Performance Enhancements for a Dynamic Invariant Detector

by

Chen Xiao

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2007

Author .....
Department of Electrical Engineering and Computer Science
February 2, 2007

Certified by .....
Michael D. Ernst
Associate Professor
Thesis Supervisor

Certified by .....
Jeff H. Perkins
Research Staff
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Performance Enhancements for a Dynamic Invariant Detector

by

Chen Xiao

Submitted to the Department of Electrical Engineering and Computer Science
on February 2, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Dynamic invariant detection is the identification of the likely properties about a program based on observed variable values during program execution. While other dynamic invariant detectors use a brute force algorithm, Daikon adds powerful optimizations to provide more scalable invariant detection without sacrificing the richness of the reported invariants. Daikon improves scalability by eliminating redundant invariants. For example, the suppression optimization allows Daikon to delay the creation of invariants that are logically implied by other true invariants. Although conceptually simple, the implementation of this optimization in Daikon has a large fixed cost and scales polynomially with the number of program variables.

I investigated performance problems in two implementations of the suppression optimization in Daikon and evaluated several methods for improving the algorithm for the suppression optimization: optimizing existing algorithms, using a hybrid, context-sensitive approach to maximize the effectiveness of the two algorithms, and batching applications of the algorithm to lower costs. Experimental results showed a 10% runtime improvement in Daikon runtime. In addition, I implemented an oracle to verify the implementation of these improvements and the other optimizations in Daikon.

Thesis Supervisor: Michael D. Ernst
Title: Associate Professor

Thesis Supervisor: Jeff H. Perkins
Title: Research Staff

Acknowledgments

First and foremost, I want to thank Professor Michael Ernst, my research advisor, for his support, advice, and feedback throughout the years. Professor Ernst always gave insightful comments about my work.

Research staff member Jeff Perkins provided me with so much guidance, which was manifested in many ways, from his detailed feedback on my writing to useful comments about research ideas. Jeff gave me so much of his time, listening to my problems and helping me brainstorm solutions and ideas for new experiments. Rest assured that without his guidance and feedback on both the research and writing, I could not have finished this gargantuan task.

I want to thank Edmond Lau for his patience while listening and critiquing my ideas and giving feedback on my writing.

Lastly, I want to thank my family and friends, especially those who gave me advice and perspective during my times of tribulation.

This research is supported by gifts from IBM, NTT, and Toshiba, a grant from the MIT Deshpande Center, DARPA contract FA8750-04-2-0254, and NSF grants CCR-0133580 and CCR-0234651.

Contents

List of Figures

Chapter 1

Introduction

1.1 Dynamic invariant detection and scalability problems

Dynamic detection of likely invariants [4] is a program analysis that hypothesizes likely program properties (invariants) based on observed values of variables during program execution. The reported results include invariants like "at exit from method `makeProperFraction`, *denominator* > *numerator*", or "at exit from method `factorial`, *product* $\neq$ 0".The technique reports as likely invariants properties that hold over *all* of the observed values. As with all dynamic analysis, the accuracy of the results depends on the quality of the test suite. However, even moderate test suites produce relatively accurate results [13, 12] and tools exist that help develop good test suites for invariant detection [7, 19].

Dynamic invariant detection is an important and practical problem. Dynamically detected invariants have aided programmers in understanding and debugging programs [6, 16, 10, 17, 5, 9, 20, 1], has assisted in theorem-proving [11], repairing data structures [2], automatically generating program specifications [12], generating test cases [14], detecting errors [8], and avoiding bugs [3]. Therefore, it is worthwhile to improve the performance of dynamic invariant detectors, which allows invariant detection to be applied to a wider range of programs.

Implementing dynamic invariant detection efficiently is challenging. A simple, brute force algorithm is straightforward but fails to scale to problems of substantial size. The simple algorithm, described in detail in section 2.2, initially creates, for each property, a candidate invariant for each combination of variables. The algorithm examines the *samples*, values for a set of variables, and removes any invariants that are contradicted by the values.

The memory costs of the algorithm are dominated by the storage of the invariants and its time costs are dominated by the application of samples to these invariants. Since both time and memory vary with the number of invariants, we can discuss the scalability of the algorithm in terms of the growth of the number of invariants.

The number of invariants depends on the number of variables examined, the number of program points, and the set of invariants examined [15]. The number of invariants grows polynomially with the number of variables [15]. More specifically, if the invariants involve up to n variables, then given v variables, the number of possible invariants is v^n, which means that the algorithm takes $O(v^n)$ time and space.

Because the algorithm processes each program point independently, the number of invariants only varies linearly with the number of program points. The algorithm does not scale poorly with respect to the number of lines of code in a program because the number of variables is the major factor in determining the number of invariants.

There are a variety of ways to address the scalability problems of dynamic invariant detection. Some detectors reduce one or more of the factors influencing the running time, by decreasing the number of variables examined, the number of program points monitored or the set of properties checked. For example, remote program sampling in [9] only checks a linear number of variable combinations in invariants over two variables, rather than all v^2 combinations. The Carrot detector limits the set of invariants checked by ignoring invariants over three variables [16]. The DIDUCE tool only checks for invariants over one variable [6]. These changes sacrifice the quality of the output.

The Daikon invariant detector attempts to achieve good performance without reducing the factors influencing runtime by adding optimizations to preserve the quality

of the results and the richness of its invariants. The optimizations, explained in detail in section 2.3, have two purposes. They allow Daikon to avoid creating and processing redundant invariants that can be inferred later from the set of created invariants. Daikon ignores these redundant invariants during runtime and also omits them from the final output. Ignoring these redundant invariants improves the scalability of Daikon's algorithm and the readability of the results for users.

Although improving the scalability of Daikon, the optimizations also introduce problems in testing for correctness (section 1.2) and suffer from performance problems (section 1.3). This thesis focuses on ensuring the correctness and improving the efficiency of invariant detection optimizations.

1.2 Daikon optimizations and testing for correctness

Daikon checks hundreds of properties over many variables and produces a large amount of output. Consequently, checking the correctness of these results is difficult to do by hand. In addition, since the optimizations ignore redundant invariants during processing and in outputting the results, thorough testing is even harder. More specifically, an invariant that is missing from the output could be missing for several reasons. The invariant could be correctly labeled as false and discarded or correctly labeled as true, and ignored by the optimizations. On the other hand, the invariant could be incorrectly labeled as false and discarded by Daikon because of a bug in the implementation of the optimizations. The difficulty lies in differentiating between the correctly labeled and incorrectly cases.

To prove correctness, the tester must make sure that none of the reported invariants are false by verifying that reported results are true over all of the samples. The test must also show that all true invariants are either present in the output or can be inferred from the reported invariants.

To accomplish this task, I use an oracle for verifying the correctness of the opti-

mizations. The method uses an easily verifiable brute force algorithm as an oracle to generate the same output as the program with complex optimizations (Daikon). Running the brute force algorithm produces a complete set of true invariants. In addition, running the complex algorithm (with its optimizations) and then undoing the work of the optimizations recovers the omitted invariants to obtain a complete set of true invariants. Differences in the two outputs of the two algorithms indicate potential problems.

I can use the approach not only to check the implementation of all of the optimizations but also any additional optimization changes that I make to Daikon. Using this approach, I found problems both in the implementation of the optimizations and in code that was not the target of this approach, the invariants themselves.

1.3 Daikon optimizations and performance issues

When implementing optimizations, often there is a memory and time trade-off. A straightforward implementation of the optimization reduces processing time but must use up more memory to store useful information. On the other hand, in order to save memory, the optimization has more complex processing because there is less information.

For Daikon optimizations, memory is the more important resource because memory has a hard limit. Experimental results show that due to garbage collection and thrashing, Daikon performance drops drastically close to the limit of physical memory [15]. Once memory runs out, Daikon cannot produce any results. In this case, users would probably rather run Daikon for an extra few hours to have results than to have Daikon run out of memory and not produce any. This philosophy that memory is more critical than time guides the decisions made when implementing the optimizations in Daikon.

The memory and time requirements of Daikon grow with the number of invariants, which grows polynomially with the number of variables. Optimizations whose requirements grow with the number of program points or the number of variables

will use much less resources than Daikon itself. Optimizations whose requirements grow with the number of invariants will use space and/or time at a rate similar to Daikon. The latter set of optimizations may not be as effective because overhead in the optimizations may be greater than their savings.

The *suppression optimization* in Daikon delays the creation of invariants that are logically implied by other true invariants. For example, if A logically implies B, as long as A is true, Daikon does not need to examine B. I focused on this optimization because its algorithms either use space or time on the order of the number of invariants, making this optimization less effective the other optimizations (see section 2.3.4).

Since the developers of Daikon prioritize memory over time, the implementation of this optimization improves Daikon's memory usage at the cost of time. The suppression optimization is the most time intensive of all Daikon optimizations. Experimentally, I found that as the number of variables grows larger, the implementation of the optimization takes up a significant amount of running time, varying from 20 to 40% of the total running time of Daikon.

The other optimizations do not suffer from the dilemma of choosing between saving processing time and saving memory. The data structures needed for the optimal time implementations of the optimizations grow linearly with the number of variables (discussed in the next chapter). However, for the suppression optimization, the optimal time implementation needs space that grows with the number of invariants, while the optimal space implementation needs time that also grows with the number of invariants (see section 2.3.4).

1.4 Thesis outline

The research in this thesis divides into three parts: investigation, design, and verification. The initial part of the research focused on the current algorithms for the suppression optimization, running experiments, and analyzing data to determine exactly where the time is spent. Based on the information found, I explored ways to

enhance the current algorithms and looked for new approaches to making Daikon perform better. I then built an oracle to test the implementation of these enhancements and optimizations.

The thesis is organized as follows. Chapter 2 gives background information on the Daikon architecture, largely using the information in [15], and Chapter 3 provides an overview of the suppression optimization. Chapters 4 and 5 detail the current algorithms implemented for the suppression optimization. In chapter 6, I describe the experimental data and its motivation for the design changes that I made to Daikon. Chapter 7 shows experimental results for the various design changes. Chapter 8 details the verification method used to check the implementation of the optimizations and the bugs found by the verification tool. I comment on some related work in Chapter 9 and close with remarks on limitations and future work in Chapter 10.

Chapter 2

Daikon Architecture

This chapter gives an overview of dynamic invariant detection and the Daikon architecture and is largely based on the structure and information from the incremental invariant detection paper [15].

2.1 Dynamic invariant detection overview

Dynamic invariant detection is the conjecturing of likely program properties at program runtime based on the values of program variables. The invariants in the output depend on the grammar of invariants, the program variables and the program points checked. The reported results include invariants like "at exit from method `makeImproperFraction`, *num* > *denom*", or "at entry to method `divide`, *denom* $\neq 0$". In the following sections, I introduce the basic algorithm used by Daikon to generate invariants and the optimizations employed by Daikon to accomplish the task efficiently.

A ***program point*** is a specific place in the program. Invariants can be generated about program variables at any program point based on their values during program execution. Daikon's basic program points are procedure entry and exit points. The reported invariants at these points correspond to the preconditions and postconditions of the method. In addition, Daikon also generates invariants over *aggregate* program points. For example, Daikon generalizes over the invariants at the entry

and exit points of the public methods and the exit points of the constructors of a class to produce the object point. The invariants at the object point correspond to representation invariants of the class.

A ***variable*** is one of the values over which Daikon looks for invariants. These include the values of program variables such as method parameters, return values, global variables, and fields of classes. Daikon derives additional variables from existing variables when applicable. For example, if an array `a` and the integer `i` are both in scope, then the variable `a[i]` may be interesting even though it does not exist explicitly in the program. Each variable has an associated type. The types are `hashcode`, `String`, `boolean`, `int`, `float`, `hashcode[]`, `String[]`, `boolean[]` `int[]`, `float[]`, and `int[]`. A `hashcode` is a unique ID for references or pointers to a specific location in memory, used for fast retrieval of the object with that `hashcode`. Daikon represents the `hashcode` type as an integer.

A ***sample*** is the set of values associated with the key variables at a certain program point for a given execution of the program. An example of a sample is "at exit from method `makeImproperFraction`, $num = 5$, $denom = 4$". Daikon verified the samples against the instantiated invariants to determine if those invariants are still true.

An ***invariant*** describes a relationship between variables in a program, e.g. $x > y$, $x = 0$, and $Ax + By + Cz = D$. An ***invariant type*** refers to a specific type of relationship such as less-than, or array-is-sorted.

An invariant in the Daikon grammar has two forms: a ***template*** form and a ***concrete*** form. The *template* form is an invariant type from the invariant grammar without reference to specific variables, but rather uses α, β, and γ as formal parameters. For example, the less-than invariant template is $\alpha < \beta$.

When Daikon instantiates the invariant template with variables at runtime, the invariant becomes *concrete*. If x, y, and z are variables from the program, examples of concrete less-than invariants are: $x < y$, $y < z$, and $z < x$. The invariant type of these invariants is $<$.

Daikon has a predefined set of invariant templates used to create concrete invari-

ants. For the rest of the thesis, I will use the term *invariant* to refer to concrete invariants and will use the term *invariant template* explicitly to differentiate between the two terms. In addition, I will use x, y, and z to compose concrete invariants and α, β, and γ to make up invariant templates.

The ***grammar of invariants*** is the set of invariant types that are instantiated and checked over the program variables. Daikon checks for invariants over one, two and three variables. We refer to these as unary, binary, and ternary invariants respectively. The grammar of invariants is the set of predefined invariant templates used by Daikon to create concrete invariants. Some examples of the invariant templates in Daikon's grammar set are: $\alpha = 0$, $\alpha > \beta$, $\alpha \bmod \beta = \gamma$, and $\alpha \in \beta$.

Invariant templates may be valid only over certain variable types. For example, in the invariant $\alpha > \beta$, α and β must be of the same type and cannot be hashcodes (it does not make sense to compare unique IDs numerically), in the invariant $\alpha \bmod \beta = \gamma$, all of the variables must be integers, and in the invariant $\alpha \in \beta$, β must be a collection and the elements in β should have the same type as α.

2.2 Invariant detection algorithm overview

A simple algorithm for invariant detection is:

At each program point:

1. Instantiate invariant templates in the grammar over all combinations of variables. For example, if the grammar consists of the invariant types "*prime*" and "=", and the integer variables are x, y, and z, then the algorithm instantiates the invariants $prime(x)$, $prime(y)$, $prime(z)$, $x = x$, $y = y$, $z = z$, $x = y$, $y = z$, $x = z$.

2. For each sample, check the sample values against each invariant, mark the invariants contradicted by the sample as false. Remove the falsified invariants. For example, the sample (3, 4, 3) falsifies the invariants $prime(y)$, $x = y$, and

$y = z$.

3. Report the invariants that remain after all samples have been processed. These are the true invariants since they still exist after all samples have been checked. The true invariants in the example are: $prime(x)$, $prime(z)$, $x = x$, $y = y$, $z = z$, and $x = z$.

The algorithm uses space to store the list of invariants that have not been contradicted by a sample. As described in section 1.1, this space grows polynomially with the number of variables. With v variables, p program points and s samples, the memory requirement for Daikon is $O(v^3 * p)$. In addition, the algorithm spends most of its time applying samples to the list of true invariants. The time requirement is $O(v^3 * p * s)$.

2.3 Optimizations

In this section, I show how Daikon optimizes the simple algorithm by reducing the number of invariants that are instantiated without affecting correctness. The invariants that are not instantiated by Daikon are redundant because they can be inferred from the other invariants.

2.3.1 Dynamic constants

A dynamic constant variable is one that maintains the same value at each sample. The invariant $x = a$ (for some constant a) makes all other unary invariants over the variable x redundant. For example, $x = 2$ implies $even(x)$ and $x > -5$. Likewise, for combinations of variables, $x = 2$ and $y = 5$ together imply $x < y$ and $x = y - 3$. Daikon takes advantage of this observation by not creating invariants over constant variables and only storing their values, which is sufficient information to create the invariants if necessary.

Daikon maintains a list of the constant variables and their values. At each sample, Daikon walks through the list, and for each variable, compares the value in the list

with the value in the sample and creates invariants over the variables that change values. The optimization uses v time and space to store and walk through the list.

2.3.2 Equality sets

If two or more variables are always equal, meaning they have the same values at each sample, then any invariant that is true for one of the variables is true for each of the other variables. More generally, for any invariant f, when $x = y$, $f(x)$ implies $f(y)$. For example, if $x = y$, then $x > 5$ implies that $y > 5$. Daikon capitalizes on this observation by putting the equal variables in an equality set and only creating invariants over the leader of the equality set.

Daikon stores all of the variables in the equality sets, including their leaders and non-leaders, and only instantiates invariants over the leaders. At each sample, Daikon checks that each non-leader still has the same value as the leader. If the value of a non-leader deviates, Daikon breaks up the equality set by making two equality sets, copies the invariants of the leader's set to the set containing the changed variable, and verifies that the copied invariants still hold for the changed variable.

For v variables, Daikon, at most creates v separate equality sets and uses v time to examine the values in each sample.

2.3.3 Variable point hierarchy

Some variable values affect invariants at multiple program points. For example, an invariant appearing at the object point implies that the invariant must be true at all entry and exit points of public methods and the exit points of the constructors in the class (see figure 2-1).

For two program points A and B, if all samples that appear at B also appear at A, then true invariants at A will also appear at B, and thus are redundant at B. In this case, we say that A is higher in the hierarchy than B. For example, if the invariant $x > y$ is true at all entry and exit points of public methods and at the exit points of the constructor, then the invariant is true at the object point.

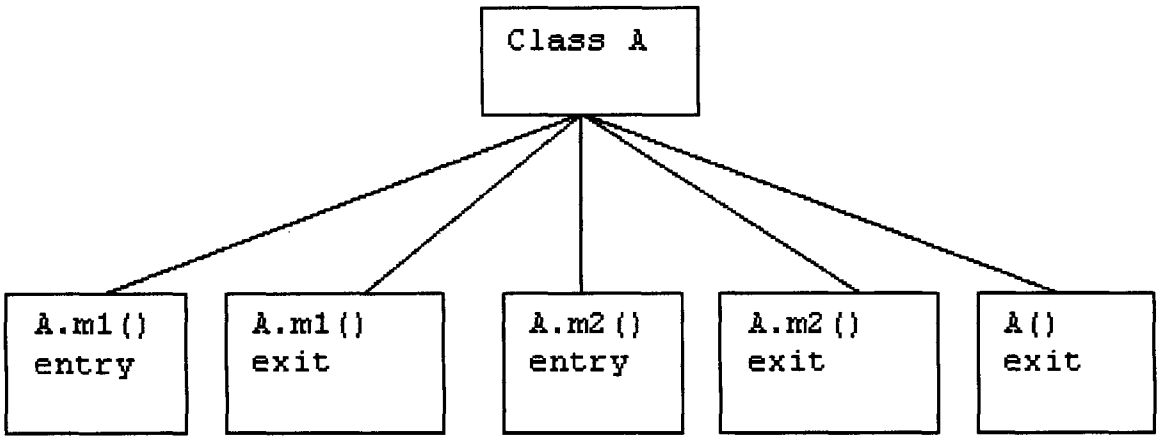

Figure 2-1: Example of the variable point hierarchy optimization. An invariant that is true at all entry and exit points of public methods and the exit points of the constructors of the class must also be true about the object point.

Daikon uses the hierarchy to avoid creating the invariants at A, and just gathering the information from its children in the hierarchy (B). Daikon only processes the leaves of the hierarchy and then infers invariants at the upper points by merging results from the level below. An invariant is true at an upper point if and only if it is true at each child of that point. Daikon infers the invariants in the upper levels after processing all of the samples at the lower levels.

Daikon uses a data structure that relates the variables at different hierarchies. With v variables, Daikon only needs to store the v variables and its relationships. Time-wise, Daikon does no work at each sample and only propagation at the end of processing the samples, which varies with v.

2.3.4 Suppressions

Some invariants are logically implied by other invariants. For example, if A logically implies B, then as long as A is true, B does not need to be examined. For example, the invariant $x > y$ implies $x \geq y$, therefore checking a sample that satisfies $x > y$

will mean that the sample also satisfies $x \geq y$. Thus, at runtime, Daikon does not check the invariant $x \geq y$ until $x > y$ becomes false.

There are two approaches to implementing the suppressions optimization: one is efficient with time and inefficient with memory, and the other is efficient with memory and inefficient with time.

The optimal time approach stores information about the validity of the antecedents of each implied invariant and hence knows exactly when Daikon needs to create the implied invariant. However, the approach needs a data structure that grows with the number of invariants (v^3) and actually uses more memory than creating the implied invariants.

The optimal space approach stores no state information about the implied invariants and their antecedents, but the lack of state complicates processing. With this approach, Daikon has to find all of the implied invariants and check whether they are still implied, making processing time grow with the number of invariants (v^3).

Since the developers of Daikon prioritize memory over time, its current implementation of the optimization uses the optimal space approach. I discuss the suppression optimization in more detail in chapter 3.

2.3.5 Time and space cost comparison

The first three of the four optimizations are special cases of the suppressions optimization (the fourth). All of the optimizations defer creating and checking invariants that can be logically inferred from instantiated invariants. In these special cases, the three optimizations improve on the suppressions optimization's v^3 space and time requirement by only using linear or constant memory and space. In addition, these three optimizations only add minimal overhead, guaranteeing that the benefits of the optimizations will outweigh any costs when compared to the v^3 time and space requirements of Daikon.

In contrast to the special case optimizations, the suppression optimization tries to solve the general problem of processing implied invariants. The optimal space implementation uses v^3 time, which is comparable to that of Daikon, but does have

the benefit of constant memory use over Daikon's v^3 requirement. The optimization does have non-trivial overhead (discussed in chapter 4). Given the time costs and overhead of the suppressions optimization, the logical step is to investigate further to understand its performance issues.

2.3.6 Determining the truth of an invariant

While the optimizations do improve the scalability of Daikon, they also complicate the method Daikon uses to determine whether a concrete invariant in its grammar set is true during processing. Since the simple algorithm instantiates all possible concrete invariants in the grammar set at the beginning and removes the concrete invariants as they become falsified by samples, the presence of an invariant implies that the invariant is true while the absence of an invariant signifies that the invariant is false.

With the addition of the optimizations however, Daikon does not create some true invariants in its grammar set because they are implied by invariants that are created by Daikon. Consequently, while the existence of an invariant does imply it is true, the absence of the invariant does not imply that it is false. Instead, Daikon must perform several checks to determine whether a concrete invariant in its grammar set is true:

1. Check for the existence of the invariant. If the invariant exists, it is true.

2. Check to see if all of the variables in the invariant are constants. If so, check the values of the constant variables against the invariant to determine if the invariant is true.

3. Check to see if the variables are the non-leaders of equality sets. If so, check for the invariant over the leaders of the equality sets. If the invariant exists over the leaders of the corresponding invariant variables, then the invariant is true.

4. Check to see if the invariant is true at a higher program point. If so, then the invariant is true. Since Daikon processes at the leaves of the tree, this problem does not come into play until processing is complete.

5. Check to see if the invariant is the consequent of a suppression. If so, check if the antecedent of the invariant is true. If the invariants in the antecedent are true, then the invariant is true.

6. If all of the above checks fail, then the invariant must be false.

Chapter 3

Suppression Optimization

This chapter explains some terminology used in the rest of the thesis and discusses the data structures and approach used by Daikon to implement the suppression optimization.

3.1 Overview

Recall that in the simple algorithm, Daikon must instantiate all of the invariant templates in the grammar over each possible combination of variables and check each invariant against every sample. The *suppression optimization* states that invariants that are logically suppressed by other invariants do not need to be created or checked by Daikon. For example,

$x > y \Longrightarrow x \geq y$,
$x = y \Longrightarrow x \geq y$, and
$0 \leq x < y$ and $z = 0 \Longrightarrow x$ div $y = z$ (div refers to integer division).

3.2 Problem definition

As a simple example, suppose that a program point has the true invariants $x = y$ and $x \geq y$, and Daikon knows that $x = y \Longrightarrow x \geq y$. Using the suppression optimization,

Daikon only creates and checks samples against $x = y$. Daikon sees and applies the samples (1,1), (2,2) and $x = y$ is still true. Daikon then sees and applies the sample (2,1) which conflicts with $x = y$, so $x = y$ no longer describes the program point. Daikon invalidates $x = y$, and checks if the suppressed $x \geq y$ is true. The invariant $x \geq y$ is still true, so Daikon creates that invariant and checks all subsequent samples against the invariant $x \geq y$.

The goal of the suppression optimization is to neither instantiate nor check suppressed invariants. The two approaches to implementing the optimization sacrifice either time or space (section 2.3.4).

Using the example above, the optimal time approach may create all of the true invariants and store a link between the invariants $x = y$ and $x \geq y$. The approach avoids checking the implied invariant $x \geq y$, until $x = y$ becomes false and is removed from the data structure by Daikon. However, Daikon must create and store both the implied invariants and the relationship. This approach fails on the goal of not instantiating suppressed invariants.

The optimal space approach does not build this relationship data structure. Instead the optimal space approach must search for these implied invariants during suppression processing using information variable information from the falsified invariants to figure out which suppressed invariants may be no longer implied. This approach satisfies the goal of neither instantiating nor checking suppressed invariants.

I discussed the decision to prioritize saving memory over time in section 1.3, which led to the decision to use the optimal space approach.

3.3 Invariants and suppressions

Like definition of *invariant* in section 2.1, all of the following terms have a *template* and *concrete* form. Since invariants are the building blocks for the following terms, the template form of each term exists when I use invariant templates and the concrete form of each term exists when I use concrete invariants in the thesis.

Recall that for the rest of the thesis, I will use the term *invariant* to refer to

concrete invariants and will use the term *invariant template* explicitly to differentiate between the two terms. In addition, I will use x, y, and z to compose concrete invariants and α, β, and γ to make up invariant templates. Similar to the differentiation of the two forms of the invariant, when I use the following terms by themselves, the *concrete* adjective is implicit and I will explicitly say *template* when refering to the template form.

An ***antecedent term*** is a single term in a logical implication. An antecedent term is an invariant from Daikon's invariant grammar. A conjunction of one or more antecedent terms makes up the ***antecedent***, the left side of a logical implication. Each of the antecedent terms in the antecedent must be true in order for the antecedent to be true.

For example, the antecedent term templates (boxed)

$$\boxed{(\alpha < \beta)} \wedge \boxed{(\alpha \geq 0)} \wedge \boxed{(\gamma = 0)} \Longrightarrow (\alpha \text{ div } \beta = \gamma)$$

make up the antecedent template

$$\boxed{(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0)} \Longrightarrow (\alpha \text{ div } \beta = \gamma)$$

The ***consequent*** refers to the right side of the implication. A consequent is also an invariant from Daikon's invariant grammar, and in this case the consequent template is:

$$(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow \boxed{(\alpha \text{ div } \beta = \gamma)}$$

A ***suppression*** defines a single logical implication. Using the antecedent and consequent examples earlier, the suppression template is:

$$\boxed{(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)}$$

$$(\alpha < \beta) \Longrightarrow (\alpha \leq \beta)$$

$$(\alpha > \beta) \Longrightarrow (\alpha \geq \beta)$$
$$(\alpha = \beta) \Longrightarrow (\alpha \geq \beta)$$

$$(\alpha > 0) \Longrightarrow (\alpha \geq 0)$$

$$(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$$
$$(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$$

Figure 3-1: These suppression sets will be used to set up examples in chapters 4 and 5.

Multiple suppressions with the same consequent form a ***suppression set.*** If any of the suppressions in a suppression set is true, the consequent is implied. Here is an example of a suppression set template:

$$(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$$
$$(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$$

Written in disjunctive normal form, the above suppression set template would look like:

$$[(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0)] \vee [(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1)] \Longrightarrow (\alpha \text{ div } \beta = \gamma)$$

Daikon has a predefined set of these suppression set templates that are used during suppression processing. From the templates, Daikon builds a map from each invariant type to the suppression set templates that contain the invariant type in any of its antecedent terms. From the suppression set templates shown in figure 3-1, Daikon builds the map show in figure 3-2.

Since these are templates, the amount of memory does not vary with the number of concrete invariants or with the number of implied concrete invariants. Processing, on the other hand, becomes more complex because Daikon must fit concrete invariants to these templates during runtime.

3.4 Approach

Recall that Daikon ignores suppressed redundant invariants during processing to save space, but as samples falsify invariants which are potential antecedents, Daikon needs an algorithm to identify when it needs to unsuppress and create an implied invariant.

The approach of the algorithms used by Daikon is simple: after applying a sample to the invariants, Daikon looks for the concrete consequent invariants that are no longer implied. In these consequent invariants, one or more antecedents were true prior to this sample *and* each of those antecedents was falsified by this sample. In order to find these consequents, Daikon needs to find suppressions where one or more of the antecedent terms has been falsified. The consequents of these suppressions are candidates for being instantiated by Daikon. Daikon has the following information available:

1. a list of predefined suppression set templates
2. a list of all instantiated invariants
3. a list of all instantiated invariants falsified by the sample
4. a list of predefined invariant templates
5. a map from invariant type to relevant suppression sets
6. the current sample

The next two chapters present two algorithms currently used by Daikon to find the invariants that should be unsuppressed. Then chapter 6 describes how I used the two algorithms to improve the performance of Daikon.

Invariant type	Relevant suppression sets
$\alpha < \beta$	$(\alpha < \beta) \Longrightarrow (\alpha \leq \beta)$ $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$ $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
$\alpha = \beta$	$(\alpha > \beta) \Longrightarrow (\alpha \geq \beta)$ $(\alpha = \beta) \Longrightarrow (\alpha \geq \beta)$ $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$ $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
$\alpha > \beta$	$(\alpha > \beta) \Longrightarrow (\alpha \geq \beta)$ $(\alpha = \beta) \Longrightarrow (\alpha \geq \beta)$
$\alpha > 0$	$(\alpha > 0) \Longrightarrow (\alpha \geq 0)$
$\alpha \geq 0$	$(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$ $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
$\gamma = 0$	$(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$ $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
$\beta \neq 0$	$(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$ $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
$\gamma = 1$	$(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$ $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

Figure 3-2: Daikon uses the predefined suppression templates to build a map from each invariant type to the suppression set templates that have at least one antecedent term template with that invariant type.

Chapter 4

Antecedents Algorithm

One algorithm for supporting suppressions is the *antecedents algorithm.* Recall that the goal is to find the invariants that were implied prior to this sample but are no longer implied. The antecedents algorithm achieves this goal by finding all of the concrete suppressions with at least one falsified invariant in the antecedent. The consequents of these suppressions are the candidate consequents that may need to be created by Daikon. The consequent should be instantiated if it is not suppressed by a different suppression.

The following sections detail the implementation of the antecedents algorithm. In preparation, the algorithm reverses the dynamic constants optimization and creates all invariants over constant variables (section 4.2). Then, for each suppression template in the predfined suppression templates, Daikon finds the list of invariants that match the invariant type of each antecedent term in that suppression (section 4.3). After finding the lists, Daikon systematically selects an invariant from each list and checks to see if the combination produces a non-conflicting suppression, by verifying the bindings to the variables in the suppression template (section 4.4). Finally, when Daikon finds a complete suppression, it creates the consequent if no other suppression for the consequent still holds (section 4.5). Figure 4-1 provides the pseudocode for the outline of the algorithm, more detailed pseudocode appear in the section detailing each step in the algorithm.

Information available:
1. a list of predefined suppression templates (suppression_templates[])
2. a list of all instantiated invariants (invs[])
3. a list of all instantiated invariants falsified by the sample (false[])
4. a list of predefined invariant templates (inv_templates[])
5. a map from invariant type to relevant suppression sets (M(i,s))
6. the current sample

Algorithm:

```
1.  create the invariants over the constant variables
2.  organize invariants by invariant type

3.  for suppression_template s in suppression_templates[]:
    a.  match invariants to antecedent term templates in s
    b.  perform the cross product of invariants
        to find concrete suppressions
    c.  check the consequents of the suppressions for resuppression
```

Output: a list of the concrete invariants unsuppressed by the sample

Figure 4-1: Pseudocode for the antecedents algorithm.

4.1 Running example

Using the suppression sets templates and concrete invariants shown in 4-2, I will demonstrate the antecedents algorithm. Figure 4-2 shows the state of the invariants before Daikon applies a sample to the current set of true invariants. After applying the sample, some invariants get falsified by the sample (see figure 4-3). Now, Daikon must find the implied consequents that are affected by these falsified invariants.

4.2 Finding all of the true invariants

The antecedents algorithm works by finding all of the concrete suppressions where at least one of the invariants is falsified. To find all of the suppressions, the algorithm enumerates all possible concrete suppressions using the set of true invariants, and this work requires that all true invariants be present in order to check the suppressions. More specifically, all true invariants that are potential antecedent terms must be present. The requirement creates a complication because Daikon does not create

Suppression Sets:

Suppression A: $(\alpha > 0) \Longrightarrow (\alpha \geq 0)$

Suppression B: $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
Suppression C: $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

Variables:

int x, int y, int z, int q, int r

Sample:

$x = 2, y = 1, z = 5, q = 0, r = 0$

Invariants:

$y > z$ (true)
$x < y$ (true)
$r < y$ (true)
$x = 0$ (true)
$y \neq 0$ (true)
$x > 0$ (true)
$y > 0$ (true)
$r = 0$ (true)
$q = 0$ (true)
$r = q$ (true)

Figure 4-2: Running example for the antecedents algorithm (before applying sample)

redundant invariants because of the optimizations.

Two of the optimizations, variable hierarchy and equality sets, do not affect this step. The variable hierarchy optimization relates invariants at different program points, so it does not affect invariant processing at a particular program point. For the equality set optimization, the equality set leaders represent the elements in their sets at all times. Any antecedent applicable to the leader is also also applicable to its non-leaders and any unsuppressed consequents over the leaders will apply to the non-leaders. Thus, Daikon does not need to create the invariants over the non-leaders for the antecedents algorithm.

The dynamic constants optimization does affect the antecedents algorithm be-

Suppression Sets:

Suppression A: $(\alpha > 0) \Longrightarrow (\alpha \geq 0)$

Suppression B: $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
Suppression C: $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

Variables:

`int` x, `int` y, `int` z, `int` q, `int` r

Sample:

$x = 2, y = 1, z = 5, q = 0, r = 0$

Invariants:

$y > z$ (falsified)
$x < y$ (falsified)
$r < y$ (falsified)
$x = 0$ (falsified)
$y \neq 0$ (true)
$x > 0$ (true)
$y > 0$ (true)
$r = 0$ (true)
$q = 0$ (true)
$r = q$ (true)

Figure 4-3: Running example for the antecedents algorithm (after applying sample)

cause the invariants of constant variables can be potential antecedent terms. The antecedents algorithm does not use the method described in section 2.3.6 to determine the truth of invariants over constants because in cases where Daikon needs to check the constant variable value against the same invariant (e.g. $x = 1$) several times, Daikon does redundant work. In these cases, it is more efficient for Daikon just to create all of the invariants over constant variables (pseudocode shown below).

```
1.  create the invariants over the constant variables
   a.  for variable v_i in variables of program point
      i.  create unary invariants over v_i from inv_templates[]
   b.  for variable v_i in variables of program point
      i.  for variable v_j in variables of program point
         create binary invariants over v_i, v_j from invs_templates[]
```

The antecedents algorithm first creates all unary and binary invariants from the Daikon grammar set over the constant variables (ternary invariants are never antecedent terms). Thus, invariants $q = 0$, $r = 0$, $r = q$ in the running example, are not present before suppression processing. Daikon creates these invariants over constants specifically for suppression processing and discards them afterwards.

The suppressions optimization also affects the antecedents algorithm. A *chain suppression* is a set of two or more suppressions where the antecedent term of one suppression is also the consequent of another suppression. The antecedents algorithm requires that all true invariants that are potential antecedent terms be present, including the implied invariants.

In the example, the antecedent term template ($\alpha \geq 0$) of Suppression B, appears as the consequent template of Suppression A. A problem arises if Daikon incorrectly creates a concrete invariant from the consequent template (α div $\beta = \gamma$) because the concrete invariant over $\alpha \geq 0$ is not present to suppress the consequent when the antecedents algorithm runs.

The problem of creating suppressed invariants when they are still implied does not apply to creating invariants over constant variables because those invariants are temporary and discarded at the end of suppression processing. The problem only applies to the invariants that are no longer implied and get created by Daikon during suppression processing. In the problem mentioned above, these newly created invariants should not be created by Daikon because they are still implied by other true invariants.

There are several ways to solve the problem of chain suppressions. One possibility,

similar to the solution to the constants problem, is to create all of the suppressed invariants. This solution is both memory and time intensive because Daikon must create, store and process all of these suppressed invariants. A second solution is to use the method described in section 2.3.6, verifying the truth of the antecedents of an invariant that is a potential antecedent term. This solution is time intensive, because Daikon must perform this verification for all absent invariants encountered in the process of checking whether an invariant is implied by other true invariants.

In order to save memory and making processing simple, Daikon solves the problem by automatically propagating chain suppressions in template form before any suppression processing, hence making the implicit suppressions explicit. The pseudocode for this process is shown below. Note that all propagations are done in template form.

```
Available information used:
1. a list of predefined suppression set templates (suppression_templates[])
2. a map from invariant type to relevant suppression sets (M(i,s))

for each suppression set s in suppression_templates[]:
   1.  find s.consequent
   2.  suppression sets v[] = M(i,s).get(s.consequent.inv_type)
   3.  if v[] == null, end (//the invariant is not an antecedent term)
   4.  if v[] != null:
   for each suppression set t in v[]:
      a.  find the suppression x in t with s.consequent
          as an antecedent term
      b.  for each suppression y in s:
          i.  make a new suppression in t, substituting y.antecedent
              for s.consequent in x
```

Since Daikon expands the suppressions in template form, memory usage is still constant. The propagation has the added benefit that any algorithm using the tem-

plates already has the solution. If the solution were built into the algorithm itself, then each algorithm would need to create its own solution. In the example above, Daikon would add suppression D to the set:

Suppression A: $(\alpha > 0) \Longrightarrow (\alpha \geq 0)$

Suppression B: $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

Suppression C: $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

Suppression D: $(\alpha < \beta) \wedge (\alpha > 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

By rolling out the chain suppressions, Daikon does not need to trace back and look for the antecedents of antecedent terms because all possible antecedents of the consequent will exist in template form.

4.3 Matching invariants to antecedent terms

Daikon first builds a mapping from each invariant type to all of the invariants of that type. Daikon can prune the map by removing invariant types that do not appear in antecedent terms. When Daikon needs the invariants of an antecedent term template, it looks up the list in the map. The pseudocode for building this mapping is shown below.

```
2.  organize invariants by invariant type
   a.  make map m of inv_type → invariant[]
   b.  for inv in invs[]
      ii.  m.put(inv.inv_type, map.get(inv.inv_type).add(inv))
```

For each antecedent term template in each suppression template, Daikon finds all of the invariants of the same type as the antecedent term template. After examining all of the invariants, Daikon checks to see that each antecedent term template has a non-empty list of invariants. In order for a consequent to be suppressed, there must be an invariant matching each of the antecedent term templates in the suppression

Invariant type	$\alpha > 0$	$\alpha < \beta$	$\alpha \geq 0$	$\alpha = 0$	$\alpha = \beta$	$\beta \neq 0$	$\gamma = 1$
Invariants	$x > 0$ $y > 0$	$x < y$ $r < y$		$x = 0$ $q = 0$ $r = 0$	$r = q$	$y \neq 0$	

Figure 4-4: Organization of available invariants by invariant type

template. Otherwise, the implication cannot hold. Thus, if an antecedent term template has no matching invariants, then no concrete consequent can be unsuppressed. In addition, there must be at least one falsified invariant in all of the lists of invariants of the antecedent term templates. If all of the invariants are true, no concrete consequent can be unsuppressed. The pseudocode for the matching of invariants to antecedent term templates is shown below.

```
3a.  match invariants to antecedent term templates
for each antecedent term template t in suppression_template s
   i.  find t.inv_type
   ii.  (inv_type → invariant[]).get(t.inv_type)
```

Starting with the available invariants:

$y > z$, $x < y$, $r < y$, $y \neq 0$, $x > 0$, $y > 0$, $r = 0$, $q = 0$, $r = q$,

Daikon classifies each invariant according to invariant type and then for each suppression template, looks up the invariants of each antecedent term template by invariant type. Figure 4-4 shows the results of organizing the invariants by invariant type. Below are the results of matching for suppression templates A through D. Suppression A has no falsified invariants in the invariants of its antecedent term template, so it can not unsuppress any consequents. Suppressions B and C has antecedent term templates with no matching invariants, so no consequent can be unsuppressed using these suppressions. Thus Daikon only processes suppression D.

Antecedent terms	$(\alpha > 0)$
Invariants	$x > 0$ (true)
	$y > 0$ (true)

Results of matching antecedent term templates to invariants in Suppression A.

Antecedent terms	$(\alpha < \beta)$	$(\alpha \geq 0)$	$(\gamma = 0)$
Invariants	$x < y$ (false)		$x = 0$ (false)
	$r < y$ (true)		$q = 0$ (true)
			$r = 0$ (true)

Results of matching antecedent term templates to invariants in Suppression B.

Antecedent terms	$(\alpha = \beta)$	$(\beta \neq 0)$	$(\gamma = 1)$
Invariants	$q = r(true)$	$y \neq 0$ (true)	

Results of matching antecedent term templates to invariants in Suppression C.

Antecedent terms	$(\alpha < \beta)$	$(\alpha > 0)$	$(\gamma = 0)$
Invariants	$x < y$ (false)	$y > 0$ (true)	$x = 0$ (false)
	$r < y$ (true)	$x > 0$ (true)	$q = 0$ (true)
			$r = 0$ (true)

Results of matching antecedent term templates to invariants in Suppression D.

4.4 Taking the cross product of antecedent term invariants to find potential consequents

Using the results of matching, Daikon takes the cross product between the invariants from each antecedent term template and checks for potential unsuppressed consequents (see pseudocode below). If there is a suppression with n antecedent terms,

and each term has m matching invariants, there would be m^n n-dimensional products. Daikon potentially must examine all m^n combinations but does save work through early pruning. Going through the list of antecedent terms templates, Daikon takes an invariant from the list of matching invariants for a particular antecedent term template and determines if the currently selected invariant can be used as part of the suppression based on the previously invariants selected.

```
3b.  perform the cross product of invariants to find concrete
   suppressions
   i.  enumerate the cross product of invariants from each
      antecedent term template
   ii.  for each combination of invariants in the cross product
      - check that the combination is valid
      (no variable binding conflicts)
```

Not all combinations of invariants will be valid because the variables in the suppression templates form constraints on the program variables that can be bound to them. For example, the appearance of α in two antecedent term templates of a suppression template constrains the same program variable to be bound to the free variable α. When Daikon selects an invariant from the invariant list of an antecedent term template, it binds the concrete invariant variables are bound to the free variables of that antecedent term template.

Invalid combinations occur when two different program variables must be bound to the same free variable. In addition, when the same program variable must be bound to two different free variables, Daikon ignores these combinations and treat them as conflicts because these invariants are not interesting and often degenerate into a simpler invariant. For example, the invariant $\alpha \wedge \beta = \gamma$ (logic and) reduces to the invariant $\alpha = \gamma$ when α and β are bound to the same variable. Variable conflicts allow for early pruning and saves time because Daikon does not need to check not all variable combinations.

In the following sequence, we continue to simulate Daikon on the running ex-

ample by performing the cross product on Suppression D. The currently selected invariants are shown in bold. In step 1, as shown below, Daikon looks at the first antecedent term template in the suppression template and selects the first invariant in the matching list, $x < y$. In order for the term template and invariant to match, α must be bound to x and β must be bound to y. No conflict arises, so Daikon continues.

Step 1:

Antecedent terms templates	$(\alpha < \beta)$	$(\alpha > 0)$	$(\gamma = 0)$	Bindings
Invariants	$\boldsymbol{x < y}$ **(*false*)** $r < y$ (false)	$y > 0$ (true) $x > 0$ (false)	$x = 0$ (false) $q = 0$ (true) $r = 0$ (true)	(α, x); (β, y)

In step 2, Daikon chooses the first matching invariant, $y > 0$, for the second antecedent term template. In order for the variables to match between the term template and the invariant, α must be bound to y, which causes a conflict because α is already bound to x from the previously selected invariant. Thus, this combination of invariants cannot form a consistent suppression. Here we see an example of early pruning: Daikon does not check any more invariants in the rest of the antecedent term templates with the invariants $x < y$ and $y > 0$ in combination.

Step 2:

Antecedent terms templates	$(\alpha < \beta)$	$(\alpha > 0)$	$(\gamma = 0)$	Bindings
Invariants	$\boldsymbol{x < y}$ **(*false*)** $r < y$ (true)	$\boldsymbol{y > 0}$ **(*true*)** $x > 0$ (false)	$x = 0$ (false) $q = 0$ (true) $r = 0$ (true)	(α, x); (β, y); (α, y) - Conflict

Since there are more invariants matching the second antecedent term template, in step 3, shown below, Daikon tries a different invariant from the same list, $x > 0$. The invariant forces α to be bound to x, which does not conflict with previous bindings, so Daikon advances to the next antecedent term template.

Step 3:

Antecedent terms templates	$(\alpha < \beta)$	$(\alpha > 0)$	$(\gamma = 0)$	Bindings
Invariants	$\boldsymbol{x < y}$ **(*false*)** $r < y$ (true)	$y > 0$ (true) $\boldsymbol{x > 0}$ **(*false*)**	$x = 0$ (false) $q = 0$ (true) $r = 0$ (true)	(α, x); (β, y) (α, x)

In step 4, Daikon selects $x = 0$, which forces γ to be bound to x. However, that conflict arises because x cannot be bound to both α and γ (the invariant is not interesting), so Daikon concludes that this set of invariants does not form a complete suppression.

Step 4:

Antecedent terms templates	$(\alpha < \beta)$	$(\alpha > 0)$	$(\gamma = 0)$	Bindings
Invariants	$\boldsymbol{x < y}$ **(*false*)** $r < y$ (true)	$y > 0$ (true) $\boldsymbol{x > 0}$ **(*false*)**	$\boldsymbol{x = 0}$ **(*false*)** $q = 0$ (true) $r = 0$ (true)	(α, x); (β, y); (α, x); (γ, r) – Conflict

Since Daikon saw a conflict in the previous step, in step 5, Daikon moves on to the next invariant in the antecedent term template's list of matching invariants, selecting

$q = 0$. The selection binds γ to q, and no conflict arises.

Step 5:

Antecedent terms templates	$(\alpha < \beta)$	$(\alpha > 0)$	$(\gamma = 0)$	Bindings
Invariants	$\boldsymbol{x < y}$ **(*false*)** $r < y$ (true)	$y > 0$ (true) $\boldsymbol{x > 0}$ **(*false*)**	$x = 0$ (false) $\boldsymbol{q = 0}$ **(*true*)** $r = 0$ (true)	(α, x); (β, y); (α, x); (γ, q) – Complete

Since Daikon reaches the end of the list of antecedent term templates and finds no conflicts with the currently selected group of invariants, the suppression is complete. In this example, Daikon finds:

$$x < y \wedge x > 0 \wedge q = 0 \Longrightarrow x \text{ div } y = q$$

If we continue with the example and complete the rest of the cross product, the following steps will occur. We need to continue even after finding the first match because other combinations of invariants may create complete suppressions.

Step 6:

Antecedent terms templates	$(\alpha < \beta)$	$(\alpha > 0)$	$(\gamma = 0)$	Bindings
Invariants	$x < y$ (false) $\boldsymbol{r < y}$ **(*true*)**	$y > 0$ (true) $x > 0$ (false)	$x = 0$ (true) $q = 0$ (true) $r = 0$ (true)	(α, r); (β, y);

Step 7:

Antecedent terms templates	$(\alpha < \beta)$	$(\alpha > 0)$	$(\gamma = 0)$	Bindings
Invariants	$x < y$ (false) $\boldsymbol{r < y}$ **(true)**	$\boldsymbol{y > 0}$ **(true)** $x > 0$ (false)	$x = 0$ (true) $q = 0$ (true) $r = 0$ (true)	(α, r); (β, y); (α, y) - Conflict

Step 8:

Antecedent terms templates	$(\alpha < \beta)$	$(\alpha > 0)$	$(\gamma = 0)$	Bindings
Invariants	$x < y$ (false) $\boldsymbol{r < y}$ **(true)**	$y > 0$ (true) $\boldsymbol{x > 0}$ **(false)**	$x = 0$ (true) $q = 0$ (true)	(α, r); (β, y); (α, x) - Conflict

Since there are no more invariants to try in the $\alpha > 0$ column, Daikon stops.

4.5 Creating the consequent

After finding a complete concrete suppression, Daikon checks the state of each invariant. If all are true, then the antecedent is still true and suppresses the consequent. If any are false, then this suppression is no longer valid and Daikon may need to create the corresponding consequent (in our example, x div $y = q$). The inclusion of one or more falsified invariants means that this suppression was true previous to this sample but is false now. The pseudocode for this step is shown below.

```
3c.  check the consequents of the suppressions for resuppression
   i.  check that at least one invariant in the suppression
       is in false[]
   ii.  check that the concrete consequent invariant is valid
       over the variables in the bindings
   iii.  check that no other suppression in the
       suppression set of the concrete consequent is valid
   iv.  check that the sample satisfies the consequent invariant
   v.  if all checks pass, create the consequent invariant
```

Daikon verifies three other things before actually creating the consequent. First, Daikon checks that the consequent invariant is valid over the variables in the bindings. Second, Daikon checks to see if the consequent is still suppressed by another suppression in the suppression set. Daikon checks this fact by seeing if a set of true invariants form a valid suppression for the consequent. If such a set exist, then Daikon knows that the consequent can still be deduced from true invariants and does not need to create it. Third, Daikon checks whether the current sample will falsify the invariant. There is no value in creating an invariant that gets falsified immediately. If the consequent passes all three tests, then Daikon creates the newly unsuppressed invariant.

4.5.1 Checking for valid variable types

Daikon verifies that the invariant is applicable over the variables in the bindings by checking the valid variable types of the invariant template against the types of the variables. In the example, the consequent passes the test because the invariant template α div $\beta = \gamma$ is valid over integers, and the variables x, y, and q are also integers.

4.5.2 Checking the consequent for resuppression

In the continuing example, since Daikon is processing Suppression D and has found a complete suppression, it must check that other concrete suppressions in the suppression set are not valid, namely those defined by suppression templates B and C:

B: $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

C: $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

more specifically, given the variable bindings of the complete suppression: (α, x), (β, y), and (γ, q)

B: $(x < y) \wedge (x \geq 0) \wedge (q = 0) \Longrightarrow (x \text{ div } y = q)$

or

C: $(x = y) \wedge (y \neq 0) \wedge (q = 1) \Longrightarrow (x \text{ div } y = q)$

must be true to suppress the consequent x div $y = q$.

In the concrete suppression formed from suppression template B, the invariants $x \geq 0$ and $q = 0$ are true but $x < y$ is false, so suppression B is not able to suppress the consequent.

In the concrete suppression formed from suppression template C, the invariant $y \neq 0$ is true, but not $q = 1$ and $x = y$. Suppression C is cannot suppress the consequent. The consequent passes the test.

4.5.3 Applying sample to consequent

Applying the sample ($x = 2$, $y = 1$, $z = 5$, $q = 0$, $r = 0$), to the consequent x div $y = q$ falsifies the consequent. Daikon does not create the invariant.

4.6 Early pruning

While I have described the basic implementation of the antecedents algorithm, there is a nuance that should be addressed. When taking the cross product of the invariants, as described in section 4.4, Daikon can terminate early on a certain branch if it finds conflicts in variable bindings.

Daikon developers facilitate early pruning by ordering the antecedent term templates in the suppression templates. Since Daikon checks the antecedent term templates in the order that they appear in the suppression template, Daikon developers arrange the antecedent term templates such that Daikon checks the terms involving more variables first and binds those variables. More specifically, Daikon will select binary invariants and binds their variables before those of unary ones. The more variables involved in the invariant, the higher the likelihood of a conflict with previous assignments.

Chapter 5

Sequential Algorithm

Another algorithm for supporting suppressions is the *sequential algorithm.* Recall that the goal is to find the invariants that were suppressed previous to the sample and not suppressed after applying the sample. The sequential algorithm's approach is to process the falsified invariants one by one and look for suppressions where the falsified invariant is one of the antecedent terms and all of the other antecedent terms are true. This ensures that the suppression was valid previously and is invalid now. The consequent should be instantiated if it is not suppressed by a different suppression.

The following sections detail the implementation of the sequential algorithm. In the first step, since only falsified invariants can unsuppress consequents, the algorithm looks for the suppression sets that contain an antecedent term of the same invariant type as the falsified invariant. Next, using the templates for the consequent in these suppression sets, the algorithm creates the candidate concrete consequents by using the variables from the falsified invariant (5.2), similar to using variable bindings to create the consequents in the antecedents algorithm. Lastly, the algorithm creates the consequent if no other suppression for the consequent still holds and the falsified invariant is the only antecedent term in the antecedent that is false (5.3). Figure 5-1 provides the pseudocode for the algorithm.

Information available:
1. a list of predefined suppression templates (suppression_templates[])
2. a list of all instantiated invariants (invs[])
3. a list of all instantiated invariants falsified by the sample (false[])
4. a list of predefined invariant templates (inv_templates[])
5. a map from invariant type to relevant suppression sets (M(i,s))
6. the current sample

Algorithm:

```
for each invariant inv in false[]:
1.  look for the relevant suppression sets
    s[] = (M(i,s)).get(inv.inv_type)

2.  for each relevant suppression set x in s[]:
    a.  use concrete variables from inv to compose concrete x.consequent
    b.  check the concrete x.consequent for resuppression
```

End: a list of the concrete invariants unsuppressed by the sample

Figure 5-1: Pseudocode for the sequential algorithm.

5.1 Running example

Using the suppression sets templates and concrete invariants shown in 5-2, I will demonstrate the sequential algorithm. Figure 5-2 shows the state of the invariants before Daikon applies a sample to the current set of true invariants. After applying the sample, some invariants get falsified by the sample (see figure 5-3). Now, Daikon must find the implied consequents that are affected by these falsified invariants.

5.2 Finding relevant suppressions and variable sets to identify potential consequents

The sequential algorithm processes each falsified invariant one at a time. Since only falsified invariants may unsuppress invariants, the algorithm first looks for the suppression set templates that are relevant to these invariant types. Recall that Daikon has a map from the invariant type to the suppression sets that have an antecedent term of that type. Using this map (shown in figure 3-2, Daikon can immediately

Suppression Sets:

Suppression A: $(\alpha < \beta) \Longrightarrow (\alpha \leq \beta)$

Suppression B: $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
Suppression C: $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

Suppression D: $(\alpha > \beta) \Longrightarrow (\alpha \geq \beta)$

Variables:

int x, int y, int z, int q

Sample:

$x = 0, y = -5, z = 5, q = 0$

Invariants:

$y > z$ (true)
$x < y$ (true)
$z < y$ (true)
$y \leq 0$ (true)
$x \leq 0$ (true)
$y \neq 0$ (true)
$x = 0$ (true)
$q = 0$ (true)
$x = q$ (true)

Figure 5-2: Running example for the sequential algorithm (before applying sample)

narrow down the possible consequents that may be unsuppressed by the falsified invariant. In the running example, we will focus on the invariant $x < y$. Looking in the map, Daikon finds the suppression set templates that contain the antecedent term template $(\alpha < \beta)$:

Suppression A: $(\alpha < \beta) \Longrightarrow (\alpha \leq \beta)$

Suppression B: $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
Suppression C: $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

Suppression Sets:

Suppression A: $(\alpha < \beta) \Longrightarrow (\alpha \leq \beta)$

Suppression B: $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
Suppression C: $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

Suppression D: $(\alpha > \beta) \Longrightarrow (\alpha \geq \beta)$

Variables:

`int` x, `int` y, `int` z, `int` q

Sample:

$x = 0, y = -5, z = 5, q = 0$

Invariants:

$y > z$ (falsified)
$x < y$ (falsified)
$z < y$ (falsified)
$y \leq 0$ (falsified)
$x \leq 0$ (true)
$y \neq 0$ (true)
$x = 0$ (true)
$q = 0$ (true)
$x = q$ (true)

Figure 5-3: Running example for the sequential algorithm (after applying sample)

Using the suppression sets, the algorithm then finds the concrete consequents that can be unsuppressed by the falsified invariant. Since only a template exists in the suppression set, Daikon uses the program variables from the falsified invariant to construct the concrete consequent.

For the first suppression set,

Suppression A: $(\alpha < \beta) \Longrightarrow (\alpha \leq \beta)$

Daikon finds that in order for the invariant $x < y$ to fit the antecedent term template $\alpha < \beta$, the template variables α and β must be bound to x and y respectively.

Applying these bindings to the consequent template, the concrete consequent that may be unsuppressed is $x \leq y$. At this point, Daikon will move on to the next step in the algorithm for first suppression set, but let us follow this stage for the second suppression set:

Suppression B: $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$
Suppression C: $(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$

Daikon again finds that in order for the invariant $x < y$ to fit into the suppression template, the template variables α and β must be bound to x and y respectively. Applying these bindings to the consequent template, the concrete consequent that may be unsuppressed is

$$x \text{ div } y = \gamma$$

In this case, there is a free variable in the consequent template. Daikon binds all possible program variables to the free variable and notes each set of resulting variable bindings as a possible consequent. Since the variables x, y, z and q are available, Daikon notes the consequent invariants x div $y = x$, x div $y = y$, x div $y = z$ and x div $y = q$ as possible consequents. Daikon eliminates the consequents x div $y = x$ and x div $y = y$ immediately because invariants with repeated variables are not interesting (see section 5.4.2).

5.3 Creating the consequent

Before creating the consequents, the checks made by the sequential algorithm are almost exactly the same as the steps taken in the antecedents algorithm in section 4.5. However, checking for valid variable types in the consequent, suppression of the consequent by another suppression, falsification of consequent by the current sample (described in section 4.5), is not sufficient for deciding whether the consequent should

be created by the sequential algorithm.

Because the sequential algorithm processes the falsified invariants one by one, a problem occurs when the same sample falsifies two invariants from the same suppression. Take for example, the concrete suppression (suppose that it is the only suppression in the suppression set):

$$(a < b) \wedge (a \geq 0) \Longrightarrow (a \text{ div } b = 0)$$

If the sample only falsifies one invariant, e.g. $a < b$, the process is simple. The algorithm checks to see if the other invariant $(a \geq 0)$ is true or false. If the invariant is true, then the algorithm knows that $a < b$ is the only falsified antecedent term, so it should create the consequent. If the invariant $a \geq 0$ is false, then the algorithm does not create the consequent because the algorithm already considered the consequent when a sample falsified the antecedent term $(a \geq 0)$.

However, consider the case when the sample falsifies both invariants in the antecedent, $a < b$ and $a \geq 0$. When the algorithm processes, $a < b$, it considers whether to create the consequent a div $b = 0$. Daikon sees that the invariant in the suppression, $a \geq 0$, is false and does not create the consequent. Similarly, when Daikon processes the invariant $a \geq 0$, it does not create the consequent because Daikon mistakenly thinks that it already considered the consequent when $a < b$ was falsified by a sample. As a result, Daikon never creates the consequent. This problem is a correctness issue.

Daikon solves the problem by treating the unprocessed falsified invariants as true. The change solves the problem because Daikon now sees the first falsified invariant in the antecedent to be processed as the only falsified antecedent term in the antecedent and creates the consequent. When Daikon processes the second falsified invariant in the antecedent, it correctly rejects the consequent because the first antecedent term is false (Daikon discards the falsified invariants after they have been processed).

With the change, even if the same sample falsifies the invariants $a < b$ and $a \geq 0$, Daikon correctly considers the consequent. Daikon first processes the invariant $a < b$,

creates the consequent a div $b = 0$, and then removes the falsified invariant $a < b$. When Daikon now processes the invariant $a \geq 0$, it recognizes that $a \geq 0$ is not the only falsified antecedent term in the antecedent because the invariant $a < b$ is also false.

In the running example, Daikon needs to create the concrete consequents $x \leq y$, x div $y = z$ and x div $y = q$. In the following sections, I will not explain the checks that are common to both the antecedents and sequential algorithm, but will instead refer to the applicable sections in the antecedents algorithm chapter. I will just show the application of the checks to the three candidate consequents.

5.3.1 Checking for valid variable types

See section 4.5.1.

Consequent: $x \leq y$

The variables x and y are integers, which is valid for comparison using $\leq$, so the consequent passes the test.

Consequent: x div $y = z$

The variables x, y, and z are integers, which is valid for the integer division invariant, so the consequent passes the test.

Consequent: x div $y = q$

The variables x, y, and q are integers, which is valid for the integer division invariant, so the consequent passes the test.

5.3.2 Checking the consequent for resuppression

See section 4.5.2.

Consequent: $x \leq y$

There is only one suppression in the suppression set, so the consequent passes the test.

Consequent: $x \text{ div } y = z$

Since there is another suppression in the suppression set, Daikon must check that whether the other suppression is valid:

$$(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma),$$

more concretely

$$(x = y) \wedge (y \neq 0) \wedge (z = 1) \Longrightarrow (x \text{ div } y = z)$$

Since the invariant $x = y$ is not true, the suppression is not valid. Thus, the consequent passes the test.

Consequent: $(x \text{ div } y = q)$

Since there is another suppression in the suppression set, Daikon must check that whether the other suppression is valid:

$$(\alpha = \beta) \wedge (\beta \neq 0) \wedge (\gamma = 1) \Longrightarrow (\alpha \text{ div } \beta = \gamma),$$

more concretely

$$(x = y) \wedge (y \neq 0) \wedge (q = 1) \Longrightarrow (x \text{ div } y = q)$$

Since the invariant $x = y$ is not true, the suppression is not valid. Thus, the consequent passes the test.

5.3.3 Checking the suppression for the only falsified antecedent term

For the running example, let us suppose that Daikon processes the invariant $x < y$ first. The other falsified invariants in the example do not actually participate in the suppressions relevant to $x < y$.

Consequent: $x \leq y$

Since there is only one antecedent term in the antecedent of the suppression, the falsified invariant trivially is the only falsified antecedent encountered by the algorithm. Thus, the consequent passes the test.

Consequent: x div $y = z$

Daikon must check that the other suppression has the falsified invariant as the only false invariant. Thus, in the suppression $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$, Daikon checks whether $x \geq 0$ and $z = 0$ are true. Since $z = 0$ is false, Daikon already considered the consequent when processing $z = 0$. The consequent fails the test and Daikon discards it.

Consequent: x div $y = q$

Now, Daikon must check that the other suppression has the falsified invariant as the only false invariant. Thus, in the suppression $(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma)$, Daikon needs to check that $(x \geq 0)$ and $(q = 0)$ are true and they are, so the consequent $(x \text{ div } y = q)$ fails the test because a valid suppression still exists for the consequent. Daikon discards the consequent.

5.3.4 Applying sample to consequent

Daikon applies the sample ($x = 0$, $y = -5$, $z = 5$, $q = 0$) tot he remaining candidate consequent.

Consequent: $(x \leq y)$

The sample falsifies the consequent so Daikon does not create the consequent.

5.4 Interaction with Daikon optimization and early pruning of variable combinations

I have described the basic implementation of the sequential algorithm, but there are a couple of nuances that I should address.

5.4.1 Interaction with the dynamic constants optimization

Similar to the antecedents algorithm, the dynamic constants optimization (see section 2.3.1) also affects the sequential algorithm. The sequential algorithm needs the invariants over the constant variables because it must check whether true invariants are still suppressing the consequent before creating the consequent. To determine whether an invariant over constant variables is true, in section 5.3, Daikon explicitly checks the constant values of the variables against the invariant to determine whether the invariant holds (see section 2.3.6).

5.4.2 Early pruning of variable combinations

As mentioned in section 5.2, when finding the concrete consequent that may be suppressed by the falsified invariant, Daikon must try all possible program variables as

a binding for any free variables in the consequent template. In the running example, since the variables x, y, z and q are available, Daikon checks the consequent invariants (x div $y = x$), (x div $y = y$), (x div $y = z$) and (x div $y = q$).

Before going through the more expensive checks in section 5.3, Daikon also checks whether the set of variables makes a valid and interesting combination. In examining the validity of a set, Daikon checks whether the combination of variables can be part of an invariant.

For example, in Daikon's grammar, there is no invariant between a variable of type `hashcode` and a variable of type `String`. In addition, variable combinations with repeated variables, such as binding the concrete variables x, y, and x to the free variables α, β, and γ respectively in an invariant, are not interesting because they often degenerate to a simpler form. For example, the invariant $\alpha \wedge \beta = \gamma$ (logic and) reduces to the invariant $\alpha = \gamma$ when α and β are bound to the same variable.

5.5 Comparison to the antecedents algorithm

The antecedents and sequential algorithm are similar in that once they find the concrete consequents that may be unsuppressed, the process of checking whether Daikon needs to create the consequents is the same. Daikon identifies a concrete consequent as a set of variable bindings and a suppression template. The two algorithms differ in the method used to find these variables bindings. The antecedents method creates these bindings exclusively from invariants while the sequential method uses the the variables from the falsified invariants and supplements with available program variables when necessary.

In addition, the algorithms each have one serious disadvantage that makes the algorithm not ideal. The sequential algorithm has the disadvantage of doing redundant work when invariants involved in the same concrete suppression are falsified by the same sample. In the running example involving the suppression

$$(\alpha < \beta) \wedge (\alpha \geq 0) \wedge (\gamma = 0) \Longrightarrow (\alpha \text{ div } \beta = \gamma),$$

because a sample falsifies the invariant $x < y$, Daikon checks the concrete consequent x div $y = z$. If the same sample falsifies the invariant $x \geq 0$, then the sequential algorithm will check the concrete consequent x div $y = z$ again (Daikon binds x to α and substitutes all possible program variables for y and z).

In addition, the sequential algorithm will check invariants over constant variables, like $z = 0$, twice. The antecedents algorithms avoids checking the consequent multiple times by taking the invariants in combination and avoids checking the same invariants over constants by creating all of the invariants over constants once.

However, the creation of the invariants over constant variables turns into a disadvantage for the antecedents algorithm when a sample only falsifies a small number of invariants. The antecedents algorithm requires that all true invariants be present at the beginning of the algorithm and creating these invariants is time consuming due to the number of invariant templates and constant variables. The sequential algorithm avoids this work because the algorithm does not use these invariants to create the variable bindings in the concrete consequent. When the sequential algorithm does need to check whether an invariant over constant variables exists, the algorithm already has the specific variables and can look up a specific invariant easily. Thus, the sequential algorithm avoids the work of creating all invariants over constant variables by only looking up invariants over constants when necessary.

The advantages and disadvantages of each algorithm suggest that the sequential algorithm performs better there is a small number of falsified invariants and the antecedents algorithm is more advantageous when the sample falsifies a large number of invariants. In the case of few falsified invariants, the sequential algorithm avoids the antecedents algorithm's work of creating all of the invariants over constant variables. When there are many falsified invariants, the sequential algorithm avoids the sequential algorithm's redundant work by processing the falsified invariants in combination and creating the invariants over constants only once.

Chapter 6

Performance Improvements

6.1 Overview

My research had two focuses: understanding the algorithms currently available for processing suppressions and exploring ways to enhance the algorithms to improve the overall performance of Daikon. I ran experiments to determine where the majority of the time is spent when processing suppressions. In the following sections, I will explain some of the observations that I made while running the experiments, and how these observations led to the enhancements that I designed to improve the overall performance of Daikon.

I considered three ways of improving performance:

1. Improve the current algorithm for processing suppressions (antecedents algorithm)

2. Use the existing algorithms in the best context possible (hybrid algorithm)

3. Explore a new approach for processing suppressions (batching algorithm)

In section 5.5, I discussed some of the observations that motivated much of the design changes made to the current algorithms. The key observations are that the antecedents algorithm has a large fixed cost because Daikon must create the invariants

over constant variables and that the sequential algorithm does redundant work when there are many falsified invariants.

Based on these observations, I explored improving performance via three methods:

1. I reduced the absolute amount of overhead in the antecedents algorithm.

2. I utilized the hybrid approach, which avoided using the antecedents algorithm with its large overhead under certain conditions and substituting with the sequential algorithm which performed well only under those conditions.

3. I queued up the results from multiple samples and processed them in single batch. Batch processing can reduce the overhead per sample and takes advantage of the antecedent algorithm's efficiency in processing many falsified invariants.

6.2 Improving the antecedents algorithm

The first part of my design focuses on improving the antecedents algorithm. Daikon's implementation of the algorithm spend a large amount of time creating invariants over constant variables during suppression processing. Figure 6-1 shows the amount of time that constant creation takes as a percentage of total suppression processing time. The percentage varies from 16% to 66%.

Daikon creates these invariants, uses them in suppression processing, and then discards them. There are a large number of invariants due to the grammar of invariants and the number of constants. Daikon instantiates constant invariants over 168 unary and binary invariant templates and in the program points studied, the number of constant leaders (see equality set optimization in 2.3.2) range from 50 to 100. The no-filter column in Table 6-2 shows the large number of constant invariants that Daikon creates. Thus, reducing the number of constant invariants created has the potential to significantly reduce the overhead of the algorithm. In addition, a reduction in the number of constant invariants also reduces the number of cross products that need to be done by the antecedents algorithm.

	Constant creation	Total suppression processing	Constant creation %
bzip2	2.467 s	4.163 s	59.26%
svm-light	5.672 s	7.386 s	76.79%
flex	88.323 s	447.485 s	19.74%

Figure 6-1: Constant invariants creation time compared to total suppression processing time on the three sample programs. Time is measured in seconds. The third column shows the constant creation time as a percentage of total suppression processing time. Daikon spends the rest of suppression time primarily on performing the cross product of invariants.

I considered all of the invariants that Daikon creates over constants and whether or not all of these invariants were necessary. I considered the types of the invariants, the types of the variables, and the number of variables and which ones are relevant to the suppression optimization. The implementation already optimizes for the number of variables since Daikon only creates unary and binary invariants over constants because ternary invariants never appear in the antecedent of any suppression. In the following sections I will explain how I discovered the relevancy of these factors and my response in making the improvements.

6.2.1 Reducing the number of invariants using invariant type information

I examined the relevant invariant types to the suppression optimization. In constant invariant creation, the only relevant invariant types are those that appear as antecedent term templates in a suppression template. Daikon, however, creates all unary and binary invariants regardless of their invariant type. This is inefficient because Daikon creates these invariants for invariant types that do not appear in any antecedent, never uses them, and then discards them at the end of suppression processing.

For example, the invariant type of the invariant template $A * \alpha + B * \beta = C$ (for constants A, B, and C) never appears as an antecedent term template. Thus, Daikon can safely ignore invariants of this type. Daikon can generate the list of invariant templates relevant to suppression dynamically because it has a map from

	# of distinct constants	# of invariants (no filter)	# of invariants (filter)
bzip2	54	2933	1084
svm-light	47	2447	865
flex	109	14438	9255

Figure 6-2: The number of invariants created over constant variables by Daikon before and after applying the invariant type filter. The number of *distinct* constants is the number of leaders in the equality sets over constant variables. The number of constants selected is the lowest number of distinct constant variables observed at the program point.

	Constant creation (no filter)	Constant creation (filter)	Improvement %
bzip2	2.467 s	0.968 s	61.76%
svm-light	5.672 s	2.033 s	64.15%
flex	88.323 s	44.313 s	49.82%

Figure 6-3: Comparison of constant invariants creation time before and after applying the invariant type filter in the three programs. Time is measured in seconds.

each invariant type to the suppression set templates that contain an antecedent term template with that invariant type. I enhanced Daikon to generate a list of suppression related invariant templates from the map and use this list when creating the invariants over constant variables.

Before my enhancement, Daikon creates invariants from a list of 168 invariant templates. After applying the invariant type filter, Daikon reduced the list to 66 templates, a 60% improvement in the number of templates. Table 6-2 shows the number of invariants created by Daikon before and after Daikon applies the filter at one program point in the test programs (see section 7.2 for a detailed description of the test programs). I studied program points where Daikon spends a large amount of time on processing suppressions. Table 6-3 shows the improvements in the constant invariant creation time after applying the invariant type filter to the invariant templates. Both the bzip2 and svm-light programs show slightly more than a 60% improvement in the number of invariants created by Daikon and the time spent by Daikon in creating these constant invariants . Flex does not show as a large improvement, which may be because of the large number of constants at the program point.

6.2.2 Reducing the number of invariants using variable type information

I then examined each of the possible variable types to find the ones relevant to suppressions. I discovered that the `hashcode` type (see section 2.1 is a valid type for invariants templates in the antecedents of suppression templates, but is never a valid type for invariants in the consequent. This is an artifact of the fact that Daikon represents both integers and hashcodes as integers. Daikon uses the same invariant type (for example, $\alpha \neq 0$) for both types. Since each concrete variable that appears in an antecedent also appears somewhere in the consequent, and no consequent contains a hashcode variable, then invariants over hashcodes cannot be antecedents. For example, consider the suppression:

$$(\alpha = 0) \wedge (\beta \neq 0) \Longrightarrow (\alpha \ \% \ \beta == 0)$$

The antecedent term template ($\beta \neq 0$) is valid over hashcodes, but the invariant template of the consequent is not. Since the invariant template of the consequent is not valid over hashcodes, it will never be created by Daikon and the suppression is not relevant to hashcodes.

Even though only 3 invariant types in Daikon's grammar are valid over hashcodes (α one of $\{\}$, $\alpha \neq 0$, and $\alpha = \beta$), these types appear in some of the most popular antecedent terms. Table 6-4 shows the breakdown of the suppressions that include each of the three hashcode invariant types. The hashcode invariants are very common antecedent term templates and account for more than half the total number of suppression templates. In addition, there are usually a fair number of pointer variables in programs so these invariants are very common.

I updated Daikon to ignore variables of hashcode type when creating invariants over constants. In some programs, hashcode variables produced many invariants (see figure 6-5). Figure 6-6 shows the improvement to the constant invariant creation after applying the hashcode filter. The improvement in time does correlate with the

	Number of suppressions (recursion applied)
α one of {}	0
$(\alpha \neq 0)$	578
$\alpha = \beta$	1034
Combined hashcode	1296
Total	1931

Figure 6-4: Tally of the number of suppression templates that involve the 3 hashcode invariants. The "combined" number of suppressions is the number of distinct suppressions that contain the 3 invariant types (i.e. suppressions that contain two or more of the hashcode types only count once). The total is the number of the suppression templates (recursion applied) that Daikon has.

	# of hashcode variables	# of invariants (no filter)	# of invariants (filter)
bzip2	6	25	0
svm-light	15	120	0
flex	54	1024	0

Figure 6-5: The number of invariants created over constant variables by Daikon before and after applying the hashcode type filter. The number of hashcode variables is the number of constant variables that has the type hashcode. I observed and averaged the number of hashcode variables at three program points in each program.

number of invariants ignored, with flex and svm-light showing more improvement in the constant invariant creation times as they have many more hashcode variables.

6.3 Hybrid approach

The antecedents and sequential algorithms perform best under different conditions. The hybrid algorithm takes advantage of this by deciding dynamically which algorithm to use depending on the context. In comparing the two algorithms (section 5.5), I mentioned that the antecedents algorithm has a large fixed cost because Daikon must create the invariants over constant variables when processing each sample and that the sequential algorithm does redundant work when there are many falsified invariants.

Experiments that compare the processing time of the two algorithms with different number of falsified invariants appear to support this observation. Figure 6-7 shows the experimental results for the program svm-light. Because the shapes of the graphs for the other two programs (flex and bzip2) are similar, throughout the section, I will

	Constant creation (no filter)	Constant creation (filter)	Improvement %
bzip2	2.467 s	2.277 s	7.70%
svm-light	5.672 s	1.542 s	72.81%
flex	88.323 s	56.305 s	63.75%

Figure 6-6: Comparison of constant invariants creation time before and after applying the hashcode type filter in the three programs. Time is measured in seconds.

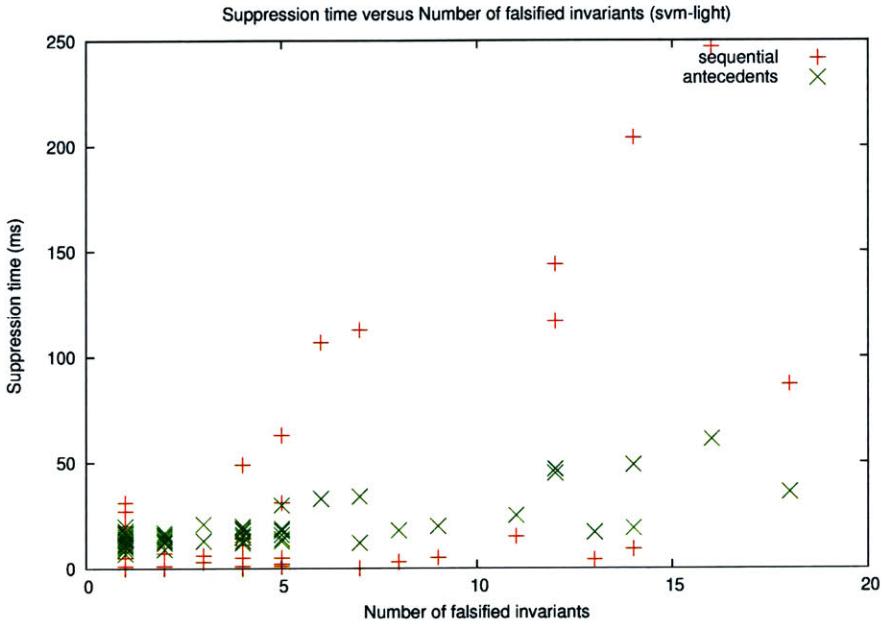

Figure 6-7: Comparison of the suppression processing time of the antecedents and sequential algorithms with respect to different number of falsified invariants on a single program point in the svm-light program. Time is measured in milliseconds.

use svm-light as the example program for simplicity.

Based on the graph, in general, for less than 5 falsified invariants, the sequential algorithm performs better than the antecedents algorithm. As the number of falsified invariants grows, the processing time of the sequential algorithm grows much faster than the antecedents algorithms. For example, the graph shows that in a sample that falsified 16 invariants, the sequential algorithm took 250 milliseconds while the antecedents algorithm took about 60 milliseconds.

To use a context sensitive approach, I need to figure out exactly when to use each algorithm. Based on the observations that I made, I decided to use a simple threshold to decide when to use each algorithm. After reaching this decision, two

questions arise. Which feature should the hybrid approach split on and what is the ideal threshold?

6.3.1 Exploring features

I evaluated three candidate features experimentally for the hybrid approach: the number of falsified invariants in the sample, the number of suppression templates relevant to the falsified invariants, and the total number of antecedent terms templates in the relevant suppression templates. The number of suppression templates to be processed is the total number of suppression templates triggered by the falsified invariants. I obtained this number by looking at the map from invariant type to suppression set templates and summing up the associated suppression templates for each falsified invariant. The number of antecedent term templates is the total number of terms in all of the suppression templates triggered by the falsified invariants.

I evaluated the three candidate features by comparing the time spent in suppression processing by the two algorithms for different values of each feature. Figures 6-7, 6-8, and 6-9 show the performance of the antecedents and sequential algorithm with respect to the number of falsified invariants, suppression templates, and antecedent term templates, respectively.

Based on the graphs, the best thresholds are 5, 2000, and 7500 for the invariants, suppression templates and antecedents term templates features respectively. However, the suppression templates and antecedents term templates split the data much better than the falsified invariants feature. At 5 falsified invariants, there is no clear division in the data although there is some indication that the sequential algorithm performs better for smaller numbers while the antecedents algorithms is better for a larger number. In the other two features, the division is much clearer.

Figure 6-10 shows a close up of the lower range of figure 6-8, from 0 to 2000. This range is exactly the range in which the sequential algorithm should perform better than the antecedents algorithm. In the figure, except for about 5 samples, the sequential algorithm does perform better than the antecedents algorithm. In the figure 6-8, beyond 2000 suppression templates, the antecedents algorithm clearly

performs better than the sequential algorithm.

I investigated why the falsified invariants feature did not split the data well by looking at samples where a small number of falsified invariants unexpectedly prompted the sequential algorithm to perform worse than the antecedents algorithm. I found that in these anomalies, the small number of falsified invariants are among the most common antecedent terms in the suppression templates. These invariants trigger many relevant suppressions. The result suggests that the number of suppression templates, rather than the number of falsified invariants, is a more accurate measure of suppression processing time. The number of falsified invariants is less reliable because because one falsified invariant may trigger one suppression template or one hundred suppression templates.

I originally conjectured that the complexity of a suppression template may be a factor because suppression templates with more antecedent term templates may takes longer to process. However, the shape of the antecedent term templates graph follows that of the number of suppression templates, suggesting that the total number of antecedent term templates is closely correlated with the total number of suppression templates.

6.3.2 Selecting the threshold

Since the performance of the suppression templatess and antecedent term templates features was similar, I picked the suppression templates feature because the feature is faster to calculate at runtime. To determine the splitting threshold, I examined the experimental graphs to find search range for the threshold. For example, in the number of suppression templates feature, I ran experiments for thresholds ranging from 1000 to 400 at increments of 500. I ran Daikon on all 3 three programs using the selected range and compared the total result by summing up the times from the three programs at each threshold. I chose the threshold that yielded the lowest total time.

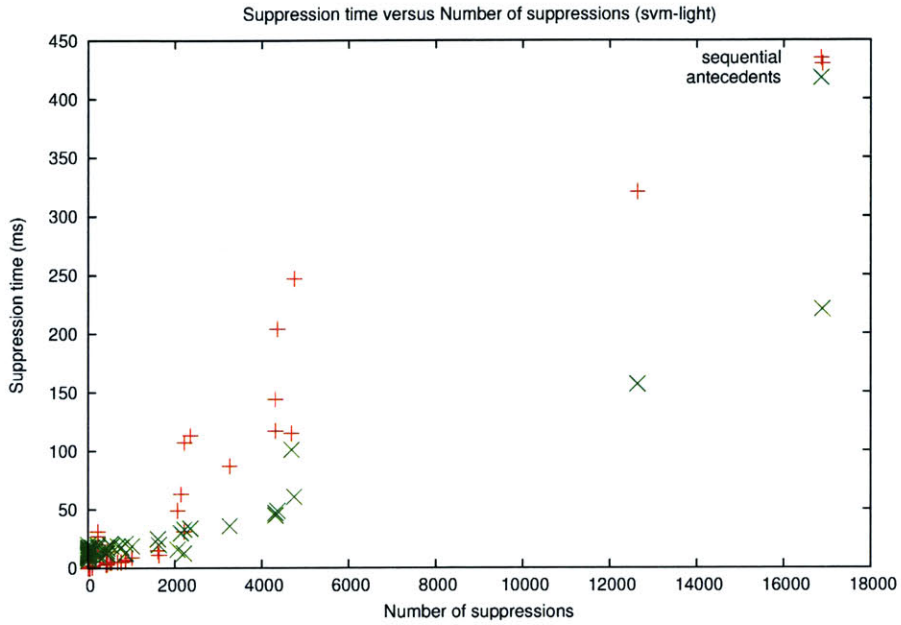

Figure 6-8: Comparison of the suppression processing time of the antecedents and sequential algorithms with respect to different number of suppression templates on the program svm-light (single program point). Time is measured in milliseconds.

6.4 Batch algorithm

I explored new approaches for improving the performance of the suppression processing algorithm. In my observations, I found that compared to the sequential algorithm, the antecedents algorithm works relatively well for a large number of falsified invariants, but triggers a large number of suppressions. Although there is a large fixed cost of creating invariants over constant variables, the cost pales next to the cost of running through the cross product with so many falsified invariants.

In my experiments, I looked at the processing time per falsified invariant used by the antecedents algorithm. I found that the time drops rapidly as the number of falsified invariants increases (see figure 6-11). The graph shows that by processing more falsified invariants together, the average cost goes down because the fixed overhead of the algorithm is spread over more falsified invariants. The idea is to batch those samples that falsified one or two invariants and wait until a certain threshold to process them. In addition, since many of the invariants in the cross product will

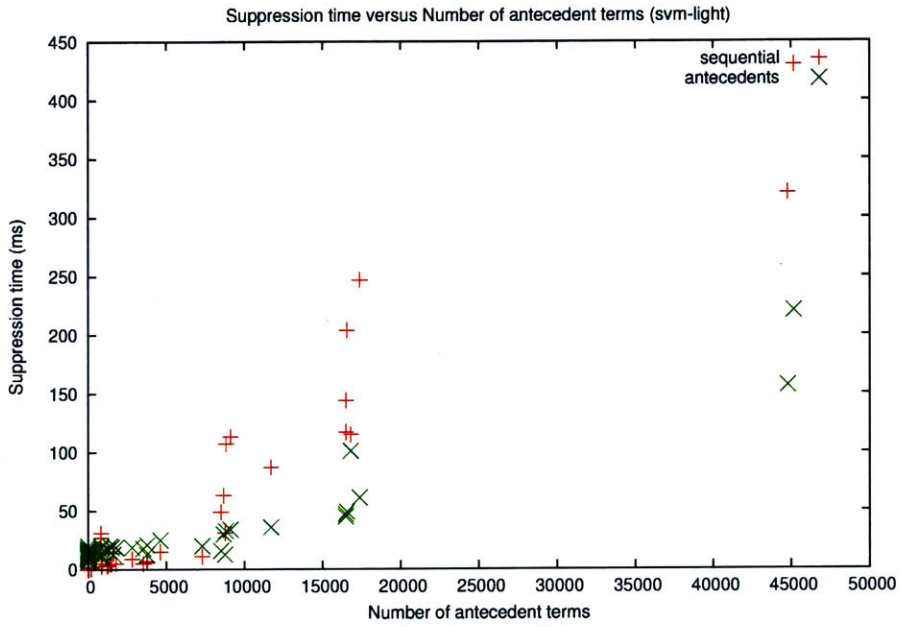

Figure 6-9: Comparison of the suppression processing time of the antecedents and sequential algorithms with respect to different number of antecedent terms on a single program point in the svm-light program. Time is measured in milliseconds.

be the same between samples (not many invariants are falsified by each sample), the algorithm can also spread the overhead of the cross product over more samples.

Similar to the hybrid approach, I needed to pick the criteria for determining when a batch is ready for processing. In my design, I batched on the number of falsifying samples. The batch algorithm stores the samples that falsified invariants and does not remove falsified invariants. The algorithm keeps that samples that falsified invariants because it needs to apply these samples to unsuppressed consequents to determine whether the consequent is false see section 4.5.2). The algorithm does not need to keep samples that do not falsify any invariants: since the samples do not falsify the potential antecedent invariants, they can not falsify any consequents.

Once the number of falsifying samples reaches the threshold for a particular program point, then Daikon uses the antecedents algorithm to process the suppressions and applies the stored samples to the newly created consequents. After the antecedents algorithm has processed the suppressions, Daikon discards the stored samples and all falsified invariants. Similar to the hybrid algorithm experiments, I se-

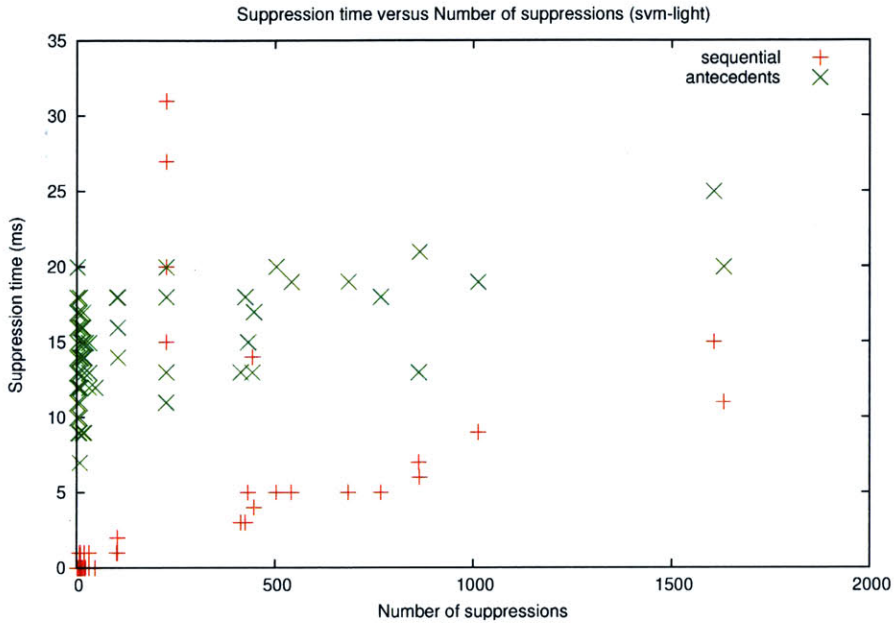

Figure 6-10: Comparison of the suppression processing time of the antecedents and sequential algorithms with respect to different number of suppression templates on a single program point in the svm-light program. This graph zooms in on the lower ranges of figure 6-8. Time is measured in milliseconds.

lected a potential range for the threshold and ran Daikon on all 3 programs, picking the threshold that yielded the lowest total Daikon running time.

Figure 6-12 shows that the number of runs of the antecedents algorithm decreased by 30-50% in the three test programs. However, the approach did not improve the running time of Daikon as dramatically as suggested in figure 6-11.

A possible explanation for this result is that Daikon stores redundant falsifying samples. For example, two samples may falsify the same invariant and Daikon stores both of these samples and applies them to the consequents. Take for example the suppression

$$x > y \Longrightarrow x \geq y$$

The samples $(x = 1, y = 2)$ and $(x = 2, y = 3)$ both falsify the antecedent invariant $x > y$ and will also falsify the consequent $x \geq y$. In this first case, the

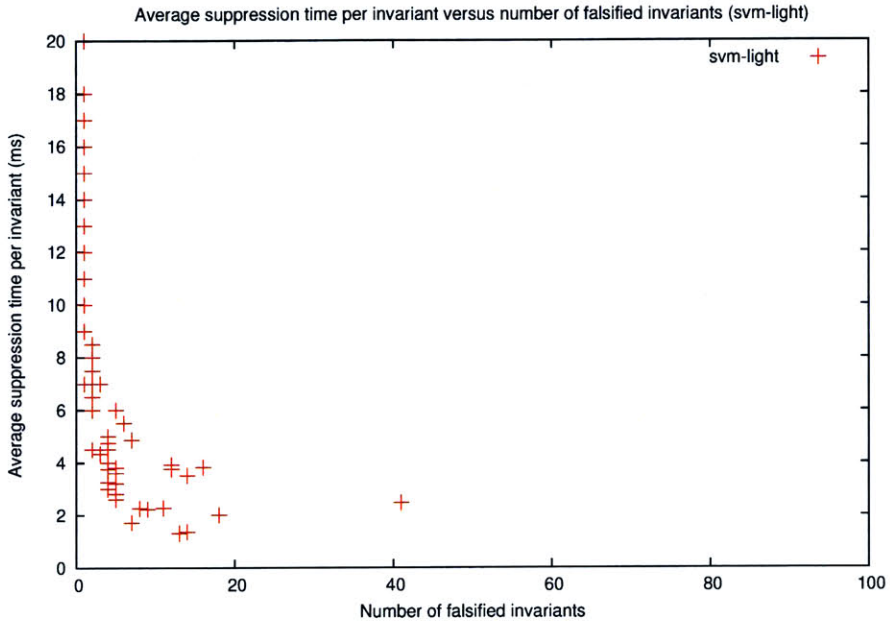

Figure 6-11: The average cost of processing a falsified invariant in the antecedents algorithm varying with the number of falsified invariants. Time is measured in milliseconds.

samples are redundant. However, in the second case, the samples $(x = 1, y = 2)$ and $(x = 2, y = 2)$ are not redundant because one of them will falsify the consequent $x \geq y$ but the other will not. It is difficult to distinguish between the two cases without creating the consequent and applying the sample.

In combination with the batching threshold, redundant samples is a problem. Since a sufficient number of samples triggers the batching algorithm, redundant samples inadvertently contribute to the accumulating threshold count, and may trigger the batching algorithm earlier than the predicted goal. Ideally each falsifying sample stored falsifies at least 1 new invariant, so the number of falsified invariants at the time of processing the batch should be greater than or equal to the threshold value.

Figure 6-13 shows the number of "under threshold" runs in the batching algorithm. These runs account for 25 to 50% of the total runs of the antecedents algorithm. In these runs, there are less falsified invariants than falsifying samples, which means that some samples falsified the same invariant. Some of those samples must be redundant and the extra calls to the antecedents algorithm force Daikon to use the expensive

	Control (antecedents)	Batching
bzip2	87	41
svm-light	142	59
flex	1279	396

Figure 6-12: Improvement in the number of calls of the antecedents algorithm in the three programs using the batching algorithm.

	$<$ threshold	$\geq$ threshold
bzip2	15	41
svm-light	32	59
flex	107	396

Figure 6-13: Below the threshold means that the number of falsified invariants processed was less than the number of falsifying samples (threshold). Above the threshold means equal or above the number of samples.

constant invariant creation code more often than necessary and apply these redundant samples to the consequents.

Chapter 7

Observations and Evaluation

7.1 Data collection and observations

Since the research is performance driven, I collected most of the data by running Daikon on the test programs and using the system clock to profile the code of interest. I initially used the Java HPROF to profile Daikon, but found the tool to be inadequate for my needs. The tool does not report aggregate time over a particular method, and can not report the time for a few lines of code in a method. By manually using the system clock to profile, I have more control over the code that gets profiled and can easily aggregate the times that I need. In addition, by instrumenting Daikon directly, I can control the information reported.

To ensure accurate times, I ran experiments only when my experiments took up at least 90% of the CPU. Using these times, I could pinpoint the pieces of code that took a large proportion of the running time and respond accordingly in my implementation of the enhancements. Using this data, I gained better understanding of the antecedents and sequential algorithms. I described most of these observations in the design section, as they motivated the enhancements that were proposed. In addition, I found that the number of invariants created over constants in the antecedents algorithm and the large number of suppressions that need to be processed contributed to the large amount of time spent in suppression processing.

I used the data to confirm assumptions about the algorithms. This step was

	Equality sets	Suppression time	Daikon time
bzip2	31	3.203 s	12.544 s
svm-light	30	4.92 s	28.635 s
flex	65	416.169 s	956.225 s

Figure 7-1: Comparison of suppression processing time compared to total Daikon running time on the three test programs. Time is measured in seconds. The number of equality sets is the average number of equality sets at the program points and corresponds to the size of the program since it is the number of *distinct* variables in the program.

essential in discovering the hashcode type filter to the antecedents algorithm (section 6.2.2). In addition, I used the data to discover and investigate anomalies. The data was important in discovering which feature is most relevant in the hybrid algorithm (section 6.3).

7.2 Framework

To evaluate the enhancements, I measured the total Daikon running time on three test programs. I chose these programs because the suppression processing algorithm takes from 20 to 40% of the total Daikon running time in these programs (see figure 7-1). These programs are good markers for whether the performance enhancements show any improvement. The three programs are written in C: the flex lexical analyzer (part of the standard Linux distribution), bzip2 (a freely available data compressor), and svm-light (an implementation of support vector machines). In the experiments, I compared the running time of Daikon on the three programs before and after turning on the enhancements. I used a computer with 3.6GHz processing speed and 4GB of memory.

7.3 Results

Figure 7-2 shows that although all of the enhancements show an improvement over the control (original antecedents algorithm), the improved batching and hybrid algorithms using the improved antecedents algorithm have the best results. While the improved

	bzip2	svm-light	flex
control (antecedents)	14.209 s	30.154 s	996.698 s
improved antecedents	13.189 s	27.930 s	880.928 s
hybrid (3000 suppressions)	13.101 s	28.782 s	965.444 s
improved hybrid	12.865 s	27.294 s	880.695 s
batching (5 samples)	13.125 s	27.552 s	1061.066 s
improved batching	12.629 s	26.206 s	967.576 s
best algorithm	improved batching	improved batching	improved hybrid

Figure 7-2: Comparison of the different performance enhancements as measured by the total Daikon running time (in seconds) on the the three test programs. The hybrid algorithm splits on 3000 suppressions. The batching algorithm batches on 5 falsifying samples. The improved hybrid and batching algorithms use the improved antecedents algorithm.

batching algorithm does have a faster time than the improved hybrid on two of the programs, the difference is very small.

Overall, the improved hybrid performed the best on all 3 programs. Using the improved hybrid algorithm, in flex, the running time of Daikon improved by about 2 minutes, a 12% improvement. The improved hybrid algorithm provided only a small improvement over the improved antecedents algorithm in all three programs. The batching algorithm performed well on the bzip2 and svm-light programs, but poorly on flex, which could be due to differences in program sizes (number of distinct variables).

Chapter 8

Verifying Implementation Correctness via an Oracle

8.1 DaikonSimple: an overview

Verifying the implementation of a program is very difficult in programs that produce a large amount of output because checking the results by hand is not feasible. The problem is even harder when the program filters its output via optimizations to make the program run faster and make the results more useful for users. This problem applies to the optimizations implemented for Daikon. The optimizations described in section 2.3 and the new enhancements detailed in the design section both ignore redundant invariants during processing and in the output in order to improve performance and the readability of the results for the readers. This situation presents a tester with two difficulties: making sure that reported results are correct, and making sure that no desired results are missing.

In order to verify the implementation of both the optimizations and the enhancements made to Daikon, I address the two difficulties by using an oracle. The approach, illustrated in figure 8-1 uses an easily verifiable brute force algorithm as an oracle to generate the same output as the algorithm with complex optimizations. Running the brute force algorithm produces a set of unfiltered results. In addition, running the complex algorithm (with its optimizations) and then reversing the optimizations

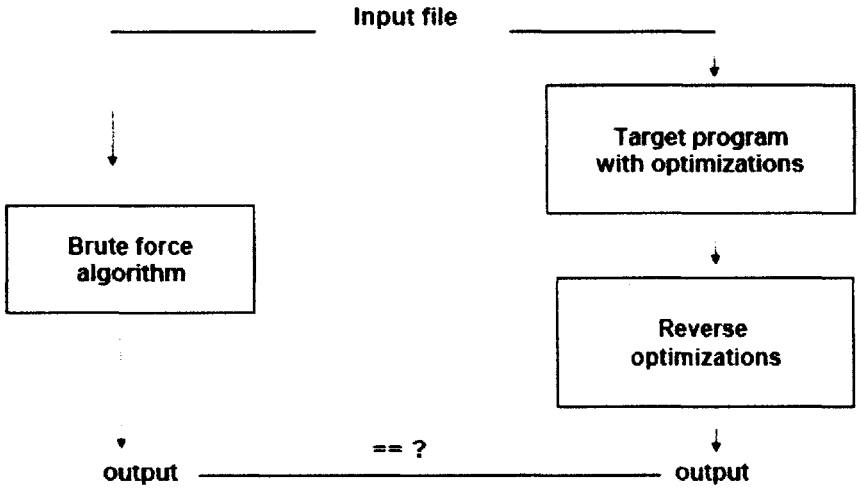

Figure 8-1: Visualization of the oracle approach to verification.

of the complex algorithm to recover the filtered output produces a complete set of results. Differences in the two outputs (brute force and complex with optimization reversal) indicate potential problems in the implementation.

The verification method should identify errors at lower cost than developing exhaustive test cases. Using such a verification method, I can be more confident about the overall correctness of the code. This approach is complementary to regression tests because I can use the slow oracle to ensure that the goal output of the regression tests is correct and then use the faster regression tests to maintain the correctness of the code.

8.2 Brute force approach: simple algorithm

The verification approach first obtains the complete results via the brute force algorithm. I implemented the simple algorithm described in section 2.2 and named this tool DaikonSimple. To focus on implementing the actual invariant detection algorithm, I reused classes from Daikon. The code shared by the implementations of the two algorithms (Daikon and DaikonSimple) are the parts of Daikon that are not the object of this testing approach. The approach tests the complex optimizations and not the the classes relating to the program points, invariants, variables, samples,

reading input and formatting output. The code not tested by this approach is more straightforward to test and largely covered by the Daikon unit tests. For example, the actual invariant classes have their own unit tests.

8.3 Reverse optimizations

The verification approach then from the target program, Daikon, procures the same complete results as the oracle. In order to obtain complete results, I implemented methods for reversing all of the four optimizations detailed in section 2.3. In several cases, Daikon already had a method that performed that task needed, but Daikon used these methods in a different context. I recycled these methods to save time. Daikon uses each of these methods at the end of processing all of the samples to recover the filtered invariants and obtain a complete list of true invariants.

To reverse the dynamic constants optimization, Daikon needs to recover the invariants over constant variables. Daikon can obtain these invariants by explicitly creating them after processing all of the samples. These invariants are almost exactly the ones that are created during suppression processing by the antecedents algorithm. The antecedents algorithm only creates the unary and binary invariants over constant variables, but the optimization reversal method creates all ternary invariants over constant variables in addition to the unary and binary ones.

To reverse the equality sets optimization, Daikon must find the invariants over the non-leaders of each equality set. Daikon can get these invariants by copying the invariants of the leaders to the non-leaders of each equality set. These invariants are exactly the ones that are created by Daikon when a variable breaks away from an equality set (see section 2.3.2), so I was able to reuse the code that performed the copying.

To reverse the variable point hierarchy optimization, Daikon must recover the invariants at the lower points. Daikon can obtain these invariants by keeping the invariants at the lower points even when they are propagated by Daikon to the higher points. Keeping the invariants at the lower points is already the default option in

Daikon.

To reverse the suppressions optimization, Daikon can recover the implied invariants by creating all of the consequents that are implied by the true invariants. Daikon already has the option of using the cross product of the antecedents algorithm to find and create all of the suppressed consequents.

8.4 Alternative approaches

Another way to verify the optimizations would be to filter the output of the brute force algorithm using the same optimizations as the optimized algorithm. This method does not yield the same effect as the proposed approach. Although this alternative approach eliminates the need to reverse the optimizations, the approach does not test the implementation of these optimizations. A bug in the optimizations would eliminate the same output from both algorithms. Thus the approach does not address the difficulty of verifying that no desired results are missing.

Another approach would be to simply test all of the reported invariants against all of the samples to ensure that they are really true. Again, this approach addresses the first problem (verifying that reported results are true) but would miss the second requirement: no desired output is missing.

In addition, the idea of turning off the optimizations in Daikon and comparing the results to the output of the brute force algorithm does not achieve the desired effect. Turning off the optimizations defeats the purpose of the approach: testing the implementation of the optimizations.

8.5 Evaluation

8.5.1 Framework

To evaluate the effectiveness of DaikonSimple, I ran Daikon and DaikonSimple on the regression tests for Daikon. These programs are written in C and Java, mostly data structure classes, ranging from stack and queue implementations to map finding

equality sets	dynamic constants	variable hierarchy	suppression	other
0	0	2	4	9

Figure 8-2: Number of optimization related and other bugs found

classes. I compared the results of Daikon and DaikonSimple, and investigated each difference in the results.

8.5.2 Results

Figure 8-2 shows the types of bugs found by the verification approach. The approach was effective in finding bugs in both the implementation of the optimizations and surprisingly also in the invariant processing code that is shared by both Daikon and DaikonSimple. The bugs found in the suppression optimization code were all caused by corner cases missing in the predefined suppression templates. For example, a concrete suppression with program variables x, y, and z states:

$$(x == y) \wedge (y < z) \Longrightarrow x = y \ \% \ z$$

However, the sample (1, 1, 0) breaks the suppression because modulo 0 is undefined. In addition, the sample (-1, -1, 2) breaks the suppression because in Java, -1 mod 2 yields 1. Thus, the concrete suppression should actually be:

$$(x == y) \wedge (y < z) \wedge (z \neq 0) \wedge (x \geq 0) \Longrightarrow x = y \ \% \ z.$$

Surprisingly, the oracle approach also found bugs in code which is shared by Daikon and DaikonSimple. Daikon and DaikonSimple differ in the algorithm used to find the true invariants; the actual invariant classes are shared by both programs. The bugs ranged from math errors to inconsistent labeling of variables. For example, the oracle found a math bug in the code of a ternary invariant that tries to fit a plane over the three variables, i.e. $A*x+B*y+C*z = D$, where A, B, C, D are constants. For positive numbers, because

$$(x < y) \Longrightarrow (x^2 < y^2)$$

is true, the developers of the invariant code wrongly assumed that:

$$(x + y + z < u + v + w) \Longrightarrow (x^2 + y^2 + z^2 < u^2 + v^2 + w^2)$$

is true. Since exponentiation is not distributive over addition, the assumption fails in the sample where $x = 1$, $y = 17$, $z = 18$, $u = 18$, $v = \sqrt{75}$, and $w = \sqrt{147}$.

Chapter 9

Related Work

9.1 Theorem provers

Theorem provers work on logical formulas and have no limitations on the number of variables used [11]. Using a set of logical statements as the antecedent and an arbitrary consequent, the theorem prover applies a set of logic rules to determine whether the implication is true.

A suppression in the form of $(a < b \wedge b \neq 0) \Longrightarrow a/b = 0$ is only one instance of a logic statement that a theorem prover can prove, i.e. suppressions are a low-level enumeration of the statements that a theorem prover can prove. The suppression takes the form of inputs to the theorem prover, namely a set of logical statements that make up the antecedent and a consequent. However, a theorem prover can not replace the suppression processing algorithm in Daikon. The antecedents algorithm finds the invariants that needs to be created because they are not suppressed anymore, but a theorem prover can only say yes or no to an actual implication. In order to use the theorem prover, both the antecedent and consequent must be present, but the consequent is not present precisely to save memory.

9.2 Verification via bounded exhaustive testing

Previously, researchers have used formal specifications for the target program

1. to automatically generate a bounded exhaustive test suite as test cases for the target program and

2. to derive an oracle designed to verify the output of the target program on the generated test cases [18].

Rather than using the specifications to generate test inputs, I use the current regression tests as test cases utilized by the verification method. In addition, I use the brute force algorithm as an oracle rather than the formal specification. The approach offers the advantage of turning pre-existing regression tests into valuable test cases for the program. The approach differs from the method in [18] in that, rather than using the specifications for the program to generate a bounded exhaustive test suite and an oracle, it tests the behavior of the target program by using the current regression tests and a brute force algorithm.

The other approach is difficult to apply to Daikon because the optimizations are too complicated for reasonable specifications. Even if the optimizations could be specified, Daikon as a whole could not. The oracle approach is more applicable to complex code that is part of a larger whole. In addition, a brute force algorithm is often easy to implement.

Chapter 10

Limitations, Future Work, Contributions

10.1 Limitations

There are several limitations to my work. In evaluating the hybrid and batching algorithms, the hardest part was picking a threshold for the two algorithms for splitting and batching respectively. There is no ideal number since the ideal number will vary from program to program. I picked a number that yielded the best results for the three programs collectively, but it is difficult to say if the three programs are representative. Lastly, DaikonSimple was an effective way to find bugs in Daikon, but since it only implements the simple algorithm, it runs into scalability problems. Consequently, there is a size limit on the input files for this verification approach. In addition, a human must trace the differences in the two outputs, which was a tedious task at times. This difficulty is generally associated system test suites because while system tests provide the benefit of testing the entire code base, it is much harder to pinpoint the bug to a particular method in the code.

10.2 Future work

In the future, more time can be spent exploring other ways to improve suppression processing algorithm. For example, currently, I batch on a set number of falsifying samples in the batching algorithm. As noted in design section 6.4, it is difficult to identify redundant samples so batching on the number of falsifying samples makes the problem worse by triggering the antecedents algorithm too early. Similar to the features in the hybrid algorithm, the batch algorithm may benefit from batching on the number of accumulated falsified invariants or relevant suppressions. In addition, we may be able to further reduce constant invariant creation time in the antecedents algorithm by using an idea from the sequential algorithm: only create the constant invariants for the invariant types in the suppressions triggered by the falsified invariants.

10.3 Contributions

In this research, I investigated and gained better understanding about two algorithms that optimize dynamic invariant detection. Using this knowledge, I designed and implemented enhancements to improve the runtime of Daikon. In addition, I built an oracle to verify the implementation of all optimizations in Daikon. I improved total Daikon runtime by 10% on the test programs.

In the process of the research, I learned the importance of verifying assumptions via experiments and investigating rather than ignoring anomalies. The two were important in gaining insight into the algorithms, and discovering potential optimizations. When evaluating the feasibility of new algorithms (e.g. the batch algorithm), I should think more carefully about the costs, not just the benefits, as the cost could outweigh the benefits. In addition, the hybrid algorithm is an interesting solution to the performance problem. I used my own feature selection methods to decide on the threshold of the hybrid algorithm. Lastly, the oracle approach for verification is applicable to other algorithms.

Bibliography

[1] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE2004*, pages 480–490, May 2004.

[2] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 233–243, Portland, ME, USA, July 18–20, 2006.

[3] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[4] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[5] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *SPIN2003*, pages 121–135, May 2003.

[6] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE2002*, pages 291–301, May 2002.

[7] K. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE 2003, Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, Portland, OR, USA, May 3–10, 2003.

[8] Nancy G. Leveson, Stephen S. Cha, John C. Knight, and Timothy J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, April 1990.

[9] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI2003*, pages 141–154, June 2003.

[10] Leonardo Mariani and Mauro Pezzè. A technique for verifying component-based software. In *TACOS2004*, pages 17–30, March 2004.

[11] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kırlı, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004.

[12] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[13] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Softwar Engineering (FSE 2002)*, pages 11–20, Charleston, SC, USA, November 18–22, 2002.

[14] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 23–25, 2007.

[15] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th*

Symposium on the Foundations of Software Engineering (FSE 2004), pages 23–32, Newport Beach, CA, USA, November 2–4 2004.

[16] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *AADEBUG2003*, pages 273–276, September 2003.

[17] Orna Raz, Philip Koopman, and Mary Shaw. Semantic anomaly detection in online data sources. In *ICSE2002*, pages 302–312, May 2002.

[18] Kevin Sullivan, Jinlin Yang, David Coppit, Safraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *ISSTA 2004, Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 133–141, Boston, Massacusetts, July 11–14, 2004.

[19] Tao Xie and David Notkin. Tool-assisted unit test selection based on operational violations. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 40–48, Montreal, Canada, October 8–10, 2003.

[20] Tao Xie and David Notkin. Checking inside the black box: Regression testing based on value spectra differences. In *ICSM2004*, pages 28–37, September 2004.