However, for closed-world applications, the class hierarchy can be statically computed. *Class hierarchy analysis* (CHA) [7] provides a set of potential receiver types for a virtual method call. Moreover, *rapid type analysis* (RTA) [2] and *variable type analysis* (VTA) [31, 32] can make the type set smaller.

Based on the results of CHA, a conservative call graph can be built for a Java application. Whole-program (interprocedural) analyses need the call graph as a backbone. Soot has implementations of CHA, RTA, and VTA, and builds a conservative call graph for other analyses. Our *rectangular array analysis* is based on the call graph provided by the Soot framework.

1.2.4 Class file annotations

Soot can also be used as a bytecode annotation framework[24]. Because the bytecode is a relatively high-level instruction set, it hides some low-level operations behind the bytecode instructions. For example, a virtual machine implicitly performs the array bounds checks for array access bytecodes, such as `iaload`, `iastore`, etc. However, at the bytecode level, even if we know that an array access bytecode has an index in the safe range, it is impossible to represent such information in the bytecode itself. The attributes of a class file provide an alternative way to pass the results of a static analysis, which cannot be conveyed by the bytecode, to the underlying systems. A JIT or ahead-of-time compiler can then generate more efficient native code when it uses the annotation information. Figure 1.7 shows the internal structure of the Soot annotation framework.

Based on this idea, the results of our analyses are encoded in the attributes of a class file. The modified Kaffe JIT and HPCJ can use these attributes to optimize the native code they produce. The details of annotation goes beyond this thesis, but the modification of JIT compiler to utilize the attributes is described in Chapter 3.

1.3 Thesis Contributions

We have designed a new algorithm to prove the safety of array references in general Java programs. In our algorithm difference constraints, which are program-point-specific, are used to approximate the run-time value relationships among local variables. A constraint guarantees that, at the respective program point, a variable's run-time value is less than or equal to another variable's run-time value plus/minus a constant integer. If an index expression has a constraint that is bound to a value less than the length of an array object, the upper bound check can be removed at the run-time. Similarly, the lower bound check is redundant when the index is greater than or equal to the constant 0.

The basic *Variable Constraint Analysis* analyzes the code of one method. It constructs a constraint graph at each important program point. By using some special techniques (e.g. ordering CFG, widening edges, and liveness analysis), the analysis propagates constraint graphs along the control-flow graph of the method until reaching a fixed point. The relationships of variables can be inferred from the constraint graphs. VCA is also extended to take advantage of the information from our *array*

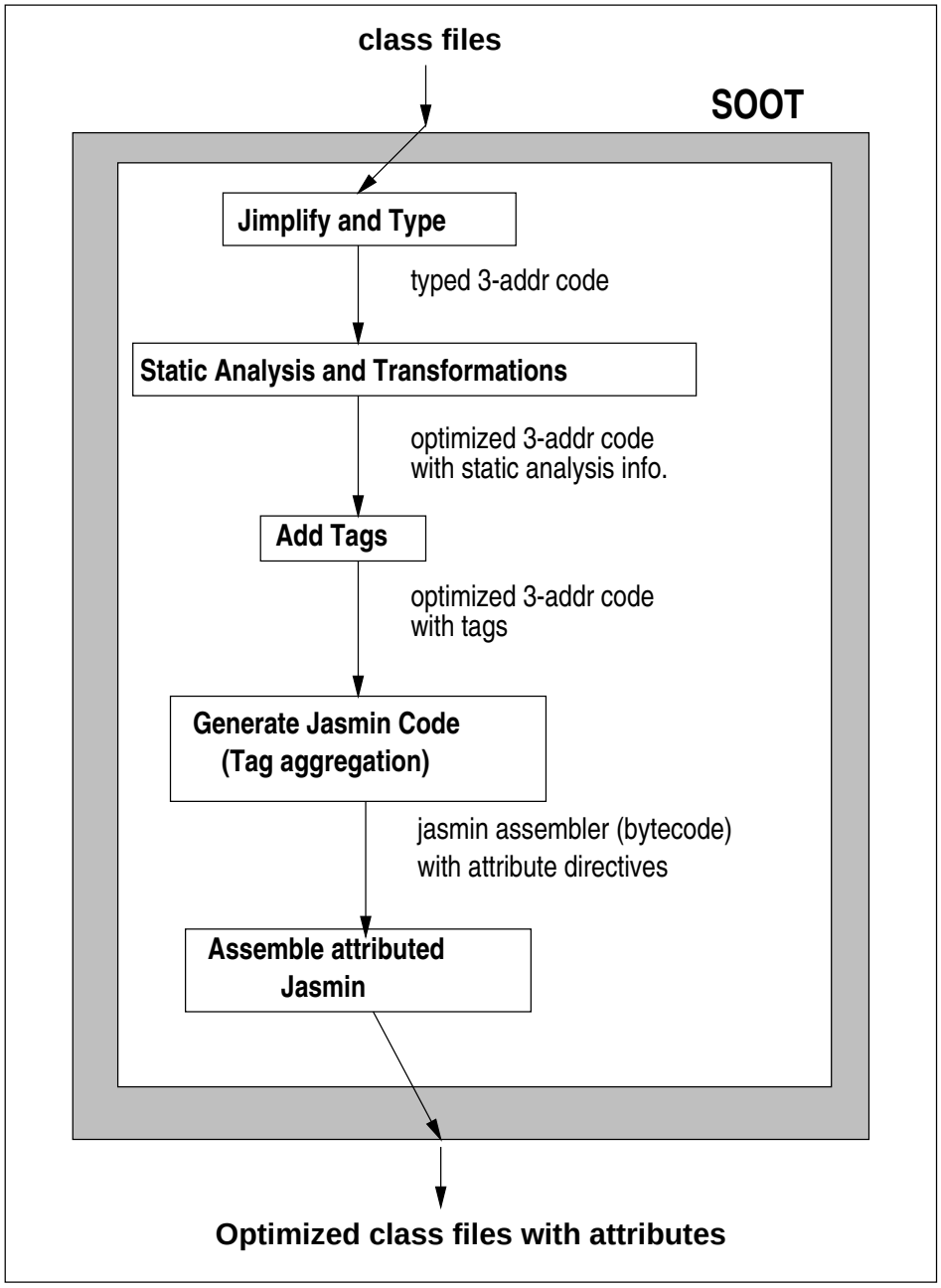


Figure 1.7: Soot annotation

field analysis and *rectangular array analysis*. We have implemented the algorithm in the Soot framework. General and array-intensive benchmarks are analyzed to demonstrate the effectiveness and efficiency of the algorithm. The results are encoded in the class file attributes via Soot's annotation functionality. We also demonstrate how to make a JIT compiler be aware of such attributes, and experiments on the Kaffe VM and IBM's HPCJ showed significant speed-ups.

In summary, the main contributions of this thesis are:

- Definition of the constraint graph and operations on it. We demonstrate how the array bounds check problem can be represented by a system of difference constraints, and how to solve the system by finding the shortest-path weight in the corresponding constraint graph. We also use several techniques to minimize the overhead of the analysis.

- Design of the array bounds check elimination algorithm, which includes three analyses:

  1. *Variable Constraint Analysis* (VCA) is an intraprocedural analysis which builds and solves constraint graphs in the scope of one method. VCA also serves as the basis for the two extended analyses.
  2. *Array field analysis* analyzes the assignments to a class field with specific modifiers. The analysis is performed in the scope of a Java class.
  3. *Rectangular array analysis* is for finding the shape of multidimensional arrays. It is an interprocedural analysis based on the call graph of a whole application. The analysis builds an array type graph and tracks down array shapes from paths leading to a method parameter or a local variable.

  The results of *array field analysis* and *rectangular array analysis* help the VCA improve the analysis of both one-dimensional and multidimensional arrays.

- Implementation of the algorithm in the context of Soot. The algorithm is implemented in pure Java language.

- Experiments on real JVMs. I defined the format of array bounds check attributes and modified Kaffe JIT compiler to use the attributes. The annotated class files were also provided to Clark Verbrugge at IBM Toronto Lab who performed the experiments using IBM's HPCJ ahead-of-time compiler.

1.4 Thesis Organization

The remainder of the thesis is structured as follows. We present our algorithm in Chapter 2. The base *variable constraint analysis* is presented in Section 2.1, the *array field analysis* and *rectangular array analysis* are presented in Section 2.2 and 2.3, respectively. We also discuss some enhancements made to the VCA in Section 2.4. Related *null pointer analysis* is described in Section 2.5. Experimental results are given in Chapter 3, where the modification of Kaffe JIT compiler is also described. The related work is discussed in Chapter 4, and conclusions are in Chapter 5.

Chapter 2

Analyses

In this chapter we introduce the three analyses used in our approach. The *Variable Constraint Analysis* is presented first because it serves the basis of the other two analyses. Then two extensions, *array field analysis* and *rectangular array analysis*, are described after VCA. Also, some extensions we made on VCA are introduced later, although they do not have obvious effects on our results. In the last section, we briefly describe an intraprocedural analysis for eliminating null pointer checks. In some cases eliminating array bounds checks requires inserting null pointer checks if the array reference cannot be shown to be non-null.

Each analysis is illustrated by graphs and examples. All examples are given in Java or JIMPLE form.

2.1 Variable Constraint Analysis

The objective of our *variable constraint analysis* is to determine the relationships between array index expressions and the bounds of the array. In Java, an array reference of the form `a[i]` is in bounds if $0 \leq i \leq a.length - 1$. If the array reference is out of bounds, an `ArrayIndexOutOfBoundsException` must be thrown, and this exception must be thrown in the correct context.

The relationships between variables can be represented as difference constraints. A system of difference constraints has a corresponding constraint graph. Finding the shortest-path weights in the graph gives a solution to the system. Our base analysis uses a *variable constraint graph* (VCG) to represent difference constraints

between variables. The VCG is a weighted, directed graph, in which nodes represent variables, constants, or other symbolic representations; and each edge has a weight to represent the difference constraint from the source to destination node. The analysis is intraprocedural and flow-sensitive. Each program point of interest (control-flow join points and array references) has a VCG to approximate the relationships between variables. These VCGs are propagated through the control-flow graph by using an optimistic worklist-based flow analysis. When the analysis reaches a fixed point, the distance in the VCG from an array variable to its index expression can be solved as the single-source shortest path problem. By reducing the size of the graphs, careful design of the worklist strategy, and the appropriate use of widening operators, we have developed an efficient and scalable analysis.

In the remainder of this section we introduce the concept of the variable constraint graph which is the essence of our algorithm. Then we describe the data-flow analysis, and finally we outline the techniques we used to improve the algorithm's performance.

2.1.1 Systems of difference constraints

Systems of difference constraints can be used to solve the general linear-programming problem[6](p.539-p543). A constraint is a simple linear inequality of the form

$$x_i - x_j \leq c_k,$$

where x_i, x_j are unknown variables and c_k is a constant. A solution to a set of difference constraints is a vector $(x_1, x_2, \cdots, x_n)$ which satisfies the constraints:

$$x_1 - x_2 \leq c_1$$
$$x_2 - x_i \leq c_i$$
$$\cdots$$
$$x_{n-1} - x_n \leq c_{n-1}$$

Now we show how systems of difference constraints can represent the array bounds check problem. Figure 2.1 is a piece of code from an insertion sorting program. Our goal is to prove three array references (except the first one) are safe, and thus no bounds checks are necessary for them. The corresponding JIMPLE 3-address code is in Figure 2.2(a). Figure 2.2(b) lists the difference constraints generated by each statement. For example, an assignment $j = i - 1$ produces two difference constraints:

$j - i \leq -1$ and $i - j \leq 1$; the array reference $a[i]$ generates $0 - i \leq 0$ and $i - a \leq -1$, where a represents the array length (because an out-of-bounds index expression can not pass the bounds checks of the array reference); and so on. The confluence point and special assignment ($j = j - 1$) need special operations (e.g. *merge* and *update*) to maintain the correctness of the analysis, we will talk about these in more detail later.

```
key = a[i];
j = i - 1;
while (j>=0 && a[j]>key)
{
     a[j+1]=a[j];
     j--;
}
```

Figure 2.1: A VCG example: Java source code

By walking through the instruction sequence, we can collect several difference constraints before an array reference. In the example given in Figure 2.2(a), we have five difference constraints before statement $\$i1 = a[j]$ (temporarily assuming there is no flow-joint point at *label_1*):

$$
\begin{aligned}
0 - i &\leq 0 \\
i - a &\leq -1 \\
j - i &\leq -1 \\
i - j &\leq 1 \\
0 - j &\leq 0
\end{aligned}
$$

where i, j, and a are variables, the 0 on the left side of inequality is a special node representing the lower bound of array references.

A system of difference constraints can be represented as a weighted, directed constraint graph, and a solution can be obtained by finding shortest-path weights in the graph. Given a system of difference constraints at the beginning of this section, the corresponding constraint graph is a weighted, directed graph $G = (V, E)$, where

$$V = \{v_0, v_1, v_2, \cdots, v_n\}$$

```
  key = a[i];          0 - i <= 0        i - a <= -1
  j = i - 1;           j - i <= -1       i - j <= 1

label_1:                                 merge(G1, G2)
  if (j<0)
      goto exit;
                       0 - j <= 0
  $i1 = a[j];          0 - j <= 0        j - a <= -1
  if ($i1 <= key)
      goto exit;

  $i2 = j + 1;         $i2 - j <= 1      j - $i2 <= -1
  $i3 = a[j];          0 - j <= 0        j - a <= -1
  a[$i2] = $i3;        0 - $i2 <= 0      $i2 - a <= -1
  j = j - 1;                             update(j, -1)
  goto label_1;

exit:
  ......
  (a) JIMPLE code      (b) Difference constraints
```

Figure 2.2: A VCG example: JIMPLE code and difference constraints

and

$$E = \{(v_i, v_j) : x_j - x_i \leq c_k\} \cup \{(v_0, v_1), (v_0, v_2), \cdots, (v_0, v_n)\}.$$

Each vertex v_i in the graph, for $i = 1, 2, \cdots, n$, corresponds to the variable x_i. An extra node v_0 makes all v_i reachable from it. The edge weight of (v_0, v_i) is initialized to 0. If the constraint graph G contains no negative-weight cycle, then $X = (\delta(v_0, v_1), \delta(v_0, v_2), \cdots, \delta(v_0, v_n))$ is a feasible solution for the system of difference constraints, where $\delta(u, v)$ is the *shortest-path weight* from u to v.

In our problem definition, however, we do not need to find a solution to all variables in the system of difference constraints. The shortest-path weight from the array variable node to the index expression node is sufficient to prove whether the upper bound check of an array reference is safe or not. Formally, if $\delta(a, i) \leq -1$, $a[i]$ has a safe upper bound check; if $\delta(i, 0) \leq 0$, $a[i]$ has the safe lower bound check. Figure 2.3 shows the corresponding constraint graph before the statement $\$i1 = a[j]$, where

$\delta(a, j) = -2$ and $\delta(j, 0) = 0$. Therefore, the $a[j]$ can be proved to be safe.

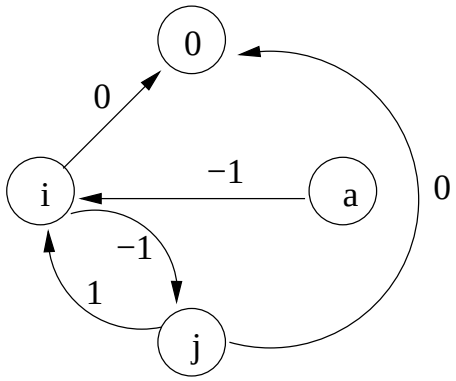


Figure 2.3: The constraint graph before $\$i1 = a[j]$

2.1.2 Variable constraint graphs

Given the JIMPLE 3-address representation of a method body, we build a control-flow graph (CFG) of basic blocks, where a statement with an array reference breaks a basic block into two smaller ones. Thus, the array accessing statement will always appear at the top of a basic block. Each basic block is associated with an input VCG. Difference constraints are collected when going through statements in the block. The new constraints are incorporated into the constraint graph directly. At the exit of the block, an output VCG is produced, and passed to successors as their input VCGs. We define a variable constraint graph as follows:

A *node* in a variable constraint graph represents one of:

- an *int* type local which is related to some array index or array object length;
- an *array* type local which is used to represent the length of the array;
- a 0 node representing the lower bound of array references; or
- an abstract representation for fields, array elements, and common sub expressions (used only in Section 2.4).

A directed *edge* in a variable constraint graph is associated with an abstraction value which is one of:

- $\perp$, the edge is uninitialized;
- an integer constant; or
- $\top$, there is no **constant** constraint from the source to the destination.

The weights associated to edges are comparable. The integer constants are in the order of ordinary integers. For any constant c, the ordering $\perp < c < \top$ holds. The $\perp$ weight is a special case, it is only used to represent the graph as uninitialized (or never visited). As we can see later, the iteration on a control-flow graph follows the graph's pseudo-topological order, and the first input graph's edges are initialized to $\top$, we never operate on an uninitialized graph except merging it with some other initialized graphs.

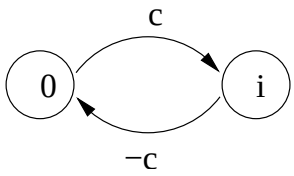
From a system of difference constraints to a *variable constraint graph*, a variable on the left hand side of an inequality has a corresponding node in the graph. The graph can be viewed as full-connected. If there is an inequality of $i - j \leq c$, the corresponding edge from j to i is associated with weight c. Other edges without corresponding constraints have weight $\top$. Using this representation, we show how constraints are generated and how to operate on the constraint graph in following text.

Constraint generation

When going through a statement, some constraints may be generated (and some may be killed, which is explained later). We have seen a few examples in Figure 2.2 how statements generate difference constraints. Generally, an assignment may build constraints between its right and left hand side variables. An array reference expression bounds its index expression in the range of 0 to array length minus 1. For branch instructions, different constraints are produced according to the outcome of the branch condition. We define the constraint generation here for different types of statements and expressions. Other effects of the statements, such as killing constraints of a node, are discussed afterwards. In our rules, c is an integer constant, i and j are integer variables related to some array references, a is an array type variable and represents the array object length.

i = c

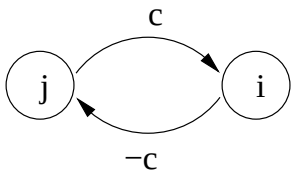
Assigning an integer to a local variable generates two constraints: $i - 0 \leq c$ and $0 - i \leq -c$. The constraint graph is changed by adding an edge from node 0 to i with weight c and a reversed edge with weight $-c$.



We do not create a node for each integer constant appearing in statements, but represent the constraint as edges to/from the 0 node with adjusted weights. This approach ensures the graph size manageable, and more important, the 0 node can connect two variables which have no direct edges between them.

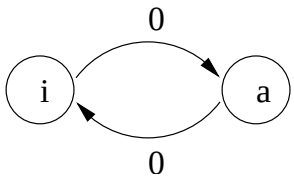
i = j + c

The statement also generates two constraints: $i - j \leq c$ and $j - i \leq -c$. The edges added to the graph are following:



i = a.length

The `arraylength` is a bytecode instruction which gets length of an array. The expression can be views as a variable like others. In our representation, the array variable a is used to represent the length of array. Then the constraints generated from the statement are $i - a \leq 0$ and $a - i \leq 0$. The edges in the graph are:



a = new T[c]

A new expression assigns the variable on the left hand side the length of c.

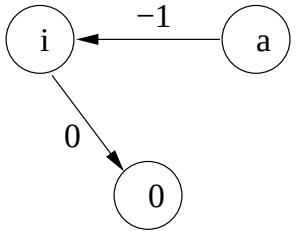
It has the same effect as the assignment $a.length = c$. Using a to represent $a.length$, the constraints from the new statement are $a - 0 \leq c$ and $0 - a \leq -c$.

a = new T[i]

This statement has the same effect as statement $a.length = i$, and constraints generated are $a - i \leq 0$ and $i - a \leq 0$.

a[i]

We know that the JVM check the bounds of an array reference. If the index i is not in the range of bounds, the JVM throws an `ArrayIndexOutOfBounds-Exception` and exits the normal execution path. So, on the normal execution path, the index i must have passed the bounds checks after the array reference $a[i]$. Then the array reference expression produces two constraints: $0 - i \leq 0$ and $i - a \leq -1$, which can be represented as following edges:



if (i < j)

The *iflt* conditional branch instruction has two out paths. In this example, the `TRUE` path has constraint $i - j \leq -1$, and the `FALSE` path has constraint $j - i \leq 0$. We can use the same way to derive constraints from other branch conditions such as *ifeq*, *ifgt*, *ifge*, and *ifle*.

i = j & c

Some constraints are not obvious in the statement. An arithmetic *and* expression of j & c will make the expression value no more than c if c is a positive integer. Then two hidden constraints, $0 - i \leq 0$ and $i - 0 \leq c$, are derived from this statement.

Two special cases have no constraint generation, but need special operations on the graph. We discuss them here, and the operations are described in next subsection.

i = i + c

A loop induction variable increases or decreases itself. The rules above can only

generate difference constraints between different variables, and obviously none can be applied directly on this case. The assignment, however, can be written in another form by using a temporary variable:

$$i^{'} = i + c$$
$$i = i^{'}$$

In this way we can find suitable rules for the new statements. In fact, it has same effect as increasing i's in-edges' weights by c and decreasing its out-edges' weights by c after bypassing the temporary variable $i^{'}$ in the graph. We defined an operation `update` to handle the changes in the graph due to these kinds of assignments.

***i* = ⋯**

When a variable i is assigned a new value, its old constraints have to be removed before new constraints are added (except $i = i + c$ where the `update` function performs this operation implicitly). Instead of removing old constraints of i directly, however, we take a special operation `detachnode` to bypass the node i. If the right hand side expression is one of the cases above, the new constraints are added in the graph, otherwise, we do not take any action.

Constraint graph operations

The implementation of the constraint graph can use either the adjacency-list representation for sparse graphs, or the adjacency-matrix representation for dense graphs. Because the graph size is relatively small, we implemented the graph as a collection of adjacency lists. As we introduced before, an edge's weight can have different values. $\perp$ indicates the edge is uninitialized. However, in our analysis, iterating the CFG in its pseudo-topological order ensures that only all edges of an uninitialized graph can be $\perp$ at the same time. Once the graph is initialized, its edges can never be $\perp$ again. Thus, in our representation, $\perp$ is indicated by a state variable of the graph. In an initialized graph, a physical edge of a pair of nodes has an integer constant weight, otherwise, it means the pair has a virtual edge with weight $\top$. In following text, we assume an initialized graph is full-connected with physical or virtual edges. The edge weight is an integer constant or $\top$.

No matter what kind of representation we use, however, the functionality of the constraint graph is independent of the implementation. In the following text, we

introduce these functions (or primitives) in further detail. All operations are only applied on an initialized graph where the edge weight cannot be $\perp$.

Creating a graph:

When we do flow-analysis, only variables related to some array references need to be examined. As can be seen later, at an interesting program point, if the set of variables under examination does not change, then the graph node set will not change. The creation function accepts a set of variables as vertices. The graph does not provide any functionality to add or delete variables. Graph edges can be set to $\top$ for the entry block's input graph, or the graph state variable is set to $\perp$ which means the graph is in an uninitialized state.

Adding a constraint:

When collecting a new constraint, we add a new edge to the constraint graph. The addition will make the graph have more than one (physical or virtual) edge from a source to a destination. However, we only need to keep one edge for each pair of source and destination, which has the smallest weight, to guarantee that both constraints hold. It can be proved as follows. Two edges can be written as two constraints:

$$i - j \leq c_1 \tag{2.1}$$
$$i - j \leq c_2 \tag{2.2}$$

where $c_1 \leq c_2$. If inequality 2.1 is true, 2.2 is automatically true. Then inequality 2.2 is redundant.

When adding an edge to a graph, we keep the one with the smaller weight. The abstract value $\top$ is greater than any other values.

```
addedge(from, to, weight)
  oldweight = edge(from, to).weight
  if (oldweight > weight)
    edge(from, to).weight = weight
```

Deleting a constraint:

When a constraint does not hold anymore, the corresponding edge weight should be changed to reflect the removal of the constraint. The edge weight is set to $\top$ in the graph. Right now, a constraint is deleted only in `detachnode` operation.

```
delete_edge(from, to)
  edge(from, to).weight = TOP
```

Updating a node's in and out edges:

For an expression $i = i + c$, we do not kill the node i. Rather, all in-edges' weights are increased by c, and all out-edges' weights are decreased by c, to reflect the constraint changes. For example, there is an existing inequality of $i - a \leq c_1$, and we use $i^{'}$ represent the new value of i after the assignment $i = i + c$. We have constraints:

$$i - a \leq c_1$$
$$i^{'} - i \leq c$$

from which we can easily get $i^{'} - a \leq c_1 + c$. The weight of in-edge from a is added by c. The same process can be used to derive the out-edge changes.

```
update(node, c)
  for each predecessor p of node
    edge(p,node).weight += c;

  for each successor s of node
    edge(node,s).weight -= c;
```

Detaching a node:

When a variable is assigned a new value, its old constraint edges should be removed before adding new ones. However, the edges may be part of some paths connecting other nodes, and we wish to retain this information. Thus, the `detachnode` primitive first builds edges from each predecessor to each successor, and then removes all in and out edges.

```
detachnode(node)
  for each predecessor p of node
    for each successor s of node
      edge(p, s).weight = edge(p,node).weight
                        + edge(a,node).weight
    delete_edge(p,node)

  for each successor s of node
    delete_edge(node,s)
```

Making the shortest path:

A constraint graph also provides methods to find the shortest path between two nodes or of all pairs. It implements the single-source shortest paths and all-pairs shortest paths algorithms[6]. If the method detects a negative cycle existing in paths, it aborts the operation. This is a conservative decision. As can be seen in following text, there should not be any negative cycles at reachable program points after reaching the fixed point.

Merging two graphs

At confluence points we must merge VCGs coming from more than one predecessor. All predecessor graphs will have the same set of nodes, but their edges may have different weights. Thus, merging graphs is done by simply merging edge weights. Note that this is different than adding an edge to a graph. Adding edges implies the new and old constraints are existing at the same time (in logic, they are `AND` relationship), and the tighter one gives the most precise information. Merging edges means different constraints from multiple paths are all possible (they are `OR` relationship). So the merged constraint should be able to contain all possibilities, as thus we must use the weakest constraint. One or more VCGs from predecessors may not be initialized. When an initialized graph (not $\perp$) is merged with an uninitialized graph ($\perp$), we simply take the initialized one. The complete merging table is given in Table 2.1.

	$\perp$	c1	$\top$
$\perp$	$\perp$	c1	$\top$
c2	c2	MAX(c1, c2)	$\top$
$\top$	$\top$	$\top$	$\top$

Table 2.1: Merge two edge weights

It is important to note that when computing the merge of an edge $p \rightarrow q$ from two graphs $G1$ and $G2$ we need not use the value stored on the edges, rather we can get a more precise answer by using the shortest path. Thus, we merge the shortest path from p to q in $G1$ with the shortest path from p to q in $G2$.

```
merge(G1, G2)
```

```
if G1 is uninitialized
  return a copy of G2
if G2 is uninitialized
  return a copy of G1

make G1, G2 be the shortest-path graphs

G = make a copy of G1

for each edge e of G1
  e1 = G1.e.weight
  e2 = G2.e.weight
  if e1 is TOP or e2 is TOP
    G.e.weight = TOP
  else
    G.e.weight = MAX ( G1.e.weight, G2.e.weight )

return G
```

Negative Cycles

In a directed constraint graph with negative edge weights, it is possible that a negative cycle exists at some points of the data-flow analysis, before the fixed-point is reached. However, after reaching the fixed point, every reachable point in the program should have a graph without negative cycles. For example, if a negative path from a to b to c, and back to a, as in the figure 2.4, the edge weight is w_a, w_b, and w_c while $w_a + w_b + w_c < 0$. So we have

$$b - a \leq w_a$$
$$c - b \leq w_b$$
$$a - c \leq w_c$$

Adding both sides, we get $0 \leq w_a + w_b + w_c$, which is a contradiction to the assumption.

It is possible to have a graph with negative cycles for programs with unreachable code due to useless branches. For example:

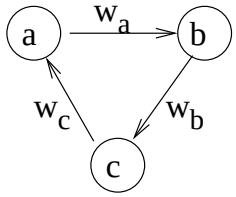


Figure 2.4: A negative cycle

```
if (i < j) {
    if (j < i) {
    P:  ......
    }
}
```

would lead to a negative cycle at program point `P:` (see Figure 2.5), but of course this point is never reached. In the presence of negative cycles in a path, we cannot compute the shortest path weight for nodes in the path. Leaving them unchanged is a conservative approach to keep the correctness of the analysis.

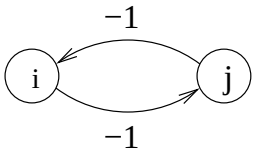


Figure 2.5: The negative cycle at P:

Properities of a constraint graph

After seeing how the array bounds check problem is converted to solving systems of difference constraints and the difference constraints are encoded in a *variable constraint graph*, we would like to study some properties of the constraint graph. A variable constraint graph has the following important properties.

Directed Edges: Instead of keeping equality relationships, an assignment statement produces two directed edges between nodes. The first five cases of constraint generation generates two edges between nodes with reversed directions.

The branch instructions and array references generate asymmetric edges. But all edges are directed and weighted. This approach unifies the graph representation for the constraints from different sources.

Inequality edges are transitive: A path from a_1 to a_n can be represented by a series of constraints, for example the constraints in Figure 2.6 are:

$$a_2 - a_1 \leq w_1$$
$$a_3 - a_2 \leq w_2$$
$$\cdots$$
$$a_n - a_{n-1} \leq w_{n-1}$$

By summing both sides, we can derive the constraint $a_n - a_1 \leq w_1 + w_2 + \cdots + w_{n-1}$, which implies the dashed edge from a_1 to a_n with weight $\sum_1^{n-1} w_i$. The transitive property simplifies graph operations. Any new constraints are added directly as edges. The edge nodes, however, can indirectly get constraints from other nodes connected in the graph. We can lazily perform some other operations , such as detaching a node, computing the shortest path, as required.

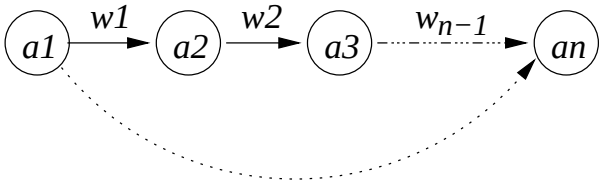


Figure 2.6: Transitivity of inequality edges

Shortest path gives the tightest constraint: Several paths may exist from a source to a destination node in the graph. Each path represents some constraints from different sources. However, only the shortest path gives the most accurate approximation. Any non-shortest paths are conservative estimations; they are correct, but not as precise.

Because the inequality graph is transitive, it has the advantage of preserving constraints when some variables are redefined. Figure 2.7(a) gives an example of four statements.

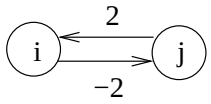
$s0 : i = j + 2;$

$s1 : a[i] = \cdots ;$

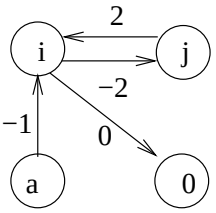
$s2 : i = \cdots ;$

$s3 : a[j] = \cdots ;$

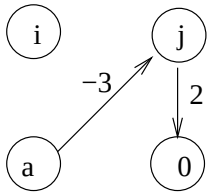
(a) a basic block



(b) the constraint graph before s1



(c) the constraint graph before s2



(d) the constraint graph before s3

Figure 2.7: The status of constraint graph changes

Figure 2.7(b), (c), and (d) show the constraint graphs before the statement $s1$, $s2$, and $s3$, respectively. We are interested in the graph before $s3$ because it has an array access and we want to know whether j is in the bounds. The other two graphs only reflect the constraint changes.

The statement $s1$ generates the constraint $\mathtt{i} - \mathtt{a} \leq -1$, which makes a path from a to j, and $\mathtt{0} - \mathtt{i} \leq \mathtt{0}$. The path from a to j implies the constraint $\mathtt{j} - \mathtt{a} \leq -3$ by adding its edge weights. Statement $s2$ detaches the node i from the graph by

bypassing it. Before the statement $s3$, i has lost its constraints from a and j, but the path from a to j, which goes through i, is shortcut by a new edge directly from a to j with weight -3. Thus the constraint $\mathtt{j} - \mathtt{a} \leq -3$ is preserved before $s3$, even when i was redefined. Therefore, the upper bound check for $s3$ can be proved to be safe (we can not derive the safe lower bound from this simple example, because it only implies $0 - j \leq 2$).

So far, we can conclude some advantages of using constraint graphs for array bounds check elimination, although there are many other abstractions that can be used too. The constraint graph offers several advantages, including:

1. As we explained in above text, a constraint graph can represent and preserve indirect constraints, even when a variable is redefined.

2. It has a unified representation for constraints from difference sources, e.g. assignments, conditional branches, and array references.

3. The lower and upper bounds relationships can be represented in the same graph. Array object, index, and constant 0 are encoded in the same graph.

4. It is flexible, and can be extended to hold other information. For example, in Section 2.4, we show how to include information about the second dimension of rectangular arrays and common sub-expressions.

Certainly, the *variable constraint graph* has some weakness. It can not represent some subtle constraints that we can infer from semantics of the language. A typical limitation is that it is hard to represent other arithmetic operations such as *multiply* and *division*.

2.1.3 Data-flow analyses

To understand how a method manipulates its data, we can apply data-flow analyses on the code of a method body. We developed two data-flow analyses in our algorithm. A special live-local analysis, which is relatively simple, determines which locals are relevant to array references. A more complicated analysis performs abstract execution of the method, and gets a conservative approximation of constraints among live locals. The first analysis limits the number of nodes in a constraint graph and therefore reduces the computation of the second analysis.

Array-related liveness analysis

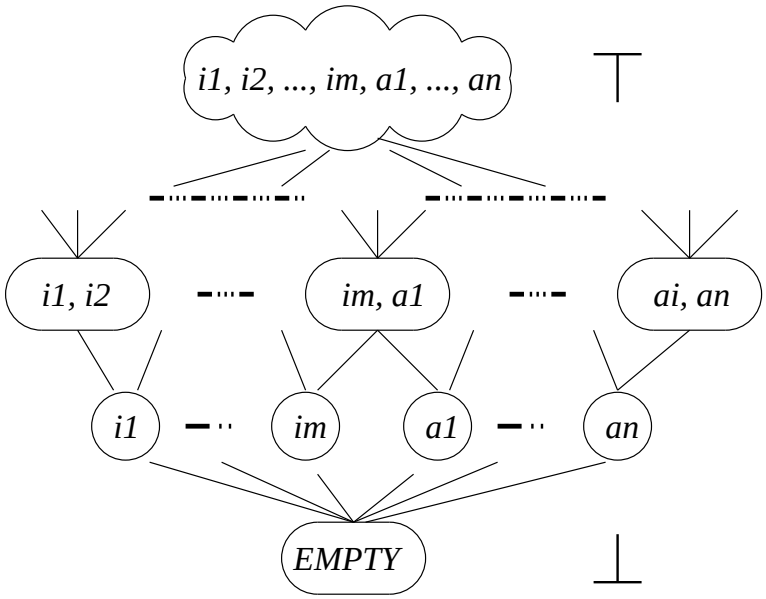
A variable constraint graph contains nodes of locals and edges between them. The size of the graph can be reduced by including only those locals that are used to compute an index or an array object length in the future. A smaller constraint graph allows faster computation of shortest paths, and may also reduce the number of iterations required for the fixed-point computation.

In our liveness analysis, a variable is live at a program point if there is an execution path from this program point to an array reference expression such that the constraints collected by using constraint generating rules defined in section 2.1.2 can form a path from the variable to the array index or array object length in the corresponding constraint graph. We briefly say that the variable is relevant to some array references. Our goal is to determine that whether we need to add a constraint collected at this point to the constraint graph by consulting the liveness of the variable.

We formulate the analysis as follows:

Partial ordering for approximation domain

In this analysis, we have a set of *int* or *array* type local variables. The extended analysis includes fields, array elements, and common sub-expressions. The partial ordering of the set is from empty set ($\perp$) to the full set of variables ($\top$). It is best represented by following picture, assuming the method has int type locals $(i_1, i_2, \cdots, i_m)$ and array type locals $(a_1, a_2, \cdots, a_n)$:



Problem statement

We already defined the liveness of a local in the above paragraph.

Direction

As with ordinary liveness analysis, it is a backward flow analysis.

Confluence operator

At the flow-joint point, we are take union operator

$$outset = set_1 \cup set_2,$$

because a local is live at this program point if it is live in any paths from this statement.

Equations for instructions

Table 2.2 provides the key flow functions. The first column gives the types of statements or expressions that may generate or kill live locals. The second and third column should be used together. Only when at least one of the local(s) in the condition set are live, does the statement generate live locals in the *gen* set. Note that array references generate live locals without any conditions. The statement $i = i + c$ needs no operations because the variable is increasing/decreasing itself. For any assignment statements that are not the case listed in the table, the left hand side variable is removed from the set.

stmt/expr	cond	gen	kill
$i = j + c$	i	j	i
$i = a.length$	i	a	i
$a = new\ T[i]$	a	i	a
$a\ [\ i\ ]$		a, i	
$if\ \ (i\ op\ j)$	i, j	i, j	
$i = i + c$			
$i = \cdots$			i

Table 2.2: Liveness for array references

When going through a statement s, we retrieve the $cond(s)$, $gen(s)$, and $kill(s)$. The equations for computing IN and OUT sets are changed to reflect the conditions.

$$OUT[s] = \bigcup_{p \in succ[s]} IN[p]$$

if $cond[s] = \phi$ or $cond[s] \cap OUT[s] \neq \phi$

$$IN[s] = gen[s] \cup (OUT[s] - kill[s])$$

else

$$IN[s] = OUT[s] - kill[s]$$

The starting approximation

The analysis starts with the safe approximation. Because the analysis is backward, all nodes' out sets are initialized as ϕ.

Now we look back the example in Figure 2.2. Although variable `$i1` and `key` can be *int* type variables, there is no path leading them to an array reference. We do not collect constraints produced by the if ($\$i1 \leq key$) statement.

One can easily extend the liveness analysis to accommodate other special nodes, such as class fields, array elements, and common sub-expressions.

Variable Constraint Analysis

We use a forward, flow-sensitive, optimistic data-flow analysis to approximate a variable constraint graph for each important point in a method body. We named the analysis as *variable constraint analysis*, or VCA.

VCA is based on the control-flow graph of basic blocks as we explained before. An instruction with an array reference appears on the top of the basic block. The entry of each basic block is associated with a VCG. The initial state of each graph has $\perp$ state, except the entry point graph which has all $\top$ edges. The analysis is driven by a worklist algorithm which computes an output VCG based on the input VCG and the effect of the statements in the basic block. When processing a conditional branch statement, it may generate different constraints for the target block and the next block. After reaching a fixed point, the information for each array access statement, S, is encoded by the VCG associated basic block starting with S.

Now we define the *variable constraint analysis* formally:

Partial ordering of approximation domain

At any program point the set of interesting variables is known from array-related liveness analysis, so the set of nodes is fixed. There is one node for each

variable of interest, plus a node representing the constant 0. The abstraction computed by our analysis is all-pairs shortest paths of a *variable constraint graph*. But instead of computing the shortest paths at every program point, we only perform such computation at the confluence point. In other places, we do simple operations on the graph. The abstract information that changes is the weights associated to edges. For any constant c, the ordering $\bot \sqsubset c \sqsubset c+1 \sqsubset c+2 \sqsubset \ldots \sqsubset maxint \sqsubset \top$ must hold.

Problem statement

A pair of nodes (i, j) has the shortest path weight of c from j to i at a program point P if the symbolic execution of the program can guarantee that the constraint $i - j \leq c$ holds at any time when it reaches the program point P.

Direction

The *variable constraint analysis* is a forward flow-analysis. Moreover, it requires the node of the CFG must be visited in its pseudo-topological order because the analysis is simulating the execution of the program. The *dominator*s of a node must be visited before that node. Recall that we initialize the entry point graph to $\top$ and other graphs to $\bot$. By keeping the topological order, the input VCG of a basic block can never be $\bot$ when we start to go through it.

Moreover, the analysis is flow-sensitive. When going through a conditional branch statement, different constraints may be produced for different out paths of the branch. The flow function of `if` statement adds different edges to the target and next graphs.

Confluence operator

At a confluence point P, we use a set of output graphs from predecessors (G_1, G_2, $\cdots$, G_n) and the old input graph *oldgraph(P)* to compute the new input graph *newgraph(P)*. We firstly call the `merge` operation to union all output graphs from predecessors:

```
newgraph = copy of G1

for i = 2 to n
  newgraph = merge( newgraph, Gi )
```

Then we apply a special operation called widening on each new graph edge weight by comparing it to the old graph edge weight.

`widen(newgraph, oldgraph)`

The widening operation looks at the changing trend of an edge weight. If the weight is increasing, we set it to $\top$ directly. But if the new weight is less than the old weight, we will discard the new weight and use the old one. The widening technique speeds up the symbolic execution and also stops infinite loops correctly. We will explain it in detail later.

Equations for instructions

The base analysis deals only with local variables. It is obvious that the integer locals cannot be aliased, nor can they be modified by method calls. The array objects referenced by array type locals have the same properties. We only deal with the first dimension of arrays in our base analysis. Once an array object was created, the only way to change the array size is to re-allocate a new array object. Then, the array lengths can be treated as integer locals in the same way. Thus, the effect of each statement on a VCG is quite straightforward. The flow function for each kind of relevant JIMPLE statement is given in Table 2.3. Variables i, j and a represent nodes in the graph, and c is an integer constant. Each graph has a node for the constant 0.

The first column shows the kinds of statement which have effect on a VCG. The second column lists the constraints can be generated from the statement in the first column. The third column shows the node of which constraints should be bypassed. The last column gives operations on the constraint graph according to the statement. We always check the liveness of variables before performing the flow-through function for a statement. Only when the variables are live, the operations on the graph are performed.

The rules in Table 2.3 use several primitives, which were defined in section 2.1.2. The kinds of statement that can affect constraint graphs depend on the semantics of languages. Table 2.3 defines some basic statements for Java. One can also add more complicated ones if they do not violate the language semantics. We will show a few extension in section 2.4.

The starting approximation

As we stated before, the edges of entry point VCG are initialized to $\top$, which is the safe solution. Other VCGs' edges are set to $\bot$.

Briefly, the implementation of the analysis uses a heap (implemented as `Bounded-PriorityList`) to maintain the topological order of blocks in the control-flow graph.

stmts	gen	detach	operations
$i = c$	$i - 0 \leq c$ $0 - i \leq -c$	i	detachnode(i) addedge(0,i,c) addedge(i,0,$-c$)
$i = j + c$	$i - j \leq c$ $j - i \leq -c$	i	detachnode(i) addedge(j,i,c) addedge(i,j,$-c$)
$i = a.length$	$i - a \leq 0$ $a - i \leq 0$	i	detachnode(i) addedge(a,i,0) addedge(i,a,0)
$a = new\ T[c]$	$a - 0 \leq c$ $0 - a \leq -c$	a	detachnode(a) addedge(0,a,c) addedge(a,0,$-c$)
$a = new\ T[i]$	$a - i \leq 0$ $i - i \leq 0$	a	detachnode(a) addedge(i,a,0) addedge(a,i,0)
$a\ [\ i\ ]$	$i - a \leq -1$ $0 - i \leq 0$		addedge(a,i,-1) addedge(i,0,0)
$if\ (i\ <\ j)$	target: $i - j \leq -1$ else: $j - i \leq 0$		 addedge(j,i,-1) addedge(i,j,0)
$i = j\&c$	$i - 0 \leq c$ $0 - i \leq 0$	i	addedge(0, i, c) addedge(i, 0, 0)
$i = i + c$			update(i,c)
$i = \cdots$		i	detachnode(i)

Table 2.3: Statements generating constraints

The VCGs are associated to the edges of the CFG instead of being attached to the blocks directly. Each head of the CFG has an auxiliary edge as its incoming edge. The input graph of a block comes from merging all graphs on its incoming edges. The output graph is associated to each outgoing edge. A block with a branch as the last instruction would produce two different output graphs for its two out-edges, which makes the analysis conditional. A block also keeps the input graph after merging the incoming edges' graphs. To better understand the *variable constraint analysis*, we provide the pseudo-code in Figure 2.8, some functions used by `worklist` are defined

in the later paragraph introducing the `BoundedPriorityList` class.

The `flowThrough` function take an input VCG and goes through a basic block. It operates on the VCG according the flow functions in table 2.3, and updates the VCGs associated to the block's out-edges. It returns the set of successor blocks whose incoming edge's VCG has changed. When going through a basic block, some variables added in the temporary graph may be not live at the end of block, we detach those nodes when updating out-edges' VCGs.

2.1.4 Improving the performance of the algorithm

A naive implementation of the algorithm requires a large volume of computation to reach the fixed point. We can analyze the expensive parts of the algorithm. There are two factors dominating the performance of the algorithm: the variable constraint graph size and the time that the data-flow analysis takes to reach the fixed point. In this section, we describe some techniques we have used to reduce the performance overheads in our algorithm.

Limiting the size of constraint graphs

The running time of computing the shortest path on a graph depends on the number of nodes and the number of edges. Since we cannot directly control the number of edges, we reduce the number of nodes, which subsequently reduces the number of edges. The array-related liveness analysis keeps the node size minimal. The experiment shows the average node size is less than 10 and the maximum node size never exceeds 13 for the base VCA.

Widening edges at confluence points

Given the long chains in the ordering for edge weights, the ordinary fixed-point computation is too expensive. We reduce the number of iterations by applying a widening at loop entry points. At these points we replace the ordinary merge operation which uses the maximum value with a widening implemented as follows. If an edge's previous weight was not $\perp$ and the current weight increases, the edge is set to $\top$. Thus, it is clear that an edge's weight at loop headers can change two times at most along the same execution path. The following is the pseudo-code for the operation.

```
units = make PseudoTopologicalOrder of the CFG
worklist = make BoundedPriorityList of units

/* initializes all VCGs to BOTTOM. */
for each edge of CFG
{
  edge's VCG = new VCG with live locals of
                  edge's source node
  edge's VCG is set to BOTTOM
}

/* initializes the entry VCGs to TOP. */
for the incoming edge of CFG heads
  edge's VCG is set to TOP

/* performs iterative flow-analysis. */
while not worklist.isEmpty()
{
  Block block = worklist.removeFirst()

  prevVCG = block's input VCG

  if the block has only 1 incoming edge
    beforeVCG = copy of incoming edge's VCG
  else
  {
    beforeVCG = merge all incoming edges' graphs
    widen ( beforeVCG, prevVCG )
  }

  block's input VCG = copy of beforeVCG

  List changedSuccs = flowThrough ( block, beforeVCG )

  add all elements of changedSuccs to the worklist
}
```

Figure 2.8: Pseudo-code of the worklist algorithm

```
widen(newgraph, oldgraph)
  for each edge of oldgraph and newgraph
  do
    if oldgraph's edge weight is BOTTOM
        continue;

    if oldgraph's edge weight is less
       than newgraph's edge weight
       set newgraph's edge to TOP.
  done
```

A subtle effect of widening edge weights is that it can stop the flow-analysis quickly and correctly on an infinite loop. For example, a programmer may unintentionally write an infinite for loop as in Figure 2.9. Without widening edge $(i, 0)$ at the loop

```
for ( int i=0; i<a.length; i-- )
  ...
```

```
  int i=0;

label_1:
  if ( i >= a.length)
    goto exit
  ...
  i = i-1;
  goto label_1

exit:
  ...
```

Figure 2.9: An infinite for loop

entry `label_1`, $\delta(i, 0)$ is increased by 1 for each iteration over the loop body. The analysis cannot ever reach the fixed point. However, the widening function can find out that $\delta(i, 0)$ is increasing when the analysis visits `label_1` the second time, then set $\delta(i, 0)$ to $\top$, and the analysis stops correctly.

Ordering the nodes of a CFG

Walking through a CFG in its pseudo-topological order can speed up data-flow analysis. However, a simple depth-first search (DFS) algorithm cannot guarantee an

optimal order for the successors of a loop exit node.

For our analysis, we prefer to visit the loop body before the loop exit. To enforce a good ordering we perform a DFS from exiting nodes of the CFG in reverse order first; then the DFS from the starting node can consult the order of reversed DFS when it meets a loop exit allowing us to put loop body nodes before loop exits.

Our worklist algorithm puts the successors of a node, whose *out* set changes, onto the worklist for re-calculation. The worklist is handled as a heap using the order computed as above. By enforcing this order we ensure that inner loops reach a fixed-point before the outer loops. Experiments show this is very effective way of making our data-flow analysis run efficiently.

The worklist is implemented as the class `BoundedPriorityList` which provides several methods:

public BoundedPriorityList(List list)

The constructor accepts a list as the *fulllist* (universal set), the order of each element is decided by its index in the list. The fulllist is a list of blocks in an optimal topological order computed as above. This list is used to keep the index of each element, another linked list is created as the *worklist*. All elements in fulllist are added to the worklist in order.

public boolean isEmpty()

The method returns *true* if the worklist is empty, otherwise returns false.

public Object removeFirst()

This method removes the first element in the *worklist* and returns it to the caller.

public void add(Object toadd)

When a block needs re-computation, it is put back to the worklist. However, unlike the usual worklist which adds the node to the end of the list, this method will find the right place in the *worklist* by its index. All elements are kept the order in worklist as the same order in the *fulllist*.

2.1.5 Running time analysis

The performance of our algorithm is decided by two factors: the size of constraint graphs and the number of iterations required to reach a fixed-point. The nodes of a

constraint graph consists of locals, therefore, the graph size is bounded by the number of locals in a method. Liveness analysis can limit the graph size even further, and our experiments confirm the graphs are small in practice.

For a control-flow graph without cycles, the data-flow analysis takes linear time to reach the fixed point. However, most of methods contain loops. At a loop entry, the special widening step of comparing an edge weight with before makes the edge weight reach a fixed-point quickly. An edge weight can not change more than twice because of visiting the same path. So the upper bound of the analysis depends on the depth of loops and the number of nodes in the loops. It can be represented as $|N| + \sum 2^{|LD|+1} * |LN|$, where N is the total number of nodes in a CFG, LD is the loop depth, and LN is the number of nodes in the loop. Theoretically, the worst case may have exponential running time in the loop depth. However, in our experiments, the practical running time is linear in the size of the method body with a constant less than 3.

2.1.6 Revisiting the example

Now we revisit the example in Figure 2.2 with consideration of control-flow information. Figure 2.10 shows the program's control-flow graph of basic blocks. Note that each statement with array reference shows on the top of a basic block. The blocks are labeled from `A` to `G`.

First of all, we perform the array-related liveness analysis on the control-flow graph. The live-local set is marked before each basic block, in which the constant node of 0 is added. The optimal topological order of the CFG is (A, B, C, D, E, F, G). The VCA creates a constraint graph G_{uv} for each edge (u, v) in the CFG with the node set before block v. All graphs are initialized to $\perp$ except the block A's input graph G_{AA}, which is set to $\top$. The analysis iterates the blocks in their pseudo-topological order. But after visiting the block F, it will visit B instead of G since the block B is added in the worklist and it becomes the first one with higher priority than G.

Now we look at the flow-joint point at block B in detail. The first iteration over block B has only one initialized input graph G_{AB} in Figure 2.11(a). After going through blocks B, C, D, E, and F, G_{FB} was initialized as in Figure 2.11(b). The merged input graph G_B is same as G_{AB}. Now the flow analysis reaches the fixed point. In this example, $\delta(j, 0) = 1$ in G_{FB} although there is a statement $j = j - 1$ in block `F`. The reason is that, in block `E`, the reference $a[j]$ always produces constraint

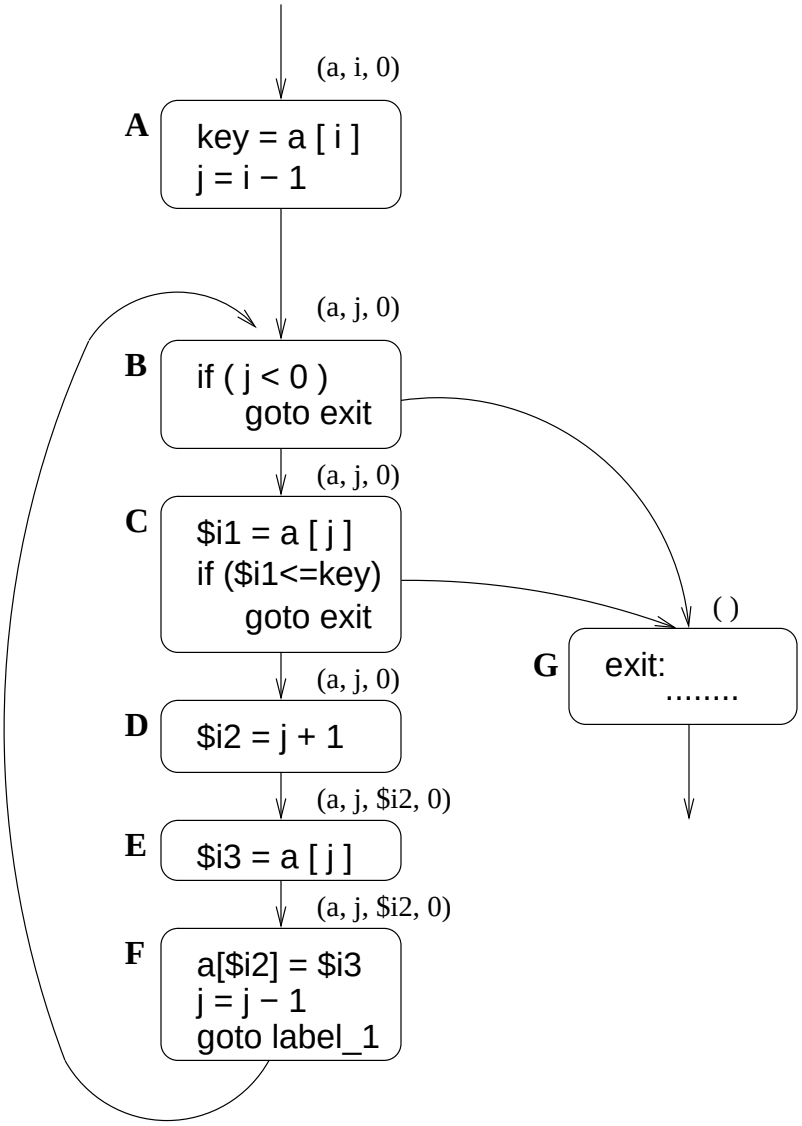
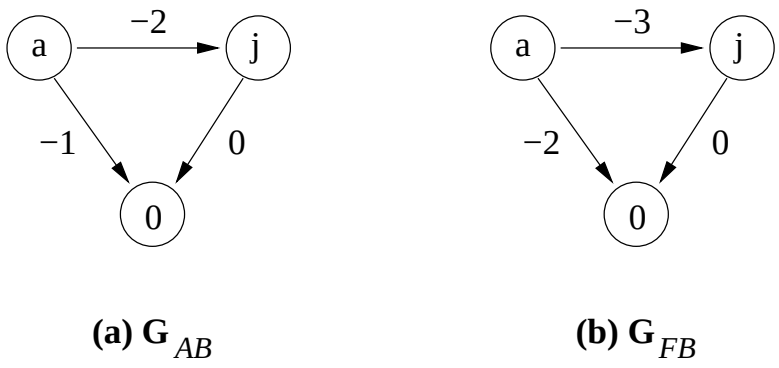


Figure 2.10: Control-flow graph of basic blocks

$0 - j \leq 0$ which may eliminate other paths of $\delta(j, 0) < 0$. At the fixed point, the input VCGs of block C, E, and F correctly give the shortest path weights: $\delta(a, j) = -2$ and $\delta(j, 0) = 0$ in G_C and G_E, $\delta(a, \$i2) = -1$ and $\delta(\$i2, 0) = -1$ in G_F. Thus, array references in these blocks were proved to be safe.



(a) G_{AB} **(b) G_{FB}**

Figure 2.11: VCGs of the block B

2.2 Array Field Analysis

The base analysis only looks at locals and analyzes the body of each method (intraprocedural). It does not know any information from outside of the method, such as fields or method parameters. There are no communications between methods. In Java applications, programmers may use fields to hold some constant value for code modularity and clarity. For example, some fields are initialized in constructors and are never changed again, or fields are assigned in some methods and used by others. To explore the full relationships of fields and on different methods is non-trivial, and needs whole program information. The analysis in our algorithm looks for special cases where a field holds a fixed length array object. This information allows us to extend the VCA analysis to include these fields.

A class field with modifier `final` or `private` can only be assigned a value in the class declaring that field. A `final` type field has more restrictions, it is assigned by a variable initializer in the source code. That means the assignment can only be in the constructors (<*clinit*> or <*init*>) of the declaring class. The *array field analysis* maintains a one-to-one map from classes to field information tables. For a class, each

array type field with the `private` or `final` modifier has an entry in the table, and a value is assigned to that field. The value can be $\perp$, an integer constant c, or $\top$. A field f declared in a class C is represented as $C.f$ no matter the f is static or non-static.

For each class C, *array field analysis* examines the class fields. Let F_C be the set of array-type fields modified by `private` or `final` declared in C. If F_C is non-empty, then a table τ_C is created, and for each $f \in F_C$ an entry $\tau_C[f]$ is created and initialized to $\perp$. Each method m declared in C is then considered. Since the Soot framework provides typed locals, and ensures that a `putfield` or `putstatic` is always in the form of an assignment from a local to a field, a simple pre-scan of the types of locals of m can be used to avoid further processing of methods that cannot change the value of any $f \in F_C$. For each method m that might change an array field, the body of m is scanned. Let $f = \ell$ be an assignment to some $f \in F_C$. A value $\delta(\ell)$ is computed as follows:

1. If ℓ is a `newarray` or `multianewarray` operation, then extract the array length expression d and return $\delta(d)$.
2. If ℓ is a local variable, the UD-DU chains provided by the Soot framework are used to locate the definitions of ℓ. If ℓ has more than one definition point, return $\top$, otherwise for a definition $\ell = x$ return $\delta(x)$.
3. If ℓ is an integer constant c, return c.
4. Otherwise, return $\top$.

Figure 2.12 is the pseudo-code for the process. The while loop ends when the length value is not BOTTOM ($\perp$). The table information $\tau_C[f]$ is then updated by merging the existing value for $\tau_C[f]$ with the computed $\delta(\ell)$ according to Table 2.4; note that $\delta(\ell)$ is never $\perp$.

	$\perp$	c1	$\top$
c2	c2	c1 : c1==c2 $\top$: otherwise	$\top$
$\top$	$\top$	$\top$	$\top$

Table 2.4: The rule for updating the field table.

When the intraprocedural VCA analysis meets an array type field read of the form *a=o.f;* where o has type of class C, it consults the array field analyzer to get the value

```
length = BOTTOM;
usestmt = currentStatement;
local = currentStatement.RHS;

while length is BOTTOM
{
    List defs = getDefsOfAt(local, usestmt);
    if (defs.size != 1)
    {
        length = TOP;
        break;
    }

    usestmt = (DefinitionStmt)defs.get(0);
    tmp_rhs = usestmt.getRHS;

    case tmp_rhs is a NewArrayExpression
    {
        size = tmp_rhs.getSize;
        case size is an integer constant
            length = size;
        case size is a local
            local = size;
        others
            length = TOP;
    }

    case tmp_rhs is an integer constant
        length = tmp_rhs;

    case tmp_rhs is a local
        local = tmp_rhs;

    others
        length = TOP;
}
```

Figure 2.12: Tracking down the array length.

associated to the field *C.f.* If the field has a constant value c, we can analyse this statement as if it was `a = new T[c]` (see rule in Table 2.3).

Our experience shows that this usually happens for a field with an initializer, where all assignments are made in the constructors. For simplicity, our implementation of array field analysis focuses only on the first dimension of array objects.

2.3 Rectangular Array Analysis

Another opportunity to improve VCA lies in rectangular arrays. Because multidimensional arrays in Java can be ragged, it is more difficult to get good array bounds analysis for multidimensional arrays. However, in scientific programs arrays are most often rectangular. Thus, we have developed a whole-program analysis using the call graph to identify rectangular arrays that are passed to methods as parameters.

Java defines a very loose structure for multidimensional arrays. A multidimensional array object can have a ragged shape (different rows in an array may have different lengths); sub-arrays can be sparse in memory or aliased; and array objects can be assigned to variables of type `java.lang.Object`. All of these properties make array bounds analysis hard. (recall the figure 1.3(b), which is an example of aliased sub-arrays.)

In order to find all arrays that are rectangular, we must find all cases where a rectangular array is allocated, and we must track those allocations to their eventual uses.

Consider the example in Figure 2.13, the `new_copy` method is taken from the `scimark2` benchmark. If we only analyze the method `new_copy`, it is not possible to say that all array references are safe because we do not know the array object passed to the parameter `A` are rectangular or not. However, if we know that the parameter `A` always holds rectangular arrays from all method calls, then we would be sure `N` equals to the length of any `A[i]`, which is the programmer's assumption. The *rectangular array analysis* tracks the array shape at each method calls of `new_copy`, and in this case can safely conclude that all method calls will pass a rectangular array to `new_copy`.

```
public class C
{
    public static void main(String[] args)
    {
        double[][] A = new double[10][9];
        double[][] B = new_copy(A);
    }

    protected static double[][] new_copy(double A[][])
    {
        int M = A.length;
        int N = A[0].length;

        double T[][] = new double[M][N];

        for (int i=0; i<M; i++)
        {
            int[] Ti = T[i];
            int[] Ai = A[i];

            for (int j=0; j<N; j++)
                Ti[j] = Ai[j];
        }
        return T;
    }
}
```

Figure 2.13: Rectangular array example.

2.3.1 Call graphs

In section 1.2.3, we mentioned that the Soot provides the call graph of an Java application. The call graph has one node for each method reachable from any starting method, which can be the `main` method of an application, or the `start` or `run` method of a runnable thread. The user can specify a set of starting methods. Each node (method) has a list of call sites, which are `invokestatic`, `invokespecial`, `invokevirtual` and `invokeinterface` bytecode instructions. The receiver of the `invokestatic` is resolved by the `javac` compiler and it has only one target. The

`invokespecial` has a fixed target also. For virtual method calls, `invokevirtual` and `invokeinterface`, the call graph provides a set of all possible targets. The edges of the graph connect each call site to its possible target methods. More details about call graphs can be found in [31].

An algorithm based on the call graph is a conservative approximation because it does not know the exact call target which is resolved at the run-time. If a method is reachable, all targets of its call sites must be marked as reachable. Our *rectangular array analysis* builds an array type graph based on the call graph. For each reachable method, it first recovers the rectangular array initializer as explained in section 2.3.2. It then constructs a propagation graph where nodes consist of locals, method parameters, and method returns. Edges are then added between nodes when values are passed, such as assignments and method calls. Creation sites for rectangular arrays are marked as **TRUE**. If a nodes changes shape it is marked as **FALSE**. All nodes reachable from **FALSE** nodes are marked as **FALSE**. The remaining nodes reachable from **TRUE** nodes are marked as **TRUE**. Nodes marked with **TRUE** after the analysis represent variables referring to a rectangular arrays.

2.3.2 Recover array initializers

Before constructing the array type graph, we have to look at some special cases. If a programmer allocates a new multidimensional array using a statement of the form `new int[10][10]`, this instruction is translated into a `multianewarray` bytecode instruction which allocates rectangular arrays. However, a multidimensional array initializer is compiled by *javac* or *jikes* as individual allocations to give a potentially ragged array of array objects. An array of arrays is created, then each element is assigned a sub-array object. Figure 2.14(a) shows a typical Java example, and Figure 2.14(b) shows the resulting bytecode.

We use a simple pattern matcher that can find this idiom and recover a rectangular array's creation from its sparse representation to a dense one, as shown in Figure 2.14(c). The pattern matcher is a state machine which identifies the patterns as in Figure2.14(b). Table 2.5 gives a simplified state table for identifying two-dimensional arrays, which is the current implementation.

The input of the state machine is a sequence of JIMPLE instructions of a method. The start state 0 accepts a statement of `r1 = new (A[])[c];`. We briefly describe the operations at each state:

```
int[][] a = {{1},
             {2}};
```

a) An array initializer

```
a = newarray (int[])[2];
$r2 = newarray (int)[1];
$r2[0] = 1;
a[0] = $r2;
$r3 = newarray (int)[1];
$r3[0] = 2;
a[1] = $r3;
```

b) Compiled code by javac and jikes

```
a = multianewarray
    int[2][1];
$r2 = a[0];
$r2[0] = 1;
$r3 = a[1];
$r3[0] = 2;
```

c) Recovered code

Figure 2.14: Recover the creation of rectangular arrays

state	input	goto
0	r1 = new (A[])[c]	1
1	r2 = new A[d]	2
2	r2[*] = $\cdots$	2
	r1[e] = r2 (e=c-1)	3
	r1[e] = r2 (e=$e^{'}$+1)	1
3	end	

Table 2.5: The state machine for matching two-dimensional arrays.

State 0 records the base type A, the length c, and the left hand side variable *r1*.

State 1 accepts a statement of array creation. The base type is checked with the recorded type A in state 0, the sub-array *r2* and the length d are recorded.

State 2 goes to different states according to the input statement. It could be the initialization of the sub-array *r2*, in which case, it will continue on state 2. Or it is a store to the first dimension of the array object *r1*, the array index e is checked with the array length c. It also ensures the reference index is incremental by 1 ($e = e^{'} + 1$) if it does not reach the array length.

State 3 returns the length of the second dimension d if the pattern is matched, otherwise it returns -1.

For any exceptional inputs, the state machine jumps to the state 3 and returns -1.

2.3.3 Array type graphs

After finding all the creation sites for rectangular arrays, we then build an array type propagation graph to find which variables must be associated with rectangular arrays. The graph has following nodes:

1. Two special nodes for **TRUE** and **FALSE**. Marking another node is achieved by adding an edge between it and one of the special nodes.

2. Method locals that are multidimensional arrays. Consider the example in Figure 2.13, the method `new_copy` in the class `C` has a local `M`. The local `M` is represented as *C.new_copy.M*.

3. Method parameters whose types are multidimensional arrays. The parameters are handled in the same way as locals. The parameter `A` in the example is represented by *C.new_copy.A*.

4. Method returns whose types are multidimensional arrays. In our example, the return of method `new_copy` is represented as *C.new_copy.return*.

5. Class fields. As in `array field analysis`, an array type field `f` of the class `C` is represented as *C.f* whether `f` is static or non-static.

Then we define rules to add edges to the graph according to the types of the statements. In general, assignment statements and invoke expressions add edges between nodes in the graph. Some special cases will add edges between normal nodes and the special nodes **TRUE** or **FALSE**. Only multidimensional array type variables are considered in this analysis. In following rules, lower-case letters are locals, and by default, they are referred in a method *C.m*.

1. $a = newA[i][j]$
   This is a site that creates a rectangular array. We add an edge between *C.m.a* and the special node **TRUE**.

2. $a = b$
   For a general assignment, we add an edge between nodes *C.m.a* and *C.m.b*. The `b` is either a local or a parameter.

3. $a[i] = b$
   If `a` is a multidimensional array type local, a store into it adds an edge between $C.m.a$ and the special node **FALSE**.

4. $o.n(a, b, ...)$
   An invocation expression needs more explanation. Let C_n be the set of possible receiver classes of this call site, and *p0, p1, ...* be the parameters of the method `n`. For each C' of C_n, we add edges between $C.m.a$ and $C'.n.p0$, $C.m.b$ and $C'.n.p1$, and so on.

5. $a = o.n(...)$
   An assignment from a method return adds edges between $C.m.a$ and the return of each possible target, $C'.n.return$.

6. $return\ a$
   A return expression adds edges between $C.m.a$ and $C.m.return$.

7. $t.f = a$ or $a = t.f$
   Field references add edges between $C.m.a$ and $T.f$ where the class `T` declares the field `f`.

8. $a = (A)b$
   For the assignment with a cast expression, we check the static type of `a` and `b`. If both locals are multidimensional arrays and have the same dimension number, the statement is treated as a normal assignment $a = b$, otherwise, $C.m.a$ and $C.m.b$ are connected to the **FALSE** node. This is a conservative approach to reduce the complexity of the analysis because array types can be casted from and to `java.lang.Object` in Java.

If a local gets a return value from a method which is out of our analysis context (i.e. we only analyze the application code without library code), we make a conservative assumption and connect the variable to the **FALSE** node. Parameters of the method invocation are treated in the same way. Figure 2.15 gives the propagation graph of the example in Figure 2.13.

After building the propagation graph, we want to find all nodes which are reached starting at the **TRUE** node (were allocated as rectangular), and are **not** reached starting at the **FALSE** node (may have become ragged). We achieve this as follows: first we traverse the graph, starting from the **FALSE** node, marking these nodes as

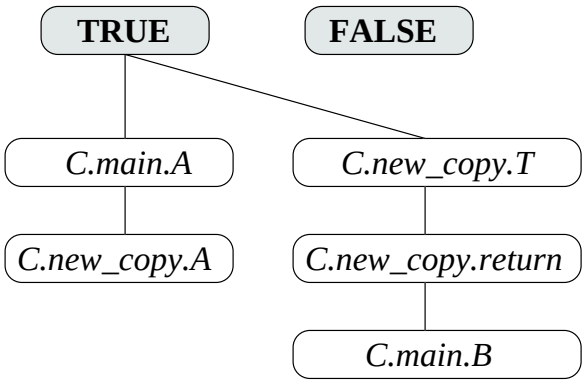


Figure 2.15: Propagation graph

reachable from **FALSE**. Then we traverse the graph starting at the **TRUE** node, finding all reachable nodes that are not marked **FALSE**. This set indicates that the members are always assigned rectangular arrays. The pseudo-code is listed in Figure 2.16.

To use rectangular array information, the constraint graph has some special nodes to represent the sub-arrays. In our rectangular example (figure 2.13), a special node $A[$ is used to represent the second dimension length of A. When the VCA meets a statement of $a = A[i]$ and A is a multidimensional array, it checks the true node set generated by the *rectangular array analysis*. When the node is in the true set, directed edges are added between node a and $A[$. In the example, since the VCA analysis will determine that local variable N is equal to $A[$, it is possible to determine that all array references are safe in the program.

2.4 Other Enhancements

Besides the multidimensional arrays, the variable constraint graph can be extended to accommodate some extra nodes, such as class fields and array references. We have done this in a very conservative way, assuming the worst-case aliasing and side-effect information. With these conservative assumptions we did not find much improvement in the result. More accurate side-effect information may improve the situation.

```
Set startNodes = successors of FALSE node
add startNodes to falseSet
add startNodes to workList

while workList is not empty
    node = workList.removeFirst
    Set succs = successors of node
    for each succ in succs
        if falseSet does not contain succ
            add succ to falseSet
            add succ to workList
```

(a) marking FALSE nodes

```
Set startNodes = successors of TRUE node
for each node of startNodes
    if falseSet does not contain node
        add node to trueSet
        add node to workList

while worklist is not empty
    node = workList.removeFirst
    Set succs = successors of node
    for each succ in succs
        if falseSet does not contain node
        and trueSet does not contain node
            add node to trueSet
            add node to workList
```

(b) marking TRUE nodes

Figure 2.16: Traverse the graph.

We did following extension to our intraprocedural algorithm. In the liveness analysis, we also add fields, array elements, and common sub expressions as locals to the live local sets. For example, $a.f$, $a[i]$, and $i * j$ can be added into the live local sets and the constraint graph can add edges connecting them to other nodes.

But it should be conservative when dealing with an assignment to a field or array element since we do not have alias information. Detailed operations are:

***a* = ⋯**

If a is an array type local, all array elements of $a[*]$ should be killed. If it is a reference type local, all fields of $a.f$ should be killed.

***a[i]* = ⋯**

Since we do not know any alias information, all array reference nodes should be killed. However, if we use the type information of a, we only need to kill the same type arrays' elements.

***i* = ⋯**

When i is an integer variable. Array elements of $*[i]$ should be killed, and all expressions containing i, such as $i * j$, should be killed.

***a.f* = ⋯**

Fields of $*.f$ should be killed. Because the declaring class of a field is resolved by the compiler, f in this statement should be understood as $T.f$ where T is its declaring class, rather to be interpreted as a symbolic name f.

m(a)

When an array or reference type local is passed to a method, all related fields and array elements should be killed since we do not know the alias information and the side effect of the method call.

a.m()

A virtual method call passes the caller as the first parameter to the callee implicitly, then it has to take the same action as $m(a)$.

In our experiment, the enhancements increased the constraint graph size dramatically, but the results has very few improvements. In Java applications, method invocations happen very often, thus the life time of a field in the graph is very short. Basically fields get killed again and again. The same situation happens to array

elements. The side-effect analysis and alias analysis may help us to make less conservative assumptions when dealing with assignments and method calls.

2.5 Null Pointer Analysis

Eliminating array bounds checks is often related to eliminating null pointer checks. Each array reference, for example `a[i]`, must first check that the array object referenced by `a` is non-null. In many modern compilers null pointer checks are performed by handling the associated hardware trap if a null pointer is dereferenced. In this case the machine architecture guarantees a hardware exception if any very low memory addresses are read or written. In order to do the upper array bounds check the length of the array must be accessed, and since the length of the array is usually stored at a small offset from the object address, this access will trap if `a` is null. Thus, the array bounds check gives a null pointer check for free. If the array bounds check is eliminated, then it may be necessary to insert an explicit null pointer check (since the address of `a[i]` may be sufficiently large to avoid the null pointer trap, even if `a` is null).

Our nullness analysis is a fairly straightforward flow-sensitive intraprocedural analysis that is implemented as an extension of the `BranchedForwardFlowAnalysis` class that is part of the Soot API. The basic idea is that variable `a` is non-null after statements of the form `a = new T();` and statements that refer to `a.f` or `a[i]`. We also infer nullness information from condition checks of the form `if (a == null)`. Since the nullness analysis is intraprocedural we make conservative assumptions about the effect of method calls.

Chapter 3

Experimental Results

We have implemented the algorithm in the context of the Soot framework[1]. In this chapter we present and discuss the experimental results that we have obtained. The results are grouped into three categories:

1. We measured the dynamic characteristics of the *variable constraint analysis* in terms of two most important factors affecting the algorithm's performance: the *size* of variable constraint graphs and the *number of iterated blocks* to reach the fixed point.

2. In section 3.5, we show the results of the base intraprocedural analysis, followed by the *array field analysis* and *rectangular array analysis* as they are added in separately, and finally combined. The results are presented as percentages of lower and upper bound checks that can be proved safe.

3. Our analyses results are encoded in the attributes of class files. To measure the real impact to the run-time performance of Java programs, we modified Kaffe JIT and HPCJ compiler to read and take advantages of such attributes. The run-time measurements show speed-ups in most of benchmarks.

In section 3.1, we briefly introduce the implementation of array bounds checks in a JVM at first, which often interleaves with the null pointer checks. Also we describe the experimental environment and methodologies. Then we show the static and dynamic characteristics of benchmarks. We measured two important factors of

[1] A brief overview of the code organization is given in Appendix A

the analysis, which show the algorithm runs in linear time with respect to the size of the method body. Finally, we describe in detail how to define the structure of the array bounds check attributes and make a VM take advantage such attributes.

3.1 Experimental Method

Our algorithm is implemented in the Soot framework as an independent package which can be found in `soot.jimple.toolkits.annotation.arraybounds`. A wrapper is created to let the Soot main method call the analysis according the command options. In this section, we introduce our profiling methodology used in our experiment, and the hardware and software environment in which the experiment is conducted.

To measure the characteristics of benchmarks and the results of the analysis, we need a profiler to tell us the run-time results. This was done by inserting instructions increasing an integer counter before each bytecode which requires array bounds check or null pointer check.

The experiment was conducted on two environments. The first one uses Kaffe open VM 1.05 with JIT engine 3 running on a dual Pentium II 400M PC with 384M memory, Linux OS kernel 2.2.8, and glibc-2.1.3. We measured the benchmark characteristics and profiling information on Kaffe VM. We also modified the Kaffe JIT compiler to take advantage of attributes and compared the results with no attributes. The second part of experiment is conducted on IBM's High Performance Compiler for Java (HPCJ), which runs on a Pentium III 500M PC with 192M memory, Windows NT operating system. The HPCJ ahead-of-time compiler understands the attributes and generate improved code for the benchmark class files. We measured the performance changes with/without attributes.

3.2 Benchmarks

We chose several benchmarks including both general and numerical ones: as well as SpecJVM and scimark2, *LCS*, an implementation of a Longest Common Subsequence algorithm, and *MCO*, an algorithm for finding an optimal order of matrix multiplication. Here a brief description of each of the benchmarks is presented (the description of first five benchmarks comes from [29]).

db : The db benchmark performs multiple database functions on memory resident database. It reads in a 1 MB file which contains records with names, addresses and phone numbers of entities and a 19KB file called scr6 which contains a stream of operations to perform on the records in the file.

jack : Jack is a Java parser generator. The workload consists of a file named jack.jack, which contains instructions for the generation of jack itself. This is fed to jack so that the parser generates itself multiple times.

javac : This is the Java compiler from the JDK 1.0.2.

mpegaudio : This is an application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification. From our experiments, we know this benchmark uses arrays heavily.

raytrace : This is a raytracer that works on a scene depicting a dinosaur.

scimark2 : SciMark 2.0 is a Java benchmark for scientific and numerical computing. It measures several computational kernels which include FFT, SOR, LU matrix factorization, Monte Carlo integration, and Sparse matrix multiply. In our experiment, we measured the run-time improvement on the first three kernels since the algorithm can prove most of their array references safe.

MCO This is an algorithm computing the *matrix-chain multiplication problem*. The function name is called `Matrix-Chain-Order` (see [6](p.306)).

LCS This algorithm finds a maximum-length common subsequence of two sequences. Both of MCO and LCS algorithm use two-dimensional arrays as main data structures.

The benchmarks are characterized by their size, array reference density, and the run-time overhead caused by array bounds checks. Table 3.1 shows benchmark characteristics. All numbers are collected from benchmark code (excluding the class libraries). The third column describes the size of benchmark as the number of bytecodes of class files in the package. FFT, LU, and SOR are packaged together in "scimark2". They share some common classes, the total size of the "scimark2" package is showed in the cell. The last two columns, density and overhead, show dynamic measurements of the benchmarks. The problem size of benchmarks from "SpecJVM98" are set as 100. The execution of benchmarks from "scimark2" is specified as "LARGE". LCS

and MCO both have loop size of 3000, which makes the benchmarks run long enough to reduce the effect of VM initialization. The density is a count of how many array references per second occur in the benchmark (not including class libraries). It is a rough estimate of the potential benefit of array bounds check elimination. The last column shows the overhead caused by array bounds check instructions. To measure the overhead, we modified Kaffe JIT to turn off generating bounds check instructions for benchmark code, then compare the time without checks against with checks.

name	source	#bytecode	density	overhead
db	SpecJVM98	14526	1,074,979/s	0.4%
jack	SpecJVM98	31604	29,962/s	1.1%
javac	SpecJVM98	54897	73,861/s	3.8%
mpegaudio	SpecJVM98	47265	19,531,665/s	22.3%
raytrace	SpecJVM98	19359	1,054,832/s	1.7%
FFT	scimark2		8,667,594/s	5.1%
LU	scimark2	2303	23,120,315/s	-0.9%
SOR	scimark2		14,528,328/s	11.3%
LCS		255	58,384,589/s	13.9%
MCO		418	33,659,647/s	15.1%

Table 3.1: Characteristics of the benchmarks

The Spec benchmarks are relatively large, while the other five benchmarks are relatively small. From the density of array references and the run-time overhead of bounds checks, we can see 'mpegaudio' has a large overhead, as do LCS, MCO and three sub-benchmarks in scimark2. (The LU benchmark exhibits a negative overhead, which is probably due to the impact of instruction caches after we removed bounds check instructions, we also find such impact in later experiments.) These benchmarks are all typical examples of array-intensive programs. Other benchmarks in our study serve as examples of normal programs which are less array intensive, but also reflect the dynamic characteristics of the algorithm in the next section.

3.3 Dynamic characteristics of the algorithm

As we analyzed in the section 2.1.5, the theoretical upper bound of the *variable constraint analysis* can be exponential. To understand the real cost of the algorithm,

we chose to measure two factors: the constraint graph size and the number of blocks iterated by the worklist algorithm.

Table 3.2 shows some of the dynamic properties of our algorithm applied to the different benchmarks. The *Blocks* column gives the number of basic blocks in the program, while the *NonZero Blocks* column gives the number of blocks that have non-empty live sets for local variables, and so non-empty constraint graphs. Only NonZero blocks were used in the calculation of average and maximum constraint graph sizes, and every (non-empty) constraint graph includes at least one node for the constant zero. From this, the size of the constraint graphs is quite reasonable: average size never exceeds 10 nodes, and maximum size no more than 13. These are quite practical factors.

Name	Graph size		Blocks	Iter	NonZero
	(avg)	(max)		(avg)	Blocks
db	3.17	6	280	1.28	89
jack	2.5	6	2076	1.04	1892
javac	2.45	6	3347	1.27	1631
mpegaudio	3.42	10	6987	1.10	6670
raytrace	2.56	6	626	1.31	476
scimark2	5.8	12	388	1.79	301
LCS	9	13	59	2.8	55
MCO	4.6	11	98	2.0	95

Table 3.2: Characteristics of the algorithm

The *Iter* column is the average number of times a block is processed as the analysis iterates toward a fixed point. It is a good indicator how long the analysis will run, and suggests that in a practical sense the running time of our algorithm is linear in the code size. There is an impact due to loop nesting; in small benchmarks, LCS, MCO and scimark2, the code bodies are dominated by nested loops and hence, the factor is higher than other benchmarks. Nevertheless, the factor remains relatively small overall.

3.4 Array Bounds Check Attributes

After the analysis phase the flow information is associated with JIMPLE statements. The next step is to propagate this information so that it will be embedded in the class file attributes. This is done by first tagging the JIMPLE statements, and then specifying a tag aggregator which packs all the tags for a method into one aggregated tag. The process of tagging/attributing is described in [23].

We first outline the attribute as it eventually appears in the generated class file. The structure of the array bounds attribute is quite straightforward. It has the name `"ArrayNullCheckAttribute"`. Figure 3.1 shows the format of the array bounds check attribute as it will be generated for the class files.

```
array_null_check_attribute
{
    u2 attribute_name_index;
    u4 attribute_length;
    u3 attribute[attribute_length/3];
}
```

Figure 3.1: Array Bounds Check Attribute

The value of `attribute_name_index` is an index into the class file's constant pool. The corresponding entry at that index is a `CONSTANT_Utf8` string representing the name `"ArrayNullCheckAttribute"`. The value of `attribute_length` is the length of the attribute data, excluding the initial six bytes. The `attribute[]` field is a table that holds the array bound check information. The `attribute_length` is 3 times larger than the table size. Each entry consists of a PC (the first two bytes) and the attribute data (last one byte), totaling three bytes. These pairs are sorted in the table by ascending PC value.

The least two bits of the attribute data are used to flag the safety for the two array bounds checks. The bit is set to 1 if the check is needed. The null check information is incorporated into the array bounds check attribute. The third lowest bit is used to represent the null check info. Other bits are unused and are set to zero. The array reference is non-null and the bounds checks are safe only when the value of the attribute is zero.

After generating the annotated class file, we need to make a JVM aware of attributes and have it use them to improve its generated native code. We modified both Kaffe's OpenVM 1.0.5 JIT and IBM's HPCJ ahead-of-time compiler to take advantage of the array bound attributes. Below we describe the modifications needed for Kaffe. The modifications to HPCJ are similar.

The KaffeVM JIT reads in class files, verifies them, and produces native code on demand. It uses the '_methods' structure to hold method information. We added a field to the '_methods' structure to hold the array bounds check attribute. Figure 3.2 shows the data structure.

```
typedef struct _methods {
    ....
    ....
    soot_attr attrTable;
} methods;

typedef struct _soot_attr{
    u2 size;
    soot_attr_entry*  entries;
} soot_attr;

typedef struct _soot_attr_entry {
    u2 pc;
    u1 attribute;
} soot_attr_entry;
```

Figure 3.2: Modified Kaffe Internal Structure

When the VM reads in the array bounds check attribute of the Code attribute, it allocates memory for the attribute. The <PC, data> pairs are then stored in the attribute table. The pairs were already sorted by PC when written into the class file, so no sorting has to be done now.

The Kaffe JIT uses a large switch statement to generate native code for bytecodes. It goes through the bytecodes sequentially. We use the current PC as the key to look up the array bounds check attribute in the table before generating code for array references. Because attribute pairs are sorted by ascending PC, and bytecodes are

processed sequentially, we can use an index to keep the current entry in the attribute table and use it to find the next entry instead of searching the whole table. Figure 3.3 gives the pseudo-code.

```
idx = 0;
...
case IALOAD:
  ...
  if (attr_table_size > 0) {
    /* the method has attributes. */
    attr = entries[idx].attribute;
    idx++;
    if (attr & 0x03)
        /* generates bounds check instr. */
        check_array_index(..);
    else
    if (attr & 0x04)
        /* null pointer check instr. */
        explicit_check_null(..);
  }
  else
    /* normal path */
    check_array_index(..);
```

Figure 3.3: Using attributes in KaffeVM

In section 2.5, we discussed the subtle relationship between array bounds check and null pointer check for an array references. Here, we turn off bounds check instructions when the array reference is non-null and both bounds are safe. We also insert null check instructions at the place where bounds check instructions can be removed but the null check is still needed. The `check_array_index` function emits following code for checking array bounds:

```
cmp reg1, [reg2+off]
jge outofboundserror
```

and the `explicit_check_null` generates instructions for checking null pointers:

```
cmp reg1, 0
je  nullpointerexception
```

HPCJ uses a slightly different scheme to handle bounds checks. If array bounds checks are required, a test-and-branch code sequence is inserted prior to the array access :

```
mov eax,[ebx+offset]
cmp eax,edx
jge outofboundserror
```

When only bounds checks are proved to unneed, the null pointer check is accomplished by a test instruction:

```
test eax,[eax]
```

The reason for using different check instructions in two experiments is that we utilized existing routines in the two systems.

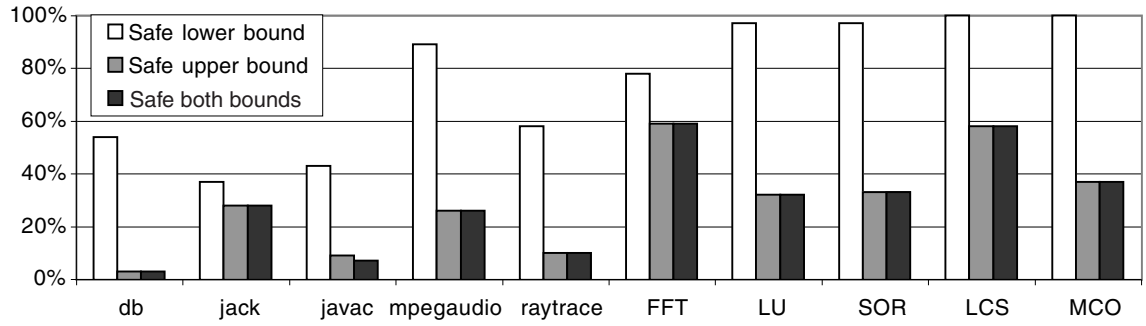
3.5 Dynamic Results and Discussion

Figure 3.4(a) shows the percentage of bounds checks that our basic intraprocedural analysis is able to detect as safe to remove. Note that these are dynamic statistics, obtained by instrumenting the class files and inserting profiling instructions before each array reference bytecode. Lower bounds and upper bounds are measured separately in the first two bars for each benchmark, while the last bar gives the percentage of array references with both safe checks.

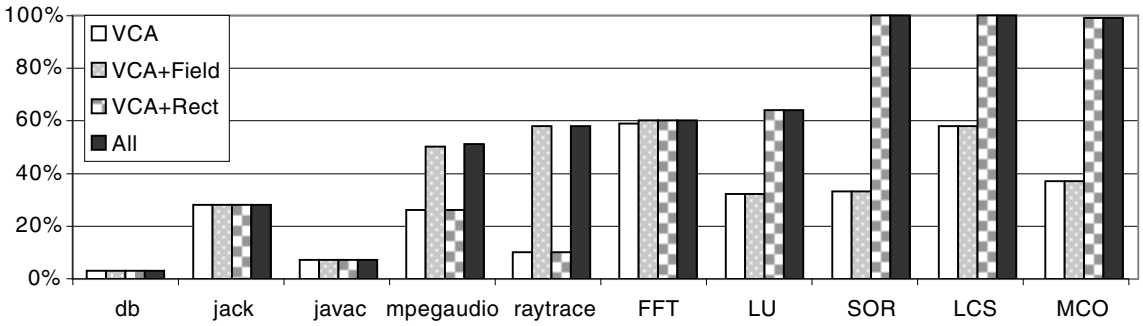
The intraprocedural algorithm can determine that a fairly high percentage of the lower bound checks are safe. Safety of upper bound checks is more difficult to ascertain. Still, the results for the array-intensive benchmarks (rightmost five) are encouraging; these are the benchmarks which will benefit the most, and also in which we achieve the best results.

Figure 3.4(b) gives the percentage of cases where both upper and lower bounds checks could be determined to be safe. The second and third bars are from the basic intraprocedural algorithm augmented with either array field analysis or rectangular array analysis; the last bar represents the intraprocedural algorithm with both array field and rectangular array analyses.

By analyzing the fields holding constant length array objects, the intraprocedural analysis can get more information about field accesses. The success of this method,



(a) Results of the base analysis



(b) Improvements due to field and shape analyses (both bounds safe)

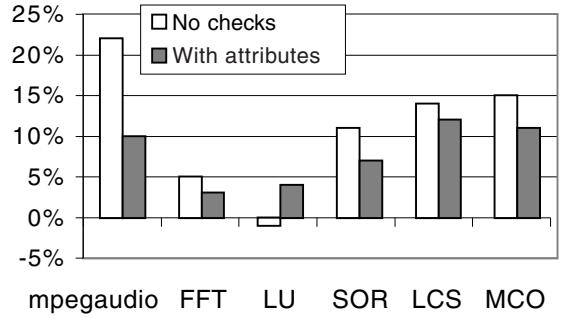
Figure 3.4: Dynamic Results of VCA

however, depends on the application: 'mpegaudio' and 'raytrace' improve greatly, while others are more or less unaffected (Figure 3.4(b)). Rectangular array analysis also proves to be very application-dependent. It is of benefit only to those benchmarks using multidimensional arrays. LU, SOR, and LCS and MCO improve dramatically with the addition of this analysis.

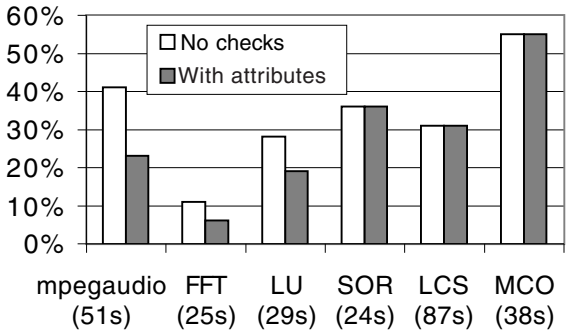
The last experiment shows the result of the combined use of field and rectangular analyses. Because these are essentially independent analyses, the combined improvement is close to the sum of the improvements seen individually. With most of our benchmarks this brings the percentage of checks we could eliminate to 50% or more; again, array-intensive benchmarks fare best, and in some cases we identify almost 100% of array bounds checks as safe.

Relative run-time performance improvements for the instrumented versions of the Kaffe JIT and HPCJ are given in Figure 3.5. Both systems were modified to read the array attribute information stored within the class file and to apply that data during

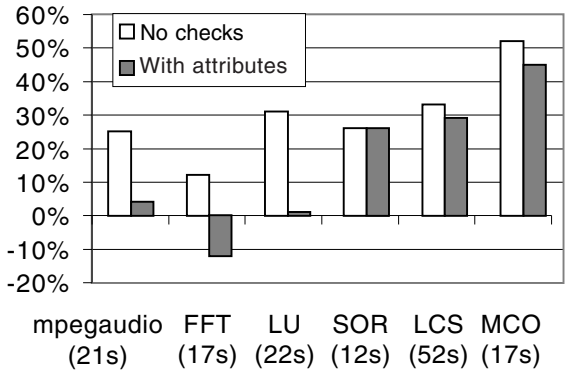
code generation.



(a) Kaffe



(b) HPJC (other optimizations off)



(c) HPJC (other optimizations on)

Figure 3.5: Speed-ups for Kaffe and HPCJ

If an array access is deemed safe from the attribute information, no such checks are created—this is done during actual (just-in-time) code generation for Kaffe, and at an internal, intermediate stage for HPCJ. In the latter case, this eliminates the

potential array bounds exception that may restrict subsequent internal optimizations, resulting in different code output. For this reason we present results with and without HPCJ's own optimizations applied.

Finally, note that every array access is an object access, and so null pointer checks are also required at these points. Depending on machine architecture and how objects are organized, this check can be combined with the array bounds check, and so removing the latter may require inserting explicit null pointer checks [23]. Best performance results therefore occur when both kinds of checks are eliminated. Our results include this optimization.

In each case the result of using the intraprocedural analysis combined with both field and rectangular analyses is compared with the effect of artificially disabling all bounds checks. A couple of cases (LU in Kaffe, FFT in HPCJ (opt)) exhibit interesting anomalous results that we have been able to attribute to code cache effects. In all other cases, however, we achieve significant performance increases, roughly corresponding to the quality of information we were able to collect.

Chapter 4

Related Work

Array bounds check optimization has been performed for other languages, such as Pascal, Fortran, and Ada[22], for a long time. We first discuss some related work developed on other languages. These algorithms can not be directly applied to Java programs because of its unique requirement of precise exceptions (Ada shares this same property). However, another unique property of Java is that multidimensional arrays are defined as array of arrays, which prevents many existing methods from applying on Java. New solutions have been emerging since the introduction of Java. We will discuss more details in the following text.

W.H.Harrison[14] described an algorithm for value ranges analysis. The algorithm consists of two mechanisms called *range propagation* and *range analysis*. Range propagation uses the data and the conditional structure of a program to derive and propagate symbolic range information. Targeting complex control flow structures (loops), range analysis tracks the changes applied to a variable at each point in a loop of the program. The information is used to derive a range of values for the loop variable. The resulting range information can be used to eliminate unnecessary tests and produce diagnostic information. While this was a novel idea to reduce redundant tests at that time, the simple mechanical propagation of symbolic value can only prove a small part of safe checks.

The problem of run-time overhead of array bounds checks was first addressed by Markstein et. al. [18]. R. Gupta[12, 13] extended their work by using data-flow analysis to eliminate redundant checks, propagate checks out of loops, and combine multiple checks into a single check. The algorithm has the same principle as partial redundancy elimination. It relies on hoisting check instructions to the earlier point.

Several kinds of checks can be subsumed: identical checks, checks with identical bounds, and checks with identical subscript expressions. Kolte et.al. [16] extended Gupta's algorithm in a partial redundancy elimination framework. A fundamental assumption of the algorithms is that the exception can be thrown at the point before original exception point (remember that Figure 1.2(b) showed such an example). This assumption is acceptable when working on languages that do not require precise exceptions. Java does not allow an exception to happen before the place it really should be. However, a more basic problem with this type of algorithm is that the language should be able to express and modify checks explicitly, where bytecode instructions can not do that.

There are several algorithms targeting different problems involved in removing bounds check overhead for Java. Scientific computing programs use multidimensional arrays. Because of Java's loose multidimensional array structure, it is very hard to optimize such programs. Moreira et.al. [21, 19, 20] designed an `Array` package for two- and three-dimensional arrays. The package provides Fortran 90-like array functionality (all array operations are performed through method calls). Internally, a multidimensional array is implemented by a one-dimensional array. To achieve good performance, an inlining technique is used to reduce the overhead caused by method calls, and a special regioning or loop-versioning technique is used to create safe regions for array accesses, and thus, remove unneeded array bounds checks. The algorithm only works on loops and relies on underlying virtual machine to be aware of the Array package and perform unusual optimizations on it.

Some JIT compilers perform array bounds check elimination when translating bytecode to native code. The Intel JIT[5] performs analysis to approximate the range that an array might access within a loop. In the case of a known range, a special check-free loop body is created, while the bounds check code is inserted outside the loop. The IBM JIT[30] uses the same technique called loop versioning, but also has a data-flow analysis to analyze checks not in a loop. The data-flow analysis is an extension of Gupta's algorithm. Both of two compilers have to obey the precise exception requirement of Java. A basic policy is to not moving checks over any bytecode which has side-effect (e.g., memory access, bytecode may cause other exceptions). Loop versioning also can cause code explosion. So the application of the optimization is limited by some parameters: the code size of loop body, the innermost loops, and so on.

More recently, Bodik et. al. [3] presented an algorithm called ABCD (Eliminating Array Bounds Checks on Demand) for general Java applications, The algorithm uses

a different form of constraint graphs to solve bounds checks. The algorithm first splits locals' definitions and uses according the value range constraints. It builds an extended SSA (static single assignment) form for a method body. In this e-SSA form, all uses of a variable would have the same value range which can be derived from the program. For example, assignments can change a variable's value range as in ordinary SSA form, and array references and conditional branches can also bound the value range of the index or condition variables in the scope after them. Thus, these statements are treated as assignments in the SSA algorithm. The e-SSA guarantees that all uses (by name) of a variable are bounded by the same constraints, the value range, at the run-time. The value range could be an approximation. Based on the new form, a constraint graph is constructed, where nodes are locals and constants, and weighted edges are constraints representing inequality relationship between nodes. To infer the relationship between array and index, the shortest path between them is solved by a customized depth first search algorithm which specially handles the ϕ nodes in the graph. If the shortest path length is less than zero, the upper bound check for that array reference is unneeded. The lower bound can be eliminated if the weight of the shortest path from array index to the node of constant 0 is greater or equal to 0. At each control flow joint point (ϕ node), the `weakest` constraint has to be taken.

Our VCA shares some similarities with theirs, both are using inequality graphs to represent constraints. However, there are several differences between our algorithm and ABCD approach:

1. The ABCD algorithm is based on an extended SSA form, and uses one graph to summarize constraints from all statements in a method. Thus, the control-flow information is included in the constraint graph. Our VCA approach does not rely on any underlying program representation form, it uses a fixed number of small program-point specific constraint graphs.

2. Based on e-SSA form, the ABCD algorithm can be used in a demand-driven manner. Each demand (query) is solved individually, and may be performed on selected array references that occur in hot spots. Each query is relatively inexpensive. The VCA approach is designed to prove all array references at once. It builds constraint graphs and solves constraints in relatively expensive costs, but the results are available for all array references immediately.

3. The VCA approach keeps constraints of lower and upper bounds in the same graph, which is not the case in the ABCD approach.

4. ABCD is capable of catching partial redundant bounds checks. VCA is not able to do that currently.

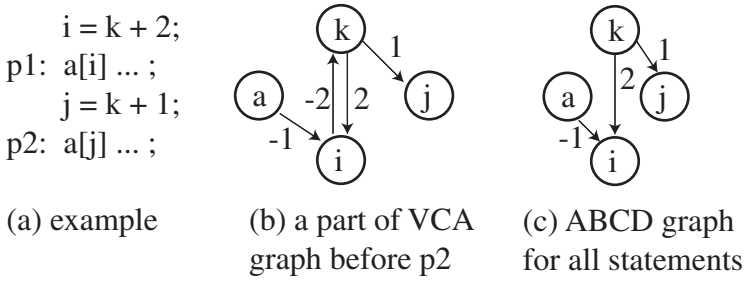
5. In some cases, a program-point specific graph can hold some implicit constraints where a summary graph based on a SSA representation form cannot. Figure 4.1 illustrates this point. Given the program segment in Figure 4.1(a), our VCA algorithm builds the constraint graph shown in Figure 4.1(b), whereas the ABCD algorithm builds the graph shown in Figure 4.1(c). Note that in the ABCD graph, constraints are only encoded along the direction of the control flow (for example, the assignment `i = k + 2;` results only in one edge, from `k` to `i`). Given this graph, it is not possible to find the safe upper bound at $p2$. However, since VCA collects a separate graph for $p2$, and the constraint gained from $p1$ is also applied to i and q, it is possible to show that the bounds are safe at program point $p2$.

6. In our algorithm, the constraint graph serves as the basis of other two analyses. We can see, for certain type applications, the impacts of the analyses are significant. Currently it is not clear how class fields and multidimensional arrays information can be used to help the ABCD algorithm.

VCA may not be faster than the ABCD algorithm, although the techniques we used make our algorithm run at a reasonable speed. In some SPEC JVM98 benchmarks, VCA can prove nearly same percentages of safe upper bound checks as reported in [3]. With *array field analysis* and *rectangular array analysis*, VCA can outperform ABCD significantly. Experiments show that VCA with *rectangular array analysis* is very effective on micro benchmarks using two-dimensional arrays. We also think the approach of formulating a problem in constraint graphs and solving it by using data-flow analysis can be useful for other problems.

The general idea of using the single-source shortest-path of an inequality graph to solve systems of difference constraints has been stated in [6](p.539-p.545).

R. Shaham et. al. [27, 26] described an algorithm for identifying live regions of arrays to detect array memory leaks in Java. In their work, the representation and analysis are very similar as our VCA. Constraint graphs and data-flow analyses are used to compute inequalities between variables. However, their focus is on finding relationships between special class fields across method boundaries based on supergraphs of a few particular library classes. Although the supergraph can make our *field analysis* more powerful, our VCA approach focuses on intraprocedural analysis



(a) example (b) a part of VCA graph before p2 (c) ABCD graph for all statements

Figure 4.1: Comparing the VCA and ABCD constraint graphs.

for general Java applications, and we handle different statements in more detail. Another important aspect of our VCA approach is that we use different techniques to reduce the cost of data-flow analysis, such as limiting constraint graph node size, and enforcing iteration in pseudo-topological order.

Compared with other algorithms, our VCA works on bytecode level and does not change the program. The analysis results are encoded in the class file attributes. Thus, there are no problems with precise exception semantics. It is capable of preserving information from various sources. Although it uses a relatively sophisticated abstraction for the data-flow analysis, the techniques used in the algorithm reduce the overhead to a minimum. VCA can be very easily extended to take advantage of results from other analyses. We demonstrated how the two extended algorithms can improve the analysis results dramatically for array intensive benchmarks.

Ghemawat et. al. [9] described an algorithm called *field analysis* which exploits the declared access restrictions placed on fields in a modular language. Java programs are based on classes. Classes, fields, and methods have modifiers which limit access to them. Some fields with modifiers `private`, or `final` can only be accessed in a limited scope. By scanning the code in the scope, all possible value or object that a field can hold at the run-time is determinable. They implemented the algorithm in the Swift optimizing compiler [25]. The analysis results is used by other analyses for object inlining, stack allocation, and synchronization removal. They reported an average 7% speedups.

To target the scientific programs which use multidimensional arrays frequently, our rectangular array analysis provides very important information to the VCA, which helps the conservative VCA remove almost hundred per cent bounds checks in some

typical applications. To the best of our knowledge, very few other works takes advantage of knowing array shapes. Further, we believe the array shape information can also help memory layout of array objects in a virtual machine[4].

Chapter 5

Conclusions

In this thesis we have presented a collection of techniques for eliminating array bounds checks in Java. Our base analysis, variable constraint analysis (VCA), is a flow-sensitive intraprocedural analysis that approximates the constraints between important program variables at program points corresponding to array access statements. The analysis has been made efficient by reducing the size of the graphs, choosing an appropriate worklist order, and applying a widening at loop entry points. As shown in the experimental results, the size of the graphs is small (around 10 nodes for our benchmarks), and the average number of iterations per basic block is always less than 3.

In order to improve the precision of the base VCA analysis, we have described two additional techniques. Array field analysis is applied to each class to find those array type fields that always hold an array with a fixed constant length. Rectangular array analysis is applied to whole programs to find those variables that always refer to rectangular, non-ragged, arrays. Given the information from these analyses, the intraprocedural VCA analysis was improved to include information about fields, and upper dimensions for multidimensional arrays.

Our analyses were implemented in the Soot optimization/annotation framework, and we provided dynamic results that showed the effectiveness of the base VCA analysis and the incremental improvements due to field and rectangular array analysis. These results were quite encouraging and demonstrated that almost all checks could be eliminated for those benchmarks with very regular computations. We also provided experimental results for Kaffe and IBM's HPCJ to demonstrate that significant run-time savings can be achieved as a result of the analysis.

Our next phase of work will be to integrate a side-effect analysis into the framework, and improve upon information for arrays stored in objects. To extend constraint graphs to represent other arithmetic operations is a very interesting topic.

Bibliography

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adapative Optimization in the Jalapeño JVM. In *Proceedings OOPSLA 2000 Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices, pages 47–65. ACM, Oct. 2000.

[2] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 324–341, New York, Oct. 6–10 1996. ACM Press.

[3] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation(PLDI)*, pages 321–333, Vancouver, BC, Canada, June 2000.

[4] M. Cierniak and W. Li. Optimizing Java bytecodes. *Concurrency, Practice and Experience*, 9(6):427–444, 1997.

[5] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation(PLDI)*, pages 13–26, Vancouver, BC, Canada, June 2000.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill and MIT Press, 1990.

[7] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In W. G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture*

Notes in Computer Science, pages 77–101, Åarhus, Denmark, 7–11 Aug. 1995. Springer.

[8] E. M. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium 2000*, Lecture Notes in Computer Science, pages 199–219, Santa Barbara, June 2000.

[9] S. Ghemawat, K. Randall, and D. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation(PLDI)*, pages 334–344, Vancouver, BC, Canada, June 2000.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 1996.

[11] D. Griswold. The Java HotSpot Virtual Machine Architecture, 1998. http://www.javasoft.com/products/hotspot/whitepaper.html.

[12] R. Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, White Plains, NY, June 1990.

[13] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems, 2(1-4):135–150*, 1993.

[14] W. Harrison. Compiler analysis of the value ranges of variables. *IEEE Transactions on Software Engineering, 3(3):243–250*, 1977.

[15] Kaffe Virtual Machine. http://www.kaffe.org.

[16] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. *ACM SIGPLAN Notices*, 30(6):270–278, 1995.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[18] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, pages 114–119, June 1982.

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

[20] J. Moreira, S. Midkiff, and M. Gupta. A Standard Java Array Package for Technical Computing. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.

[21] J. Moreira, S. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, Mar. 2000.

[22] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[23] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A Framework for Optimizing Java Using Attributes. Sable Technical Report 2000-2 (revised), Sable Research Group, McGill University, Sept. 2000. To appear in CC2001.

[24] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing java using attributes. In R. Wilhelm, editor, *Compiler Construction, 10th International Conference*, volume 2027 of *Lecture Notes in Computer Science*, pages 334–354, Genova, Italy, April 2001. Springer.

[25] D. J. Scales, K. H. Randall, S. Ghemawat, and J. Dean. The swift java compiler: Design and implementation. Technical Report 2000/2, Compaq Western Research Laboratory, 2000.

[26] R. Shaham. Automatic removal of array memory leaks in Java. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, September 1999. Available at http://www.math.tau.ac.il/~ rans/thesis.zip.

[27] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in java. In D. A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 50–66, Berlin, Germany, March 2000. Springer.

[28] Soot - a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[29] Spec JVM98 benchmarks. http://www.spec.org/osg/jvm98/index.html.

[30] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[31] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, and Étienne Gagnon. Practical Virtual Method Call Resolution for Java. In *Proceedings OOPSLA 2000 Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices, pages 264–280. ACM, Oct. 2000.

[32] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings OOPSLA 2000 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, oct 2000.

[33] R. Vallée-Rai. Soot: A Java Bytecode Optimization Framework. Master's thesis, McGill University, October 2000.

[34] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In D. A. Watt, editor, *Compiler Construction, 9th International Conference*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, Germany, March 2000. Springer.

Appendix A

Implementation classes in Soot

Classes implementing three analyses locate in the directory rooted from Soot project: `%SOOTDIR%/soot/jimple/toolkits/annotation/arraycheck/`. The class files are listed below:

```
Array2ndDimensionSymbol.java
ArrayBoundsChecker.java
ArrayBoundsCheckerAnalysis.java
ArrayIndexLivenessAnalysis.java
ArrayReferenceNode.java
BoolValue.java
BoundedPriorityList.java
ClassFieldAnalysis.java
ExtendedHashMutableDirectedGraph.java
FlowGraphEdge.java
IntContainer.java
MethodLocal.java
MethodParameter.java
MethodReturn.java
RectangularArrayFinder.java
WeightedDirectedEdge.java
WeightedDirectedSparseGraph.java
```

The `ArrayBoundsChecker` class is a wrapper handling parameters and calling

other analyses. The `ArrayBoundsCheckerAnalysis` implements VCA, and the `WeightedDirectedSparseGraph` implements VCG. The `ClassFieldAnalysis` and `RectangularArrayFinder` implement `array field analysis` and `rectangular array analysis` respectively. Other classes are utility classes.