# Related Field Analysis

Aneesh Aggarwal
University of Maryland
aneesh@glue.umd.edu

Keith H. Randall
Systems Research Center
Compaq Computer Corp.
randall@pa.dec.com

## Abstract

We present an extension of field analysis (see [4]) called *related field analysis* which is a general technique for proving relationships between two or more fields of an object. We demonstrate the feasibility and applicability of related field analysis by applying it to the problem of removing array bounds checks. For array bounds check removal, we define a pair of related fields to be an integer field and an array field for which the integer field has a known relationship to the length of the array. This related field information can then be used to remove array bounds checks from accesses to the array field. Our results show that related field analysis can remove an average of 50% of the dynamic array bounds checks on a wide range of applications.

We describe the implementation of related field analysis in the Swift optimizing compiler for Java, as well as the optimizations that exploit the results of related field analysis.

## 1 Introduction

Modern programming languages like Java are becoming popular because they provide features like type safety, portability, object-oriented type dispatch, and automatic memory management which improve programmer productivity, reduce bugs, and improve security. These features, however, often add additional run-time overhead to a program which would not be required of an equivalent C or C++ program. In order to improve the feasibility of these modern languages for high-performance programming, compiler optimizations must remove as much of this run-time overhead as possible. This paper concentrates on removing the overhead of array bounds checks using a novel technique called *related field analysis*.

Array bounds checks are required in type-safe languages to make sure that applications don't inadvertently (or maliciously) write data outside the allocated portion of an array. In Java, bounds checks throw an exception if the index of an array reference is negative or greater than or equal to the length of the array. Each bounds check requires only a few instructions, but in tight loops which access arrays, bounds checks can add a significant overhead to the program. The overheads of bounds checks can be eliminated, however, if at compile time we can prove that the index of the array reference will always be in bounds.

The task of removing bounds checks from Java programs is well-studied [5, 8, 12, 7, 6, 9, 13, 10, 1]. A simple example of the type of bounds check that can be removed using existing techniques is the code below:

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

In this case, information local to the method is enough to deduce that the array accesses will always be in bounds. In more complicated cases, however, traditional techniques break down because there is no obvious relationship between the loop bound and the array bound.

To motivate the need for related field analysis, consider the following code:

```
class IntVector {
    int size;
    int[] elements;
    int length() { return size; }
    int elementAt(int i) {
        return elements[i];
    }
}
int addup(IntVector v) {
    int x = 0;
    for(int i = 0; i < v.length(); i++)
        x += v.elementAt(i);
    return x;
}
```

This new version of addup follows the same programming idiom as in the previous example, but the integer array has been abstracted to an IntVector and the array accesses are hidden inside the elementAt method.

How might we prove that the bounds checks inside elementAt never fail? A first step is to inline the implementations of the length and elementAt methods of IntVector:

```
int addup(IntVector v) {
    int x = 0;
    for(int i = 0; i < v.size; i++)
        x += v.elements[i];
    return x;
}
```

Even after inlining, proving that no array bounds exceptions will occur is difficult. In order to prove that the accesses to the array v.elements are in-bounds, we need to know something about the relationship between v.size and v.elements.length, in particular, that v.size is at most v.elements.length. Establishing this relationship is the job of related field analysis. In our example, if related field analysis is able to prove that the size field of an IntVector is never greater than the length of the array in the elements field of the same IntVector, then we can remove the bounds check in our inlined version of addup.

Using this technique, we are able to reduce by half the number of bounds checks executed in the SpecJVM98 benchmarks. These numbers include as a starting point only those bounds checks that were not eliminated through any other means, including standard optimizations and field analysis [4] which already have removed many of the simple cases. Interestingly, our technique can remove bounds checks from complex collection classes like java.util.Vector. Thus, we think our analysis provides a good step towards removing the remaining unnecessary bounds checks from Java programs.

In this paper, we describe our implementation of related field analysis in the Swift compiler. In Section 2 we give an invariant we try to prove about two related fields, and describe how we prove this invariant. In Section 3 we show how to use this invariant to remove bounds checks from programs. Section 4 discusses modifications to the basic technique to handle multithreaded programs. Then, in Section 5 we give some examples that show where our analysis technique succeeds and where it breaks down. Section 6 details our performance analysis, Section 7 describes some related work, and Section 8 concludes.

# 2 Related field analysis

In this section, we describe our implementation of related field analysis. Related field analysis can be viewed as proving an *invariant* about the related fields. The general framework for related field analysis allows for many different types of invariants, depending on their usefulness in optimization. We concentrate in this paper on proving an invariant that is useful for removing array bounds checks.

## 2.1 The Invariant

The invariant we use for removing array bounds checks is parameterized by two fields, called a and b. We require the field a to be an instance field of array type, and the field b to be an instance field of integer type. The invariant we prove is the following:

**Invariant 1** *Let* a *be an instance field of class* c *of array type, and let* b *be an instance field of class* c *of integral type. Then* $\mathtt{a} = \mathtt{null}$ *or* $0 \le \mathtt{b} \le \mathtt{a.length}$ *for every instance of class* c*.*

This invariant captures the situation where the array a contains elements of a set, list, or other data structure, and the integer b contains the number of elements which are valid in the array a.

For single-threaded programs, we require that Invariant 1 hold on entry and exit from every method in the program. The requirement for multithreaded programs is somewhat more complicated - see Section 4 for details.

To provide intuition for why Invariant 1 is useful, we demonstrate how we use it to remove bounds checks. Consider the removeLastElement method of a list abstraction which might look like the following:

```
Object removeLastElement() {
    if (b >= 1)
        return a[--b];
    else
        return null;
}
```

The list abstraction maintains the field b as the number of elements of the array a which belong to the list. If we are able to prove Invariant 1, then we know upon entry to this method that either a is null, or $0 \le \mathtt{b} \le \mathtt{a.length}$. In the former case, a NullPointerException will be thrown and the bounds check is unnecessary.[1] In the latter case, combining Invariant 1 with the branch condition $\mathtt{b} \ge 1$ proves that the access a[--b] will always pass its bounds check. In either case the bounds check is unnecessary and can be removed.

## 2.2 Finding related fields

Our compiler performs related field analysis by trying to find a relationship between every potential pair of related fields. We run our analysis for every pair of fields a and b such that a and b are instance fields of the same object, where a has an

[1] The field analysis technique from [4] can be used to prove that a is always non-null and hence the null pointer check will never throw an exception.

| Class | Field | Code to scan |
|---|---|---|
| `public` | `private` | containing class |
| `public` | `package` | containing package |
| `public` | `protected` | containing package and subclasses |
| non-public | `private` | containing class |
| non-public | non-private | containing package |

Table 1: Field Access Regions

array type and `b` is an integer. We record each pair of fields for which we are successful in proving Invariant 1.

In order to prove Invariant 1, it is necessary to find all accesses in the program to the fields in question. We use the same technique as is described in [4], where field access control modifiers are used to limit the scope of the search. The mechanism is summarized in Table 1.

## 2.3 Computing Related Fields

We prove that two fields satisfy Invariant 1 by looking at all assignments to the two fields. In this section, we assume a single-threaded program. We defer discussion of multi-threaded programs to Section 4.

We check Invariant 1 by examining each method which contains an assignment to either field. We use assume-guarantee inductive reasoning to check Invariant 1. We assume Invariant 1 holds on method entry and at the return point of every invocation, and prove that it holds at the call point of every invocation and on method exit. Note that newly allocated, but not yet constructed, objects are initialized to a state that satisfies Invariant 1.

Analysis of a single method proceeds by case analysis. We analyze two types of assignments, those that assign to the array field `a` and those that assign to the integer field `b`. We first examine all assignments to the array field `a`. We check that the invariant is maintained by each assignment. If `a` is assigned a null pointer, then the invariant is trivially maintained. Otherwise, we must ensure that the length of the new array is at least `b`. Typically, code will allocate a new array with a length that is a non-decreasing function of either `b` or the old length. Our analysis currently detects non-decreasing functions of the form $f(x) = c_1 x + c_2$, with $c_1 \geq 1$ and $c_2 \geq 0$. We also handle the cases where `a` is assigned something other than a newly allocated array, and `b` is assigned either 0 or `a.length` immediately before or after the assignment to `a`. Finally, let us define *dependent reads* as the reads of `a` and `b` that are used to calculate the new size for `a` or value for `b`. We require that no assignments to `a` or `b` or any method calls which might modify `a` or `b` occur between any of the dependent reads and the subsequent write to `a`. This requirement ensures that the state of the object observed by the reads has not changed by the time that `a` is written.

Next, we examine all assignments to the integer field `b`. For each assignment, we attempt to find a preceding assignment to `a` or a preceding conditional branch (or both) that will help us prove that the invariant is maintained by the assignment to `b`. If the assignment to `b` decrements `b` by $k$, then we require the assignment to `b` to be on the taken branch of a conditional instruction of the form $\texttt{b} \geq k$. Similarly, if the assignment increments `b` by $k$, then we require the assignment to `b` to be on the taken branch of a conditional instruction of the form $\texttt{b} \leq \texttt{a.length} + k$. Alternatively, if we find an assignment to `a` preceding the assignment to `b`, then we prove the invariant by ensuring that that assignment to `a` increases the size of `a` by at least $k$. As in the previous case, we also have to ensure that no assignments to `a` or `b` or method calls which might modify `a` or `b` occur between the dependent reads and the write to `b`.

Related field analysis is a fairly inexpensive analysis to perform. It requires only a single pass over the methods in a program. For each method, we build the SSA graph for the method and perform the above analysis on it, for each `a` or `b` field accessed by the method.

# 3 Removing array bounds checks

In this section, we describe how the related field analysis results are used to remove array bounds checks in the program. Again, we assume a single-threaded program. We defer discussion of multithreaded programs to Section 4.

The related field analysis from Section 2 generates a list of fields which are known to be related by Invariant 1. In the optimization phase, for every array access in the code, we analyze the access to see if its associated bounds check can be removed. First, we consider only array accesses whose array is directly loaded from an instance field `a` for which we have a related field `b`. Then, we check that the index `i` used in the array access is non-negative and less than `b` (any one of the `b`'s, if there happen to be more than one associated with `a`). Although the techniques for proving these two facts are outside the scope of this paper, typically they will be checked using traditional value-range techniques, augmented with the information contained in Invariant 1. In our compiler, we use a simple intraprocedural dataflow analysis [11] to check these inequalities.

For example, consider the following simple loop:

```
for(int i = 0; i < b; i++) x += a[i];.
```

To prove that the bounds check is unnecessary, we observe that the inequality $\texttt{i} < \texttt{b}$ together with the inequality $\texttt{b} \leq \texttt{a.length}$ (from Invariant 1) imply the inequality $\texttt{i} < \texttt{a.length}$. This fact, together with the inequality $\texttt{i} \geq 0$, is enough to prove that the array index is in bounds, so no bounds check is required. The associated bounds check for that access is then removed.

## 4 Handling multithreaded programs

The analysis from Section 2 assumes a single-threaded program, which is an invalid assumption for many Java programs. This section describes how the analysis can be extended to handle multithreaded programs. Although this portion of related field analysis has not yet been implemented in our compiler, we expect that it will be straightforward to incorporate.

The difficulty encountered with multithreaded programs is that several threads may be accessing an object at once. This interference presents two problems. First, two threads might be modifying the same object at once, destroying Invariant 1 for that object. Second, the optimization phase described in Section 3 may observe the state of an object when Invariant 1 is not satisfied (remember, Invariant 1 is only proved to hold on method entry and exit).

We handle both of the above difficulties by relying on synchronization declarations in the user's program. We require that Invariant 1 not only hold on method entry and exit, but also that it hold for every object whose lock is not held by any thread. Furthermore, we require that all modifications of the `a` and `b` fields, as well as their dependent reads, are contained in a synchronized block that synchronizes on the object containing `a` and `b`. Typically, application code will enforce this requirement by putting the code that accesses `a` and `b` in a `synchronized` statement or a `synchronized` method.

Similarly, when performing the optimization phase described in Section 3, we must ensure that the reads of `a` and `b` are both contained in a single synchronized block which is synchronized on the object containing `a` and `b`. This rule guarantees that the optimization phase can rely on Invariant 1 holding for the accessed object.

Several relaxations of this requirement are possible. If the object containing `a` and `b` can be proved thread-local, then these extra multithreaded requirements are not necessary. Similarly, well-behaved constructors which do not leak a reference to object being constructed (called leak-free constructors in [4]) do not need synchronization.

None of the modifications described require any more complicated analysis than the analysis required for the single-threaded case.

## 5 Example applications

In this section, we take a closer look at some interesting cases where we can prove Invariant 1. In addition, we also explore cases where Invariant 1 is false, and where Invariant 1 is true but related field analysis is unable to prove it. These cases are examples which occur in or are derived from either the SpecJVM98 applications or the `java.util` library classes.

### 5.1 Helper methods

A common idiom in many codes is to use a helper function to change the length of the array field. A typical example of this idiom is where the integer field is compared with the length of the array, and if it is greater than or equal to the array length, a helper method is invoked to increase the array length. The following piece of code illustrates this idiom:

```
public class PointSet {
    private Point[] a;
    private int b;

    public PointSet(int size) {
        a = new Point[size];
        b = 0;
    }
    public void add(Point x) {
        if(b == a.length)
            increase();
        a[b++] = x;
    }
    private void increase() {
        int[] d = new int[b * 2 + 1];
        System.arraycopy(a, 0, d, 0, b);
        a = d;
    }
}
```

To handle this case, we modify our analysis rules so that when we find a method invocation while searching for an assignment to `a` backward from an assignment to `b`, then the invoked method (if it can be determined statically) is checked for a write to `a`. If such a method is found, then that method is inlined[2] and the write to `a` inside the helper method is used to check Invariant 1. In the example above, when analyzing the method `add`, we need to inline the method `increase` to ensure that Invariant 1 is maintained. Currently, we only inline a single method invocation and disallow recursive calls or calls to other methods. Thus, we might miss more complicated cases where the inlined method in turn invokes another method to change the array length.

A related idiom, which is also very common, is to have a constructor call another constructor to initialize the array. This case is handled similarly.

### 5.2 Limitations of Related Field Analysis

In this section, we describe some of the difficulties which arise when attempting to prove Invariant 1 about two related fields. Some of these difficulties derive from the fact that Invariant 1 is simply not true. Others difficulties derive from the limitations of our analysis and could be overcome with more sophisticated analyses. We give an example of each.

[2] This inlining occurs for analysis purposes only, and has no effect on the inlining performed by the rest of the compiler.

Our first example comes from the `compress` program in the SpecJVM98 benchmark. The following code appears in a buffer abstraction:

```
public void putbyte(byte c) {
    OutBuff[OutCnt++] = c;
}
```

In this method, the programmer is adding another element to the array without first checking for overflow. Our related field analysis fails to remove this bounds check. However, it may turn out that the programmer knows some *a priori* bound on how many bytes will ever be written to the buffer, so the bounds check may be unnecessary. In fact, comments in the code indicate that the buffer will overflow, but only if the compressed version of a file is larger than its uncompressed version.

The second case we examine is a situation where Invariant 1 is true, but is difficult to prove. Consider the following code:

```
public void removeAllElements() {
    while (count > 0)
        removeElement();
}
private void removeElement() {
    count--;
}
```

We cannot prove Invariant 1 using related field analysis, because we notice that the method `removeElement` does an unprotected decrement of the `count` field (the `b` field in our analysis). To prove Invariant 1 in this case, we would have to notice that all callers of `removeElement` ensure that `count` is not zero before calling `removeElement`. A more sophisticated analysis which includes an interprocedural component would be required to handle this case.

# 6 Performance results

In this section, we demonstrate the benefits of related field analysis. We first describe the experimental setup and then give some performance results for the SpecJVM98 applications.

## 6.1 Experimental setup

Our performance results are for the Swift compiler [11] running under Tru64 Unix (formerly known as Digital Unix) on an XP1000 Alpha workstation. The workstation has one 667 MHz Alpha 21264 out-of-order processor, which has 64 Kbyte on-chip instruction and data caches, and a 4 Mbyte combined board-level cache. Swift is a complete Java optimizing compiler that implements numerous optimizations in addition to those described in this paper, including: method inlining, method splitting, global common subexpression elimination, global code motion, conditional constant propagation, interprocedural alias analysis, exact type analysis, peephole optimizations, and trace scheduling. It also includes an effective register allocator based on biased graph coloring. Swift is written in Java and translates Java bytecodes to Alpha machine code. The generated code is installed into a high-performance JVM for Java 1.2 that has a mostly-copying garbage collector and extremely fast synchronization [2]. All results are for applications running with the Java 1.2 standard library.

Our implementation of related field analysis currently handles only single-threaded applications. Although more work would be required to implement the multithreaded extensions described in Section 4, we do not expect the performance results to change significantly when this extension is implemented.

| | **problem domain** | **lines of code** | **run time w/o RFA** | **run time w/ RFA** | **run time Max.** |
|---|---|---|---|---|---|
| compress | text compression | 910 | 9.38s | 9.34s | 8.84s |
| jess | expert system | 9734 | 4.75s | 4.62s | 4.49s |
| db | database retrieval | 1026 | 15.53s | 15.27s | 15.22s |
| javac | Java compiler | ∼18000 | 11.74s | 11.56s | 11.49s |
| mpeg | audio decompr. | ∼3600 | 5.86s | 5.67s | 5.28s |
| jack | parser generator | ∼7500 | 5.77s | 5.58s | 5.57s |

Table 2: Experimental results for related field analysis. For each application, we give its problem domain, an estimated line count, and the running time using three different versions of the Swift compiler. The first running time is for Swift-compiled code with no related field analysis. The second running time is Swift-compiled code with related field analysis turned on. The last column shows the running time when all bounds checks in the program have been removed.

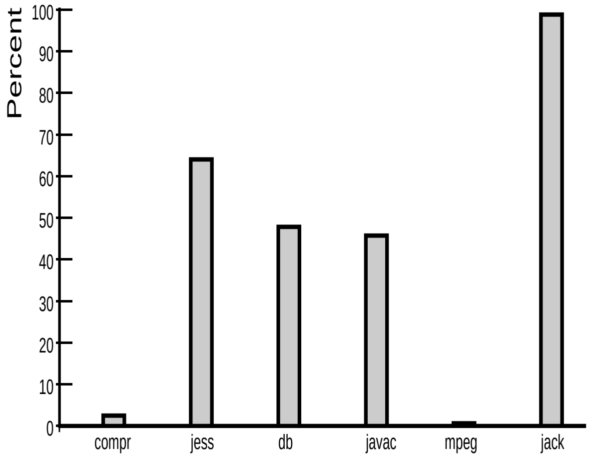


Figure 1: Percentage of dynamic bounds checks removed

## 6.2 Performance Results

We ran our compiler on a number of SpecJVM98 applications to determine the effectiveness of related field analysis.

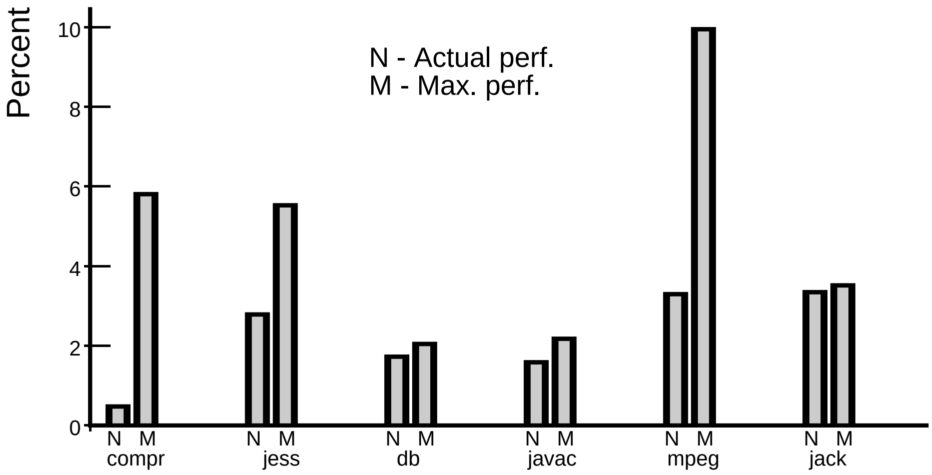


Figure 2: Percentage improvement in running time. The "N" bars represent the improvement with related field analysis. The "M" bars represent the maximum possible improvement if all bounds checks in the program are removed.

The results are listed in Table 2. Figure 2 displays the resulting performance improvements graphically. For each application, we show the running-time improvement generated by related field analysis, as well as the running-time improvement when all bounds checks are removed.

Figure 2 demonstrates that related field analysis achieves an average performance improvement of approximately 2.5%. Compared to the maximum possible average performance improvement of 5%, we can see that related field analysis is able to obtain approximately 50% of the maximum possible speedup. Because runtime measurements contain noise and are difficult to repeat experimentally, we also show in Figure 1 the percentage of dynamically executed bounds checks removed in each application, measured by instrumenting the generated code. The general improvements seen in the timing measurements are echoed by the execution counts.

### 6.3 Analysis

Apart from the applications, related field analysis also applies to the `java.util` library, because this particular library consists of many classes on which related field analysis performs well. These classes tend to be used frequently in applications and hence overall performance could be improved with faster versions of these classes. The majority of the performance improvements for javac and db are due to optimization of the `java.util` classes. Optimization of `java.util` classes also helped jack and mpeg. In contrast, almost all of the improvements for jess and compress are due to optimizations of the programs themselves.

Related field analysis performed poorly on both compress and mpeg. We believe this failure is caused by these applications using more complicated invariants to ensure in-bounds array accesses than our related field analysis is capable of discovering.

## 7 Related work

The problem of removing array bounds checks is well-studied, precluding a comprehensive overview of the area. Originally, removal of bounds checks was done using generic optimizations like code motion [7]. Recently, there has been a renaissance in bounds check removal as Java has become popular. Techniques can be divided by the use of their underlying technology, for example theorem provers [13, 8, 12], linear program solvers [10], or value range propagators [5, 9]. We believe related field analysis, although limited to removing bounds checks which conform to a particular invariant, is nonetheless an inexpensive technique for achieving similar bounds-check removal rates to these other more expensive techniques.

Program checkers such as ESC-Java [3] allow annotations that specify properties of related fields that are verified as part of the checking process. In particular, ESC-Java allows the programmer to specify explicitly invariants like Invariant 1. Our compiler attempts to discover these properties automatically, rather than using programmer annotations.

A precursor to this work called field analysis [4] proves properties about individual fields, and can be used to remove many Java overheads, including bounds checks on fields which contain arrays of constant size. Regular field analysis, however, cannot remove bounds checks from array fields which change size. As far as we know, no systems have been proposed which perform related field analysis.

## 8 Conclusion

We described related field analysis, an interprocedural analysis that determines useful properties relating fields of objects. We demonstrated that related field analysis can be used to remove a substantial fraction of array bounds checks from application programs. Like field analysis [4], related field analysis does not require analysis of an entire program to be effective. Its strength lies in removing array bounds checks when field properties of individual fields do not provide enough information to do so.

We implemented a working version of related field analysis in the Swift compiler. The overall speedup obtained using this optimization ranges from 0.5% to around 4%, whereas the maximum performance improvement that can be obtained by removing all the bounds checks ranges from 2% to around 10%. We showed that in many cases almost all of the performance improvement could be exploited by our implementation, while in most of the other cases a considerable improvement was noted. On average, about 50% of the dynamic array bounds checks were removed by our technique. We observe that with a more sophisticated interprocedural analysis, more related field information could be extracted. Nevertheless, related field analysis is widely applicable and performs well, especially when applied to modular code like the Java libraries.

Finally, we believe that other relations between fields would be useful for optimization. A similar invariant to the one we used could prove that an integer field represents an index into an array field, as is often used in enumerations. Also, our technique could be generalized to relate an integer field to the size of any abstract collection class, not just arrays. Other types of relations that might prove useful include alias relations, type relations, and lifetime relations. For instance, if we could prove that two reference fields are never equal, then memory operations on the objects contained in those fields would be reorderable.

# References


[1] 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, June 2000.

[2] Compaq Computer Corporation. Compaq Fast Virtual Machine V1.2.2-4 for Alpha. At URL `http://www.compaq.com/java`.

[3] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, Compaq, 1998.

[4] S. Ghemawat, K. H. Randall, and D. J. 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, June 2000.

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

[6] P. Kolte and M. Wolfe. Elimination of Redundant Array Subscript Range Checks. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, June 1995.

[7] V. Markstein, J. Cocke, and P. Markstein. Optimization of Range Checking. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June 1982.

[8] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Oct. 1998. Available as tech report CMU-CS-98-154.

[9] J. R. Patterson. Accurate Static Branch Prediction by Value Range Propagation. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, pages 67–78, June 1995.

[10] R. Rugina and M. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 182–195, June 2000.

[11] 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, Apr. 2000.

[12] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 132–143, Jan. 1977.

[13] Z. Xu, B. P. Miller, and T. Reps. Safety Checking of Machine Code. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 70–82, June 2000.