AN ABSTRACT OF THE DISSERTATION OF

Chris Chambers for the degree of Doctor of Philosophy in Computer Science presented on July 10, 2014.

Title: Helping End-User Programmers Find and Fix Performance Problems in Visual Code

Abstract approved: _____

Christopher P. Scaffidi

End-user programmers often struggle to create programs that run quickly and effectively, which can be a major deterrent in completing their tasks as desired. Current research has primarily focused on catching user mistakes, such as errors or misused formulas. However, end users deal with issues other than just correctness. In particular, there are very few tools and very little research aimed at helping end-user programmers to find and fix performance issues. This thesis details three specific methods: detecting code smells, combining static code smell detection with profiling information, and the semi-automatic or tool-guided removal of code smells. These methods have been prototyped to interface with the Labview IDE with the support of National Instruments. These methods have been evaluated through several user studies to ensure that they are effective and helpful.

Helping End-User Programmers Find and Fix Performance Problems in Visual Code

by

Chris Chambers

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented July 10, 2014
Commencement June 2015

Doctor of Philosophy dissertation of Chris Chambers presented on July 10, 2014.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Chris Chambers, Author

ACKNOWLEDGEMENTS

I would like to thank my advisor Chris Scaffidi for all of the assistance and support he has given me, and for the example that he has set. It was often intimidating, but it was always motivating. I would certainly not have gotten here without his guidance.

I would also like to thank Andrew Dove of National Instruments for his invaluable guidance and help throughout my research. His desire to support research in all areas lead to this project and opened the door for more opportunities than I imagined.

I would like to thank my committee, (Carlos Jensen, Alex Groce, Glencora Borradaile, and Margaret Burnett) for their guidance and advice during this process. I would also like to thank all faculty members at Oregon State for the work that they do everyday. Being a professor and a researcher can often be a thankless job, but every class that I took and professor that I engaged with helped to shape the person and researcher that I am today. In particular I would like to thank Dr. Cull for providing coffee hours and engaging discussions, and Dr. Burnett for always making me think about how research should be done.

I need to of course, thank all of my friends for making the journey during grad school fun and exciting. I would not have reached this point as well adjusted as I am without having them to interact with, brew beer, discuss fantasy plots, play board games, and accuse of being spies (while of course being innocent myself). Eric for being the brother that I never knew I needed, Chris for his calm, measure approach to everything, Andrew for his culinary inventiveness, Will for his technical inventiveness, and especially Katie who has been, and will continue to be, the best friend that I have

ever had. Finally, I would like to thank the KEC Krew basketball team for putting up with my mediocre play and mostly competent managing.

And of course my family. I would not be where I am today without all of the love and support from my parents. They have always supported me in whatever I have done which was invaluable in helping me pursue my dreams.

Finally, thank you to National Instruments for the grants that funded this work and for the opportunities and belief in my research.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

This thesis is dedicated to the memories of my grandfathers, Arthur John Chambers and Newcomb Lee Cleveland.

Chapter 1: Introduction

It is estimated that there are nearly 80 million end-user programmers in the U.S. alone [83]. These individuals create programs for work or for hobby and use a variety of different languages and tools. One aspect of programming that often gets ignored until it becomes a problem by end-user programmers, and professional developers, is performance. Often, performance bugs are handled with a "Fix-it-later" approach [53][87], which can exacerbate the problems and make them more difficult to find and fix.

To meet this challenge, the primary contribution of this thesis is *Smell-Driven Performance Analysis* (SDPA) which employs the idea of performance smells to detect potential performance problems in dataflow code, such as LabVIEW or Yahoo! pipes (both common end-user environments). SDPA can detect and notify users of smells as well as provide users the ability to easily and efficiently apply transformations that can fix the issues. An evaluation showed the transformations provided by SDPA nearly matched the performance improvements of expert optimizations, without the wait time of forums or the expense of professional support. Moreover, a user study showed that SDPA helped end-user programmers become over 3 times more efficient at debugging performance problems compared to the existing techniques.

1.1 Motivation

Performance errors have been shown to be among the trickiest types of errors to debug. In particular, research [101] has shown that performance bugs take on average 32% more time to debug than non-performance bugs. In addition, these problems are also more likely to cause frustration during debugging [102].

It is perhaps because of this that many professional programmers employ the "Fix-it-later" approach when it comes to performance bugs [87]. In short, they tend to ignore these defects as they are hard to find, but this decision can make the entire development process take longer. Unfortunately, studies have shown that the cost of fixing performance problems increases the later on in the process they are addressed [7][103]. Being forced to fix a deferred performance error is 21 to 78 times more expensive than fixing it during design or when it is introduced. If the bug is not found until the program is deployed, the costs skyrocket to 29 to 1500 times more expensive [103]. This indicates that it is vital to identify these problems early in the programming process, but the tools available are often lacking, particularly for end-user programmers.

End-user programmers, unlike professional software engineers, are people who create programs that they plan to use for an immediate purpose [83], and it is estimated that there are nearly 80 million in the U.S. alone. While professional software engineers have long known that the performance of a program can have a powerful effect on its value, many end-users spend little time thinking about long term quality attributes, such as performance [53]. However, that does not mean that performance is

unimportant to end-users. For example, empirical studies have revealed that scientists value performance so much that they take time to learn textual general-purpose languages such as C++ or Fortran for high-performance computing, in addition to domain-specific languages designed for high usability [12][46][82].

LabVIEW is one example of a programming language that has high usability but sometimes leads to programs with inadequate performance. National Instruments, the maker of LabVIEW, claims that LabVIEW is the "most widely used development environment for instrument connectivity and [hardware] test application," particularly among engineers and scientists [49]. An independent survey of LabVIEW users investigated the reasons for this high level of adoption, and found that users appreciated LabVIEW primarily due to its visual dataflow language and secondarily due to its support for code reuse [95]. One respondent summarized, "The development time for LabVIEW is less than half that for C," while another claimed to be "3X more productive than programming in C". Yet at the same time, survey respondents sometimes found LabVIEW performance to be inadequate. In particular, they mentioned that LabVIEW has no way to optimize usage of registers, memory, disk, and CPU. The researchers conducting the survey noted, "LabVIEW solves these problems by allowing the programmer to call code written in other languages". In other words, the "solution" to the problem is to step outside LabVIEW.

These challenges are not unique to LabVIEW. Nardi lists HP VEE and Prograph as two other canonical examples of visual dataflow languages [68], where data input/output nodes are connected to one another via computation nodes and virtual dataflow "wires". Scientists and engineers using these languages (e.g., [48][86]) also

encountered performance problems. To solve such problems such as these, scientists would often have to rewrite parts of the application in a completely different language, such as C, and link that code to the VEE application [48].

At times these problems arise because end-users lack the knowledge necessary to find or fix the problems that they encounter. In one example, a LabVIEW user contacted National Instruments for help with a poorly performing program. This program would at first run fine, but after 10-15 minutes of execution, it would slow down and eventually the computer would crash. To try and fix the problem, the user kept increasing the virtual memory size, until this solution was no longer viable, when he contemplated buying a completely new computer. After a brief code review with an AE Specialist at National Instruments, the program was found to be continually creating duplicates of a 650 MB array. Each duplicate caused the memory usage of the program to increase until the computer crashed. The user was unaware of this behavior of arrays and thought that the same copy of the array was used each time.

Much of the research oriented towards end-user programmers has thus far focused on creating tools that aid in detecting functional errors [13][79][10][72], improving maintainability [14][33], and aiding reusability [25]. The research in maintainability and reusability, both quality attributes, shows that tools can be successful at getting end-users to think about the quality of their code. However, current research has not focused on code performance despite studies showing that end-users, particularly scientists [46], need well-performing code. There are many systems where when something happens, or how fast code can execute is of utmost importance. For example, code written by a lab technician might need to read a signal every 100 milliseconds.

While the code might be correct, it could be written inefficiently and execute every 250 milliseconds, thus missing the desired read time and limiting the usefulness of resulting data. End-users currently have very few options to debug such performance issues.

Research has shown that for professional programmers, collaboration is often the key to solving performance problems. One study [102] showed that successfully-debugged performance problems were more often the result of collaboration with other developers. In this study 47% of all performance bugs required collaboration before they could be fixed compared to only 25% of non-performance bugs. End-users programmers who do not have access to other developers must seek help in other ways.

One common method is to pay for customer support services. While this can be helpful, it is also expensive. The Standard Service Program for LabVIEW, which includes ongoing technical support and upgrades, costs several hundred dollars per year per user. [43]. This often means that end-users have to decide between paying several hundred dollars a year get access to professional help, or saving the money and trying to survive without it. Individual users who do not purchase the product, such as those in academia with a site license, do not have the option to pay for customer support and are left to seek other resources.

Paying for this support does not guarantee satisfaction. While the metrics determining an effective call center are varied, studies have shown that understaffing, abandoned calls, and long call waiting times are the key issues that affect customer satisfaction [78][57]. When users have no choice but to call for support, this increases

the waiting time for everyone and potentially makes more users frustrated with the service. Providing tools that aid users in debugging performance programs could reduce the number of calls. This in turn may reduce the stress on the support personnel and allow for better service to all customers.

For those who choose to forgo paying for professional support, one common resource is to post a question on a forum to try and get a quick and easy solution. Unfortunately, this too has problems. A quick survey of the National Instruments LabVIEW forum found that only 7.9% of 1290 randomly selected forum posts actually had a correct solution provided by other forum users. Even when a solution was given, the average wait time was longer than 21 hours. These results indicate that there are a large number of people who are not getting the help that they need. This could create frustration among users and adversely affect their productivity and the performance of the programs they create.

In short, end-user programmers need help with finding and fixing performance problems. The goal of this dissertation research is to meet this need.

1.2 Approach

The overall thesis of this research is that performance problems often appear in dataflow code structures that tools can semi-automatically detect and correct, thereby helping end-users find and fix performance problems in visual dataflow languages. Tools and methods were developed to find patterns in the code that are known to cause performance problems. Once these patterns have been detected, transforma-

tions and suggestions are provided to users to help them fix the detected issues.

Chapter 3 discusses the general approach in detail, as well as nine specific claims about the approach's effectiveness, usefulness, and generalizability. Chapter 4 details Smell-Driven Performance Analysis (SDPA) which develops the idea of performance smells, gathered from experts, to detect potentially problematic code in dataflow programs. This method was designed to work in graphical, dataflow driven languages and was initially implemented in LabVIEW [44]. Code smells are heuristics that help find areas of the code that can be improved. They were originally designed to check for maintainability issues [29], and this idea has been modified in this research to find performance problems.

Simply detecting the smells may not always be enough as it is possible that not all detected smells will impact the performance of the program. Removing false positives from the information provided to users can make the tool more accurate. To achieve this, SDPA was enhanced by adding the ability to profile a program. Smell-Aided Profiling (Section 4.4) allows users to profile the areas in the code that have smells in them.

Once problems have been detected and isolated, semi-automatic transformations (Chapter 5) are provided to help users fix the problems. Some transformations can be applied with the click of a button, while others require input from the user to be fully applied.

The effectiveness and usefulness of these ideas were evaluated through three studies (Chapter 6). The first of these tested SDPA on a corpus of real world LabVIEW programs that were known to have performance problems as well as solutions pro-

vided by experts. This study evaluated metrics such as the prevalence of detected smells (6.1.1), the impact of the transformations (6.1.5), and the impact of the tool compared to experts (6.1.6). The other two studies were user studies that investigated how well the tools helped users find (6.2) and fix (6.3) performance problems. Together these evaluations showed that the approach aided users in finding and fixing performance problems in a dataflow language.

Finally, these ideas were generalized to show that they are not limited to LabVIEW. Chapter 7 details how SDPA and transformations were implemented in another dataflow language, Yahoo! Pipes [38], to detect performance problems and apply transformations to fix the problem. A set of four smells were collected from pervious research [34][59][90][100]. The prototype was then evaluated to determine if smells could be detected and the impact of those smells. This evaluation revealed an average performance improvement of 29% in terms of execution time.

1.3 Contributions

The overall contribution of this Thesis is Smell-Driven Performance Analysis (SDPA), which gives end-users the ability to quickly find and fix performance problems in dataflow code by detecting performance smells and providing transformations to fix the detected issues. The process of creating SDPA resulted in following specific contributions:

- The concept of *Performance Smells*. Code smells have primarily been used in related work to detect maintainability issues in code. This work modifies this

concept and creates performance code smells, which can detect performance problems.

- An efficient algorithm to detect performance smells. SDPA relies on being able to detect issues quickly and effectively. Interrupting the user to run a smell detector would likely annoy the user. The algorithm used for SDPA runs in the background while the user is programming and can provide results quickly when the user requests them.
- The concept of *Smell-Aided Profiling*. The majority of profilers are only able to detect that there is an issue, but pinpointing what the issue is and where it is located can be problematic. By using the detected smells to aid profiling, hot spots can be filtered and solutions can be given based on the smells in a given area.
- *Semi-automatic smell-driven performance transformations*. These help end-users quickly and efficiently fix the problems found by SDPA. The transformations in this dissertation in part rely on user guidance and information to be fully applied. This system helps guide users through the process by providing information and asking specific questions.
- A user interface that can notify users of the smells that have been detected and provide the necessary information for users to correct the code.
- A description of how this work might be generalized to other data flow languages. This dissertation demonstrates how to take SDPA and implement a small prototype for another dataflow language. This provides the framework to create tools for other similar languages.

Chapter 2: Related Work

The objective of the proposed research is to help end-user programmers to find and fix performance problems in their code, with a specific focus on visual dataflow languages. Related work has investigated ways to use design critics, code smells, and log analyzers to check for a variety of problems in programs such as maintainability, correctness and performance. In addition, once these problems have been found, there has been research on the best way to apply fixes include code refactoring and program transformation. However, currently there are no methods that check the performance of dataflow code using a combination of static and dynamic analysis and provide possible transformations to the user that will allow them to easily fix the problem.

2.1 Approaches for Finding Problems in Code

Before problems can be fixed, they must first be found. There are two main categories of approaches for finding problems. One kind of approach is to dump large amounts of data on the programmer, who is then required to find the problem revealed by the data. The second is to perform some analysis to generate some guidance to the programmer, and in some cases even to specifically identify potential problems automatically. The subsections below describe these methods with a specific emphasis

on tools and methods related to performance problems.

2.1.1 Approaches that require programmers to sift through data to find performance problems

The most basic method for helping programmers to find performance problems is log analysis, which records data about the activity of a program during runtime and presents that data to the programmer. For visual programming languages, a typical example of a basic logging program is the Desktop Execution Trace Toolkit [41], which performs dynamic code analysis on a LabVIEW program to give users low-level insight into the code execution. The results provide a chronological view of the VI events, queue operations, reference leaks, memory allocation, unhandled errors, and subVI execution. A user can also highlight individual events to obtain additional information such as the call chain or double-click on events to highlight the corresponding object on the block diagram. The results of a trace are shown in Figure 2.1.

Tools such as this provide an overwhelming amount of data to the user, and often the results include every operation that has been run. This means that any potential problems get buried in the large amount of output that is provided. For example, imagine that the performance issue arises only when a loop is running. The trace results will not only show timing and memory information for the loop, it also shows the information about every other node in the VI. To actually recognize a performance issue, a user would have to scroll through every item in the log to determine if it is

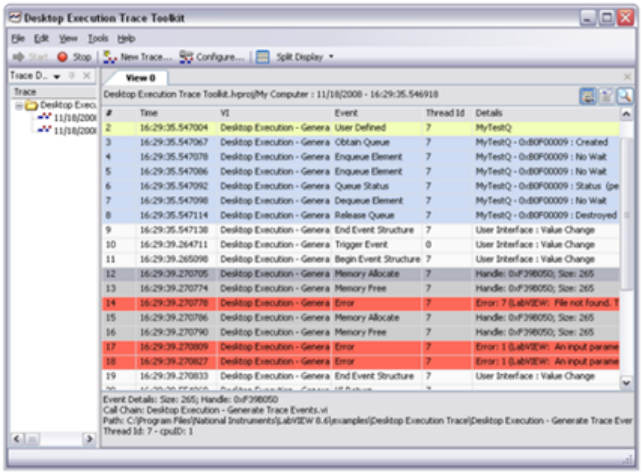

Figure 2.1: Sample output of the Desktop Execution Trace Toolkit

problematic.

More advanced logging tools still dump large amounts of data on the user, but they provide different visualizations of that data than the tabular layout shown in Figure 2.1. Given the rising importance of parallel programming in science [11], many of these tools are focused on presenting data from parallel programs. For example, one tool analyzes the log of a MapReduce program and then visualizes it in terms of MapReduce-specific behaviors [91]. Another option is to summarize the trace logs and generate reports with hotspot visualizations [30]. Just because a program spends much time in a certain area, however, does not mean that that amount of time is "too much" or actually a performance problem. The programmer is still expected to look at visualizations and judge whether a real performance problem even exists.

There are other tools that take this data dump and attempt to filter it to reduce the amount of work for the developer and attempt to draw out some conclusions. One tool, DISTALYZER [67], analyzes the logs from distributed systems. It uses

using machine learning techniques to compare system behaviors extracted from the logs (both good and bad logs) and infers connections between system components and performance so that it can summarize the relationship between these two.

While these filtering tools do reduce the amount of information provided to users they still lack the capacity to pinpoint specific errors and instead can simply detect that something has happened, and provide perhaps a vague idea of where it might have occurred.

In contrast, the tools created in this dissertation do not require end-user programmers to sift through large amounts of data or visualizations to track down performance problems, thereby potentially saving programmers a great deal of time - particularly for programmers who lack the skill or experience needed to interpret program logs or visualizations.

2.1.2 Tools and Methods that detect that a problem but do not guide programmers to its cause or fix

The tools discussed in this subsection primarily focus of detecting *what* has happened, not *why* it is happening. For example, a method might notice that there is excessive garbage collection or memory use and be able to notify the user that this has happened, but not why it happened.

Two tools [51][71] look for anomalous behavior in the execution. Ganesha [71] aims to diagnose faults transparently (in a black-box manner) in MapReduce systems, by analyzing Operating System level metrics. It uses the fact that fault free nodes

will have similar behavior, so it looks for asymmetric behavior to identify that a problem may be present. MACEPC [51] tests distribute systems by first determining the expecting behavior and then comparing the resulting behavior. Any anomalous executions indicate a potential performance problem.

Two tools [28][32] work on filtering information from the stack to inform users of potential problems. The first such tool, X-TRACE [28] tries to help developers better understand the performance of their system by using extensions to the existing protocol stack to trace the flow of messages across protocol layers, networks and applications. The other tool StackMine [32], mines callstack traces for information and then applies pattern matching and clustering to try and discover abnormal uses of CPU consumption or when a program is waiting too long to perform and action.

Two other tools [3][35] attempt to find the potential root cause of performance problems, focusing on configuration and human errors. X-Ray [3] applies binary instrumentation to monitor applications as they execute. It then attributes performance costs at a fine grained level, which it calls a block (e.g. instructions, system calls). It then uses dynamic information flow tracking to estimate the likelihood that a block was executed due to a performance problem. Performance regression Root Cause Analysis (PRCA) [35] uses unit tests to continuous monitor the program. When a regression in performance is found, it can indicate that something has changed and indicate which version in the revision history is problematic.

PIP [80] is a technique for finding bugs in distributed systems. It requires users to annotate their programs along with providing expected behavior. It then is able to compare the actual behavior and expected behavior to expose potential performance

problems and structural flaws.

Another tool, Lag Hunter [47], is designed to find lag errors in widely deployed systems. It does this by gathering run time data from many different users. It then analyzes this data to find times when the user experienced noticeable lag. These instances can be reported to developers as potential performance problems, but it does not pinpoint the cause of the problems .

All of these tools are able to detect that there is a problem, but they lack the ability to give users any assistance at clarifying the cause or even finding the problem. In addition, they all require many executions to find potential problems, which end-user programmers would likely find cumbersome or impractical. They are of limited use, therefore, in providing a starting point for solving the problem.

2.1.3 Methods and tools that guide programmers to potential problems in non dataflow code

One approach for helping programmers find problems and their respective causes is for a tool designer to collect expert wisdom about program problems, and then to create tools that analyze programs (statically or dynamically) for those problems. Such a tool is often called a "design critic", and each tool has a set of internal rules that a tool that it uses to detect issues. The specific heuristics used differ from tool to tool, but some common heuristics involve the use of "code smells" [29] or "antipatterns" [9][54].

The most well-developed tools in this area, which have already made it into indus-

trial practice, incorporate heuristics for finding potential maintainability problems in object-oriented code (e.g., [66][84][93]) Some of the code smells for object oriented programming are excessively long methods or excessively large classes. Tools inform programmers about apparent problems via a standalone application [93], via a dialog window within the IDE [84], or via annotations inserted alongside the code [65]. Different heuristics will be required to find performance problems in dataflow code rather than maintainability problems in object-oriented code, but the lessons from this related research did help guide this work. In particular, of these three methods for informing programmers about maintainability problems, the third provides the most direct visual association between potential problems and the related code, so a similar method was tried for displaying alerts to programmers about performance problems in the user interface for this work.

In addition to tools that directly search a program for maintainability code smells, some research has attempted to add on to these smells in novel ways. Two tools [50][60] have rules for both good and bad smells. Another [1] tries to define a risk and severity for a subset of the code smells to try and estimate the potential consequences and help users focus on the worst smells. Another tool, DECOR [64] allows users to check for code smells as well as define their own maintainability smells. A final tool, IPLASMA [61], detects deviations for good code smells to try and detect design flaws. While these systems do extend the idea of code smells, they still do not provide heuristics to detect performance problems in dataflow code.

Heuristics have also been used to find correctness problems in spreadsheets. Spreadsheets users struggle with many correctness and maintainability problems [16] and

often have difficulty being able to solve the problems they encounter [15]. Researchers have developed heuristics for spreadsheets by soliciting expert spreadsheet experts for heuristics [21], and by adapting the maintainability-focused object-oriented heuristics (above) for use in finding correctness problems in spreadsheets [2][36]. Tools based on these heuristics output a list of smells to a separate file, requiring the programmer to map back to the related portions of spreadsheets. Empirical studies show that these heuristics show promise for finding correctness problems [2][20][21][36]. The smells that are used to detect correctness problems in spreadsheets tend to look for elements specific to spreadsheets. For example, the Middle Man smells in [36] tries to detect formulas that only gather data and then pass then on to other cells or worksheets. This smell tends to complicate the structure of the spreadsheet and reduces the quality. This thesis focuses on finding performance problems in dataflow code, rather than correctness problems in spreadsheets.

A few tools include heuristics for finding performance problems in object-oriented systems via dynamic analysis. For example, one paper presents an approach that analyzes logs to detect performance problems in enterprise applications built using component based frameworks [73]. The tool is focused on detecting patterns that affect large internet-enabled enterprise systems rather than programs that end-users might create, which means that the heuristics are specific to those kinds of applications. For example, one heuristic detects when Java session beans, which are used to implement process logic or workflow for business entities, are being overused in an application. Other tools find problems due to poor cache use [75], race conditions [8][24][27], memory leaks [98], or loop inefficiencies [69]. Another Tool [4] gives users

the ability to create small scripts that can check their programs for data races and poor memory system behavior.

Several tools provide heuristics for finding readability and performance problems in object-oriented systems via static analysis. These include LINT [45] and CodeAdvisor [23], which check for coding style issues that can impact readability and even correctness, such as variables being used before being set, conditions that are constant, and calculations whose result is likely to be outside the range of values that can be represented in the type used. Other tools, such as Java Critiquer [77] and LISP-CRITIC [26], check for performance problems. For example, the LISP-CRITIC includes heuristics to detect the need for replacing compound calls of functions with simple calls to more powerful functions, or to detect dead code. All of these tools rely on textual pattern matching methods such as regular expressions for expressing and running heuristics. Although some of these heuristics might be conceptually relevant to dataflow code (such as checking for opportunities to combine operations into single operations), an entirely different non-textual method will be required for expressing and evaluating the heuristics.

There are also a set of tools that use both static and dynamic checking to determine correctness of object-oriented code, primarily through the generation of test cases. One recent example [85] combines the results from three distinct bug finding tools, two of which provide static analysis, and one which uses dynamic analysis to generate test cases. Each of the distinct tools was run on a set of 105 java classes and then the results were analyzed to determine how many of the bugs were found. The results showed that combining static and dynamic analysis was able to find more of the bugs

in the code. While this is a good result, this method is not actually integrating the different tools; it is simply applying static and dynamic analysis separately and then looking at the results of both. In addition, this is focused on test case generation, rather than performance and wouldn't apply to the work discussed in this thesis.

There are several tools that use the idea of performance anti-patterns [88][89] to find performance problems. These anti-patterns look for common design decisions that may lead to performance problems. One example is One-Lane Bridge, which describes the a point in the execution where one, or only a few, processes may continue to execute concurrently, while all other processes must wait. These anti-patterns are detected by monitoring the execution for specific symptoms. Two tools [92][99] apply these ideas to models to try and detect the performance problems in the design phase before they have been implemented. Another tool [18] checks for two anti-patterns in Object-Relational Mapping, a conceptual abstraction for mapping database records to objects in an OO language. The final tool, Performance Problem Diagnostics [94] runs on an already deployed application. It monitors the execution while changes to the workload are made to try and detect any potential anti-pattern.

The tools that detect these anti-patterns are all looking at design decisions that lead to performance problems, and none of these tools can pinpoint areas of the code that might be problematic. They can recommend methods or classes to investigate and potential design solutions, but much of the work is left to the developer.

Due to the differences in how memory is used between object-oriented and dataflow code (that is, creating/deleting objects versus shared references to arrays of data that act as queues), it is likely that different heuristics will be required for finding

performance problems in dataflow code. It is also unclear whether heuristic based methods would be as readily useful to end-user programmers as to the well-trained professional programmers who use object-oriented languages.

2.1.4 Methods and tools that guide programmers to potential problems in dataflow code

So far, there have only been a few substantial lines of work specifically aimed at finding performance problems in end-user programmers' dataflow code, which is the specific focus of the work detailed in this dissertation.

The first of these is VI Analyzer [42], a program checker for LabVIEW, which is a graphical programming language and supporting environment [39]. This tool includes many checks for coding style and a handful for performance. However, there are only a few tests to detect performance issues, and some of these focus on settings inside of a user's program rather than the code itself. For example, one of the tests checks to see if debugging is enabled as this often slows down execution speed. With so few performance related tests, VI Analyzer can only catch a small portion of performance issues, which greatly reduces the usefulness of this tool for finding performance problems. The tool is not situated in the task, meaning it is contained in a separate window from the LabVIEW IDE. To perform the suggested fixes the user has to continuously switch between the editor and this tool. Consequently, the format of the results makes it difficult for users to follow the suggestions. This thesis broadens the range of performance problems that can be detected, and directly

incorporates feedback about performance problems in a way that reduces the need for the user to flip back and forth from the problem to the related code. Figure 2.2 shows an example output from this tool.

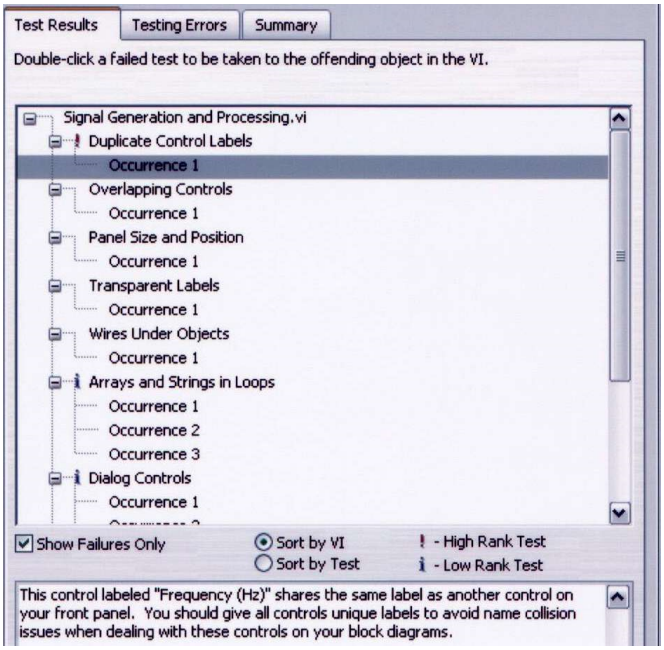

Figure 2.2: Example output from VI Analyzer

Two other lines of work specifically related to finding performance problems in end-user programmers' dataflow code were focused on analyzing Yahoo! Pipes. Like VI Analyzer, each of these can only check for a very narrow set of performance problems. One line of work yielded a tool that can find all operations that are performed multiple times, and it can identify parts of the code where data-generation operations can be moved earlier in the program to improve performance [34]. The other area of work provided a tool that identifies opportunities for re-ordering (scheduling) operations to improve memory utilization [100]. The work in this thesis investigated

a broader range of performance problems, including problems that require user intervention to correct.

Another line of work has focused on finding maintainability problems in dataflow code. This work focused on Yahoo! Pipes, a dataflow programming language and supporting environment for creating mashups. Researchers investigated how to adapt the heuristics above maintainability problems in object-oriented for the purpose of analyzing dataflow code [90]. They then ran these heuristics on a corpus of Yahoo! Pipes programs, finding that most of these programs had at least one problem according to the heuristics. Many of the specific heuristics used could have potential performance implications and therefore be useful for improving performance; an example is the "Noisy Module" heuristic, which determines if a dataflow element "has unnecessary fields, making a pipe harder to read and less efficient to execute." However, unlike the work detailed in this thesis, this research did not investigate the extent to which fixing problems found by these heuristics would improve performance.

2.2 Methods for Helping Programmers to Fix Problems in Code

Once a problem has been found in a program, the next step is to fix it. Some tools ignore this step entirely under the assumption that detecting and informing the user about a problem is enough for the user to correctly fix the issue in their code. However, some approaches aim to fix these problems automatically or provide a guide that provides assistance so that programmers can control how the code is fixed.

Of particular interest to this thesis are methods that help users to refactor or

transform their code to fix the problem. Refactoring and transformation are similar ideas, but they affect the code slightly differently. Refactoring is the process of changing software in a way that does not alter the external behavior of the code but improves its internal structure and makes it easier to understand or maintain [29]. Code transformation, on the other hand, takes a program and generates another program. The transformed program is most often required to be semantically equivalent to the original, relative to a particular formal semantics, but some transformations result in programs that semantically differ from the original in predictable ways [74]. Hence transformation is a generalization of refactoring.

2.2.1 Refactoring Tools

When the idea of code smells [29] was proposed, the smells were designed to aid programmers in the refactoring of their code to make it easier to understand and maintain. As such, there are many tools that have been built that help users refactor their code based on the smells defined by Beck and Fowler [29]. The tools that use these code smells are designed to help users find maintainability issues in object-oriented code. These tools [31][63][66] detect the code smells in a user's code and then provides refactoring help to eliminate the maintainability issues. How these refactorings are presented vary greatly between systems. In one system [66], the tool acts as a guide to the user and helps them step through the refactoring process. This is designed to help teach the users about the refactoring process so that they won't always be reliant on the system. Two similar methods [31][63] apply automatic

refactorings on user's code or suggest automated refactorings rather than helping users through the process. Unlike the work done in this dissertation, these tools are focused on fixing maintainability issues rather than performance problems.

While the original code smells have been shown to be effective in refactoring textual code, the idea has also been expanded to new smells and languages. Of particular interest to this thesis are tools that apply refactoring to dataflow code. One method [90], discussed in more depth in Section 2.1.4, is focused on using code smells to find maintainability problems in dataflow code, particularly Yahoo! Pipes. Researchers investigated how to adapt the heuristics above maintainability problems in object-oriented for the purpose of analyzing dataflow code [90]. They then ran these heuristics on a corpus of Yahoo! Pipes programs, finding that most of these programs had at least one problem according to the heuristics. With these heuristics in place, refactorings were introduced to help fix the issues that had been found. These refactorings were aimed to reduce the complexity of the code, increase the abstraction, and adjust the code to community style standards. Some of these heuristics may be useful for improving performance, but most of the performance benefits are secondary to the result of getting rid of unnecessary code. In addition, this research did not investigate the extent to which fixing problems found by these heuristics would actually improve the performance of the programs.

There are also two systems [34][59] that are designed to improve the performance of dataflow code through refactoring. The first method [34] applies refactorings to Yahoo! Pipes mashups to minimize the redundant operations that are in a program and to reorder the program so that all data creating operations occur first. For

example, if the same operation occurs five times in the program, this tool would refactor the code so that the operation only occurs once and the result is saved for use the other four times. The second method [59] enhances the performance of a Yahoo! Pipes mashup by automatically refactoring the structure of the data flow. It does this by first annotating the operators in the program. The tool then applies refactoring rules to the code to generate all semantically equivalent data flows. Once this process is complete there will be many different semantically equivalent programs. The tool then applies heuristics to determine which program has the smallest execution time, which the tool then recommends to the user. While these tools do focus on the performance of dataflow code, they only apply refactorings ensuring that the semantics of the mashup remains the same, which would not fix many of the performance problems that appeared in real programs (See Sections 4.1.1 and 4.1.2). Therefore, only applying refactorings is not the best approach to solve performance problems in dataflow code.

There has also been some research in applying refactorings to object-oriented code to help improve performance. One tool, ReLooper [22], helps users create parallel programs by converting sequential code into parallel code using existing parallel frameworks in C# or Java. In particular, this tool takes advantage of the Java data structure ParallelArray which is an array data structure that supports parallel operations over the array elements. For example, one can apply an operation to each element, or reduce all elements to a new element in parallel. To refactor an array to a ParallelArray, this tool analyzes loop iterations to determine if they are safe for parallel execution, and it then replaces loops with the equivalent parallel operations.

Preliminary experience with refactoring real programs has shown that ReLooper is useful, and it has been shown to produce a speedup between 1.3 and 1.9 on a two core machine. This tool takes advantage of a data structure that is specific to Java and would have to be modified to work with dataflow code. Namely, LabVIEW already parallelizes dataflow execution when possible, yet performance problems still remain. Clearly, more remains to be done.

2.2.2 Program Transformation other than Refactorings

The approaches in Section 2.2.1 apply refactoring to try and fix a range of problems from maintainability to performance. However, at times the best way to apply a solution to a performance problems is to perform code transformations (See Section 5.1). That is to semantically change the code to eliminate the offended code and increase the performance. For example, consider a program that contains a parallel loop that runs continuously to read information from a device. Since the loop is continuously running, it causes the rest of the program to slow down and the user interface to be unresponsive. A simple solution to this problem is to add a wait to the loop. This is a very simple transformation that changes the semantics of the program, which is necessary to fix the performance issue.

Program transformations often attempt to improve the performance of a program, and can be of two types. The first type is very similar to compiler optimizations, in which transformations are applied to a program without notifying the user of what is being done [5][76]. These tools have a set of transformations that it will apply right

before a program is compiled. However, research has shown that simply applying transformations without involving the user is not as effective as manual transformations to improve performance. The gap between automatic transformations and what can be done by hand has been widening over time, due to several factors. Among them are the complexity increase in microprocessor and memory architectures [19][52], and to the rising level of abstraction of popular programming languages and styles [6]. In addition much of the program transformation research has focused on partial evaluation, specialization and simplification, which are not focused on performance optimization.

Due to this, rather than simply applying the transformations to a program, there have been many tools designed to help users apply transformations to their code and choose which transformations would be best. One area where this method is particularly useful is Digital Signal Processing (DSP) applications, which usually measure, filter and/or compress continuous real-world analog signals and often require precise timing and performance. The research in the area takes a DSP application and applies transformations to improve performance. One method [56] does this by investigating the loops and arrays in a program and determining if there are any mismanagements of the cache. If any are identified, the tool will recommend transformations that could improve cache performance.

Another method tries to apply ideas from functional programming to DSP programs and apply transformations such as removing loops to improve their performance. A final method [37] tries to improve the throughput of DSP applications by using an ordered set of transformations and showing which ones could be applied to

the user. If desired these transformations could also be automated. All of these methods were shown to improve the performance of the code if the transformations were applied to a program, however, some of the methods used are specific to DSP programs, while others [70] try to force functional programming fundamentals on users instead of helping them learn and understand the language they are using. Although semi-automatic transformations have proven successful in specialized domains such as DSP, little work has yet investigated how to analyze dataflow programs. For instance, one line of work produced a tool that can find all operations that are performed multiple times and that can suggest operation reordering to improve performance [34]. This thesis and related research will be the first work to investigate how to help end-user programmers fix a wide range of performance problems in data flow code.

Chapter 3: General Idea

Smell-driven performance analysis (SDPA) fills the gap identified in Chapter 2, namely the need of end-user programmers for help with finding and fixing performance problems in dataflow code. The section below describes this technique at the conceptual level, from the standpoint of an end-user programmer who needs help with solving a performance problem. Technical implementation details are deferred to the following chapters.

3.1 General Idea

Suppose an end-user programmer has a certain program and is unsatisfied with some aspect of performance. For example, the program might be missing realtime deadlines or dropping samples.The programmer will open up the problematic program in the development environment for the visual dataflow language. This environment would be enhanced with a new tool that aids in analyzing performance.

Such a tool would take advantage of the fact that the causes of many performance problems are specific structures in the source code (Section 4.2 will discuss in detail). Some of these might resemble canonical "bad code smells" used to characterize object-oriented code that appears to be hard to maintain [29]. In object-oriented code, smells are found using heuristics, such as a rule that any method with more than some N lines

of code is potentially poor quality. Likewise, a tool in SDPA would statically analyze the program to detect areas matching heuristics, though in this case, the heuristics would be "bad performance smells", rather than focused on maintainability. Each heuristic would describe one common structure that often causes problems (e.g., each performance-related structure listed in Section 4.1.2.2 for LabVIEW).

The next step in SDPA is for the tool to insert visual alerts (e.g., icons) into each area of code that matches a heuristic. To avoid annoying users with uninvited advice, the tool by default would not actually show alerts unless the user asked for help finding a performance problem (e.g., by clicking a "Diagnose Performance" button). A configuration setting could allow users to continuously run the tool in the background, with any alerts appearing as passive notifications.

Providing users with a list of detected smells could help them in the debugging process, but there is no guarantee of the detected smells actually being responsible for performance problems. If the tool gives users too many false positives, (i.e., smells that do not actually cause a performance problem), it is very likely that they would grow frustrated with the tool and stop using it.

To help alleviate this potential problem, the tool can provide users with the ability to profile their code and find the actual hot spots. If the user selects the "profile code" option, the tool will first begin by gathering timing and memory data for every smell that has been detected in the program, as well as the total execution time and total memory usage of the program. Once the program has finished executing, the information for each smell can be compared to the information for the program. Any smell that takes up a large proportion of memory or time can be said to be a "True

Positive" as it actually causes a performance problem. The smells that are profiled by the tool that do not greatly impact performance could be considered likely "False Positives" and the tool will filter them out.

When code is profiled, the hot spots of a program are detected, and the next step is fixing those areas. If the user fixes a smell, it is likely that the hot spot will dissipate or disappear.

The tool would provide advice on how to fix these problems. Specifically, if a user clicks a visual alert, the tool would summarize the heuristic that had led to that alert. It would explain relevant programming concepts and describe a potential code modification aimed at eliminating the problematic structure (ideally without altering the code's functionality). Fixing a problem would make the alert disappear (or the user could just dismiss it).

In addition to a text description of a solution, the tool will supply users with the ability to semi-automatically transform the detected smells. There may be a set of available transformations for each specific smell rather than just one.

Some straightforward transformations can be applied at the click of a button, but others might be more complex. So the tool would need to be able to handle multiple types of potential transformation. For the simpler transformations, all that would be required is the click of a button. Other transformation may require a wizard that can guide users through the process of applying a transformation by asking relevant questions, supplying users with a thorough step-by-step guide, and providing advice on possible ways to remove the smell.

In terms of user interaction, SDPA resembles Surprise-Explain-Reward, whereby

a tool helps users to find and fix spreadsheets' computational errors [96]. In that approach, users can enter a testing mode, within which the tool shows question marks alongside elements of the program (spreadsheet cells) on the screen to pique curiosity. When the user responds, the tool explains how to test part of the spreadsheet. Users are rewarded for testing by changes in the user interface. SDPA uses alerts to draw attention to targeted parts of a program, presents advice to explain how the user should respond, and hides alerts to reward user responses. Our technique differs from Surprise-Explain-Reward in that it is guided by heuristics (smells) and is oriented toward helping users with fixing performance problems rather than just finding computation errors (functionality).

3.2 Research Claims

While the ideas described in the previous section were implemented as part of the LabVIEW IDE, they can apply to any dataflow language. To validate these ideas, nine claims were specified that summarize what the approach should be able to accomplish. These claims were used to design evaluations that determined if the research met the goals. Table 3.1 below shows these claims as well as the corresponding evaluations that will be used to determine if they are met.

The first six claims required testing a prototype tool on real world LabVIEW examples. Claim 1 relates to the prevalence of the gathered smell by determining how often they actually occur in real world examples. This is important because if few of the smells that can be detected exist in real programs, then it will not be a

Table 3.1: Key claims and corresponding evaluations

	Claim	Evaluation	Section
1	Smells are common in end-user dataflow programs that have performance problems	Real World Examples Study	6.1.1
2	SDPA automatically identifies most performance problems that experts find	Real World Examples Study	6.1.2
3	Smells identified by SDPA usually indicate real performance problems	Real World Examples Study	6.1.3
4	Smell-Aided Profiling improves the false positive rate of SDPA and negligibly impacts the true positive rate	Real World Examples Study	6.1.4
5	Tool-based transformation of most smells improves program performance	Real World Examples Study	6.1.5
6	Tool-based code transformations semi-automatically improve performance almost as well as experts do	Real World Examples Study	6.1.6
7	SDPA helps end-user programmers to more quickly and successfully find and diagnose performance problems	User Study	6.2
8	Smell-based performance tools (including SDPA, Smell-Aided Profiling, and transformations) together help end-user programmers to more quickly and successfully fix performance problems	User Study	6.3
9	SDPA and tool-based transformations can be applied in other dataflow languages	Generalizability Study	7.4

very effective tool. Claim 3 relates to the impact of those detected smells, to ensure that the tool mostly detects smells that actually affect performance.

Claims 2 and 4 relate to the validity of the detected smells and that they are filtered correctly once profiled. Claim 2 compares the prototype's impact to the optimizations performed by expert LabVIEW programmers to determine if there are any optimizations that the tool misses. Claim 4 focuses on the false positive rate of

SDPA and Smell-Aided Profiling to ensure that profiling does filter out false positives without removing many actual problems.

Claims 5 and 6 relate to the transformations and how much they improve performance. Claim 5 focuses on the transformed code compared to the original code, while Claim 6 compares the resulting improvement to the performance improvement that experts were able to achieve. Combined, these will show how well the tool could potentially help to fix problems.

Testing Claims 7 and 8 will involve evaluating the tool with actual users to ensure that it is usable and that it helps them *find* and *fix* performance problems in their code. These studies will involve participants trying to debug real world LabVIEW examples for performance problems both with and without the tool. The time taken and success rate will be tracked to allow a comparison between the two methods.

Finally, Claim 9 essentially states that the overall approach is generalizable. This is important to show that SDPA and transformations are not specific to LabVIEW, which they were implemented in, and in fact apply to similar languages.

Chapter 4: Smell-Driven Performance Profiling

In the debugging process, a key task is to identify the problem and find where it is occurring in the program. Once it has been found, a fix can be made that will correct the problem. Finding and fixing performance problems can be particularly challenging for end-users who lack the expertise needed know what to look for or even where to begin looking. This can lead to frustration and limit adoption or usage of a language.

This chapter will describe the SDPA technique developed to help end-users find performance problems by detecting patterns that have been known to cause performance issues. The approach has been implemented in a lightweight add-on to LabVIEW as a way to evaluate their effectiveness [17]. This prototype also provides guidance on ways to fix such problems. Chapter 5 will discuss the more extensive assistance provided to semi-automatically fix performance problems.

4.1 Preliminary Studies

Helping end-users with performance issues requires knowing what problems end-users struggle with, and how they might best be solved. Consequently, two studies were performed to investigate problems that LabVIEW users typically encounter as well as potential ways to use this information to design better tools. The first study involved

new LabVIEW users who were interviewed after a brief tutorial about the barriers that they faced while learning LabVIEW. In the second study, LabVIEW technical support personnel were interviewed about common problems often found in end-user code and coding elements that led to bad LabVIEW programs.

4.1.1 Post Tutorial Interview

The first study was performed after a LabVIEW workshop given at Oregon State University by a representative from National Instruments. This workshop was designed to help teach new LabVIEW users about the language and guide them in the creation of a simple program. After the workshop was over, the participants were interviewed in a group setting (focus group) to gather informal feedback about any issues or barriers they faced while learning LabVIEW as well as topics that they found particularly confusing. This study was designed to ferret out possible research areas in LabVIEW and ways to help new users. It was not focused specifically on performance problems.

4.1.1.1 Methodology

To recruit participants, information about the LabVIEW workshop was distributed to the Robotics Club and Mechanical Engineering classes at Oregon State University that use LabVIEW for assignments. A total of 11 individuals showed up to participate in the workshop. Of those, nine were interviewed for the study (two were under 18

years old and had to be excluded). LabVIEW programming experience varied greatly amongst the participants, though all of the interviewees had at the very least used LabVIEW previously in a class or club.

The participants were interviewed in a group setting (focus group style), with questions being suggested to the group by a researcher. These questions covered topics such as the material they had just learned, any problem they experienced programming in LabVIEW, and about their experiences with LabVIEW in the past. Any answers were written down by the researcher.

4.1.1.2 Results

The biggest issue revealed during the interview was that programming in LabVIEW is very different from programming in a textual language. All of the participants mentioned that it had taken (or was taking) a long time to get used to a dataflow language and that they often would create LabVIEW programs using textual language concepts, such as using variables to store data rather than passing values directly between operations as in the dataflow paradigm. For example, one participant mentioned that at first he created variables any time he wanted to store information in a LabVIEW program. As the program got larger, the number of variables got unwieldy, causing issues with the behavior and results.

Another issue that arose was the handling of arrays in LabVIEW, particularly when in conjunction with loops. Users of textual languages commonly use a **`For Loop`** to access all the elements in an array. However, in LabVIEW there are multiple

methods that can be used to achieve this. Several users mentioned that there is no guidance on what methods will cause performance problems and which will not. They would often just go with the method that they could get to work first, which sometimes led to inefficient code. If the wrong method was used, the problems would often not be revealed until the program had gotten larger. By this time refactoring the code was a nightmare as the "wrong" method had been integrated with all the existing code.

In general, the participants mentioned that while they could create a program in LabVIEW rather easily, they often had a hard time debugging their LabVIEW programs if they ran into problems. This was sometimes due to the lack of knowledge and not knowing a more efficient way to use certain modules.

In conclusion, this small, preliminary study suggested that users encounter several different problems when learning to use LabVIEW. Sometimes performance problems arise because people are uncertain of the best techniques to use while coding in LabVIEW. In addition, the participants did not feel like they had sufficient tools to debug the problems inside of LabVIEW and often had to start over from scratch to create a working program.

4.1.2 AE Specialist Interviews

After the small, informal focus group study (Section 4.1.1), a second study was done to carefully assess (1) what specific performance problems are most common with LabVIEW users, and (2) the extent to which these problems are attributable to users'

code. In this study, application engineer (AE) specialists from National Instrument were interviewed. These participants are considered expert LabVIEW programmers, and their job is to answer customers questions about LabVIEW and to assist in troubleshooting programs.

4.1.2.1 Methodology

National Instruments provided an email list consisting of 13 AE specialists. These individuals were emailed and nine agreed to participate in a 30 minute interview session. Six of these interviews were done over the phone, while the remaining three were done in person. The questions and process remained the same regardless of the interview method.

The questions asked during this study were aimed at determining the kinds of problems that the AE specialists had experienced while assisting customers. To get the participants thinking about their experiences, they were first asked to describe a recent customer call where they had provided assistance. They were asked to describe the program as well as the problems the customer had been dealing with. After this discussion, questions were asked about the typical types of problems that they often see in customer code as well as problems that they are aware of that users often run afoul of.

Thematic analysis, a grounded content analysis method that identifies recurring topics [55], was used to analyze answers about user problems. This created a set of mentioned problems. The number of times each problem was mentioned was counted.

A second researcher reviewed each categorization to verify that it appeared appropriate.

4.1.2.2 Results

All participants had at least 3 years of experience, with the average level being just under 5 years. Each had served as an Application Engineer (AE) for 1-2 years before becoming an AE Specialist. Most reported their workload had been 5-10 straightforward support issues per day as an AE, and that their workload as an AE Specialist was approximately 10 relatively complex issues per month.

Participants' answers revealed a long list of recurring problems that, for the most part, resulted directly from the structure of end-user programs. These structures were typically problematic because of their impact on the performance of programs. For example, ***Build Array in Loop*** refers to the situation when a **`Build Array`** node (which allocates memory) is nested inside of a **`Loop`**. This causes frequent memory allocation and de-allocation, which in turn, causes memory fragmentation, unnecessary memory collection overhead, and ultimately poor execution time for the program as a whole. As another example, ***Too Many Variables*** refers to a situation where one virtual instrument (VI) contains more than a few variables. A VI is a single block of computation, analogous to a function; the program as a whole is also considered a VI. AE Specialists indicated that people sometimes get confused about how to keep variable values consistent when many variables are used, and in code that compiles to multiple parallelized threads, this can result in race conditions.

Performance problems manifested at runtime in a variety of ways that included missed deadlines, race conditions, jitter , dropped samples, system crashes, and data corruption. In one example, a LabVIEW user contacted NI about a program that would at first run fine, but after 10-15 minutes of execution would slow down and eventually blue-screen the computer. To try and fix the problem, the user kept increasing the virtual memory size, until this solution was no longer viable. After a brief code review, the program was found to be continually creating duplicates of a 650 MB array (***Multiple Array Copies***). Each duplicate caused the memory usage of the program to increase until the computer crashed.

One topic that interviewees did not mention was a unified resource for finding and solving performance problems. The names of problems above are names that were assigned; even though different AE Specialists often saw the same problems, they had no standardized names for problems, let alone a consolidated reference for characterizing problems. Instead, they relied on experience and intuition to diagnose trouble. Lacking such expertise, end-users' choices are to purchase technical support, buy better hardware, or use another language.

Overall, 10 of the 13 problematic structures that the AE Specialists came up with had direct implications for performance according to interviewees. These implications were not minor, either: All of these code structures had such an obvious, often-catastrophic impact on performance that users had taken the time and expense to talk with technical support. In addition, several AE Specialists mentioned that training materials created by National Instruments to aid users in creating better performing code might have patterns that the interviewees were unable to remember. From these

materials, two additional smells not mentioned by AE Specialists were found.

Table 4.1 briefly details these smells. This table also includes the total number of AE Specialists who mentioned that smell during their interview. The smells gathered from training materials have a T in this column. All smells are discussed in more detail in Section 4.2.

Table 4.1: Smells gathered from AE Specialists. [P] indicates a smell that may impact performance.

Smell [P] = Performance Problem	Description	# who reported
Too Many Variables [P]	Too many variables in a VI can lead to race conditions, The suggested limit ranges from 2 to 5 variables per VI, more than this raises the risk of inconsistent behavior.	8
Build Array in Loop [P]	When a build-array node is inside of a loop, it builds a new copy of the array every iteration. This can cause slow performance and memory issues.	6
Multiple Array Copies [P]	When an array is forked and passed to multiple nodes, a new copy of the array is made-and large arrays forking multiple times can cause memory issues	5
No Wait in Loop [P]	If there is no wait inside of a loop, it could cause synchronization issues and also usually makes front panel elements unresponsive	4
Unconnected Front Panel Elements	When elements on the front panel are not connected to anything in the block diagram, this can lead to unexpected behavior or confusing VIs	3
Redundant Operations[P]	When there is large data, such as a large array, having operations that are redundant can waste time and memory in the VI (e.g., adding 0 to each element of an array)	3
No Queue Constraint[P]	When there are no constraints on a queue, the queue can grow infinitely large. This can cause performance problems due to memory and loop synchronization issues.	3
Infinite While Loop[P]	A loop that runs infinitely (sometimes due to the lack of any rule for termination) may cause issues with expected behavior and execution time.	3
Non-Reentrant subVI[P]	Non-reentrant subVIs have a data space shared between multiple calls. If a non-reentrant subVI is in a structure that can be parallelized, blocking causes poor execution speed.	2
String Concatenation in Loop[P]	When string concatenation occurs inside of a loop, it builds a new copy every iteration. This can cause slow performance and memory issues.	2
Sequence Structure[P]	Sequence structures remove a lot of the power of LabVIEW, limiting the compiler's optimization options and potentially reducing readability.	2
Deeply Nested Loops	Deeply nested loops reduce readability and make for confusing programs.	2
Only Loop Once	If a loop only iterates once, then having the loop structure reduces readability	2
Uninitialized Shift Registers [P]	If a shift register is not initialized then the value held in it will remain after the program as finished executing. This can result in unnecessary memory use, especially if the shift register holds an array.	T
Terminals in Structure [P]	Having terminals inside of a structure means that the UI thread gets blocked every time that structure executes. Removing terminals can decrease execution time.	T

There were a few other smells that were focused more on how a program looked such as wires going in the wrong direction or a general misuse of dataflow. These are discussed briefly after the performance smells below Of the performance code smells, only three, ***Build Array in Loop***, ***No Wait in Loop***, and ***String Con-***

catenation in Loop, can be detected by the existing VI Analyzer tool (Section 2.1.4).

4.2 Performance Smells

The following subsections discuss the performance smells that were generated from the AE Specialists as well as the two from training materials provided by National Instruments.

4.2.1 Too Many Variables

This smell was mentioned by all but one of the AE Specialists as a coding pattern that often causes problems in end-user code. One participant suggested that the frequency of this problem is often due to users having experience with textual languages where variables are an important part of the program. In LabVIEW, having too many variables often cause problems with the dataflow aspect of the language, and can lead to race conditions, particularly when one variable is written to in parallel locations. These can lead to unintended behavior and often confuse the user. This is especially problematic since LabVIEW is an inherently parallel language, and a variable could be set or read in many places at nearly the same time. In addition, the participants mentioned that having too many variables can make the program confusing to users, as it makes it unclear where the data is coming from and what is being calculated.

The exact number for too many variables varied amongst the AE Specialists, with

one stating that anything more than two was often problematic, but the consensus was that if there were more than four that was often emblematic of programs that had problems. Several participants stated that if there are two or more writes to a variable then it should be flagged as an issue as this is the first sign of a potential race condition.

4.2.2 Build Array in Loop

This smell was cited as one that caused slow performance and excess memory usage. AE specialists said that this smell often occurs because, to novices, it seems like the simplest way to fill an array inside of a loop. However, the **`Build Array`** node has the side effect of allocating memory to a new array every time an element is added to an array that is full. Every time the loop iterates it has the potential to allocate more memory to the array, which can cause the program to slow down. Overall, it results in unnecessary memory usage. An example of this is shown in Figure 4.1.

In this figure, the case statement contains three **`Build Array`** nodes. Every time this loop executes, these nodes will append a value onto the passed array. If the array runs out of space then it will have to allocate more memory. Since there are three of these nodes, this can be especially problematic for the performance of this program.

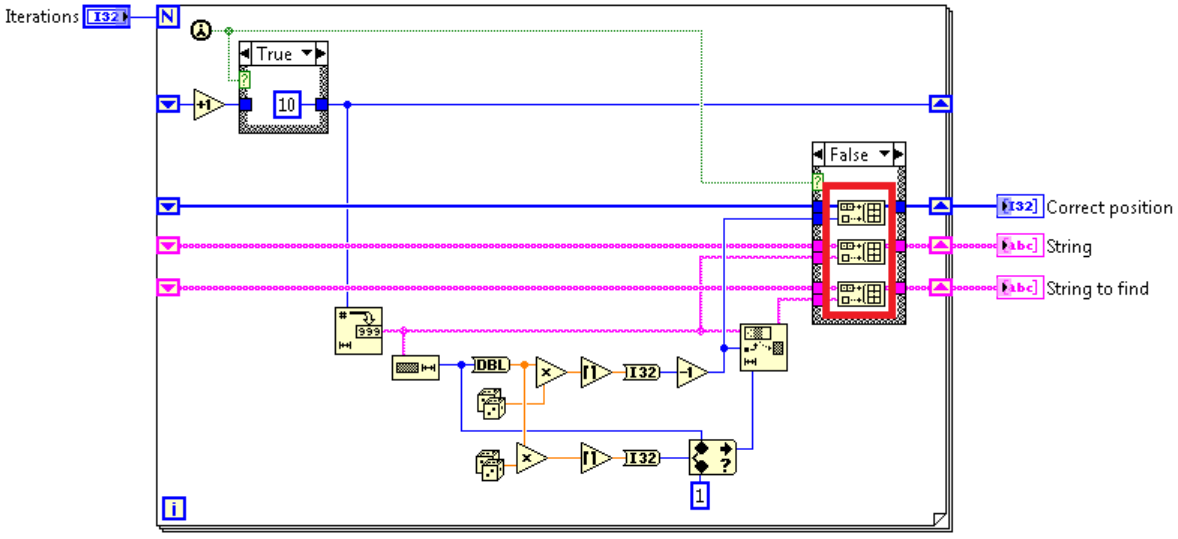

Figure 4.1: LabVIEW program showing three instances of Build Array in Loop (highlighted)

4.2.3 Multiple Array Copies

This smell occurs when the wire holding an array is split and then manipulated on both of the branches. When this is done, two copies of the array are created. This is especially problematic if the split occurs inside of a loop or if the array is especially large. An example of this smell is shown in Figure 4.2.

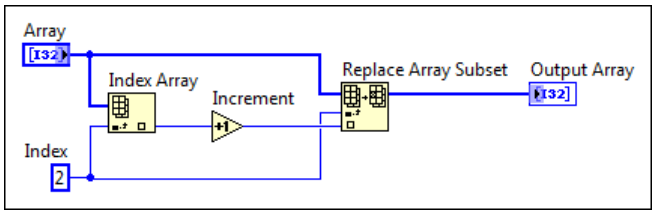

Figure 4.2: LabVIEW example demonstrating when multiple array copies are created (The wire split leading to the Index Array Node and the Replace Array Subset Node)

In this figure, the third element in an array of integers is accessed and then incremented. That value is then placed back into the array to replace the previous value contained in the third element. In this case, when the array is indexed to access the third element, a new copy of the array will be created and stored in memory.

4.2.4 No Wait in Loop

This smell is the result of having no timing nodes inside of a loop. Not having any timing can result in the program spiking the CPU (since the loop runs as fast as possible without waiting on each iteration) and taking as many resources as possible, which starves the other threads of CPU time. This can make front panels unresponsive and can make other elements of the program, particularly parallel loops, slow to execute.

One participant noted that this smell is particularly difficult for many users because they do not routinely think about timing while programming and thus leave it out. In addition, the reasons behind the use of timing structures such as a **`Wait`** node, are not explained by training materials in any detail, leaving users ignorant to the problems posed by having no timing in their loops.

Figure 4.3 shows a small LabVIEW program containing a **`While Loop`** that has no timing structures in it. In this case it is especially problematic because every loop iteration is contacting hardware and requesting data.

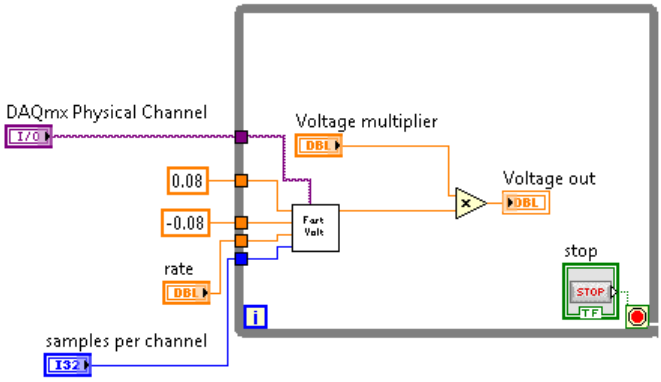

Figure 4.3: LabVIEW program containing a While Loop with no Wait Node

4.2.5 Redundant Operations on Large Data

This smell is similar to ***Multiple Array Copies***, but it involves a program performing unnecessary operations on data. This is especially problematic when the dataset is large, such as an array. The participants had one primary example that they thought could be checked for which was to look for unnecessary math. In these cases, the program is performing mathematical operations that do not change the dataset. One example of this would be a set of nodes that adds zero to every element of an array. This adds nothing to the program other than excess time and causes the program to run slower than necessary. A similar example is multiplying every element in an array by one.

In addition, the participants mentioned several cases where they had seen users write code that was very inefficient and did many more operations that were necessary. However, they could not boil this down to a specific smell, and fixing these instances often required refactoring of the entire program.

4.2.6 No Queue Constraint

Having no constraint on a queue will cause a problem when many elements are being added to a queue while very few are being removed. When this occurs on a queue that has no size constraint, the size of the queue will continue to grow and use excess memory. This problem can be remedied by adding a size constraint on the queue and specifying what happens when that limit is hit.

4.2.7 Infinite While Loop

While there can be cases when a loop should run continuously for an extended period of time, many users will create loops with no way to stop them, other than aborting the execution of the program. A loop that runs infinitely may sometimes cause issues with expected behavior and execution time. This primarily occurs when the user is expecting something to happen after the loop is finished executing. However, since the loop is infinite and will never end, they never see the expected behavior from their program.

Figure 4.4 shows a LabVIEW program that contains this smell. The cause of the smell is highlighted. In this instance the user has attached a constant Boolean value of False to the parameter that determines when the **`While Loop`** should stop.

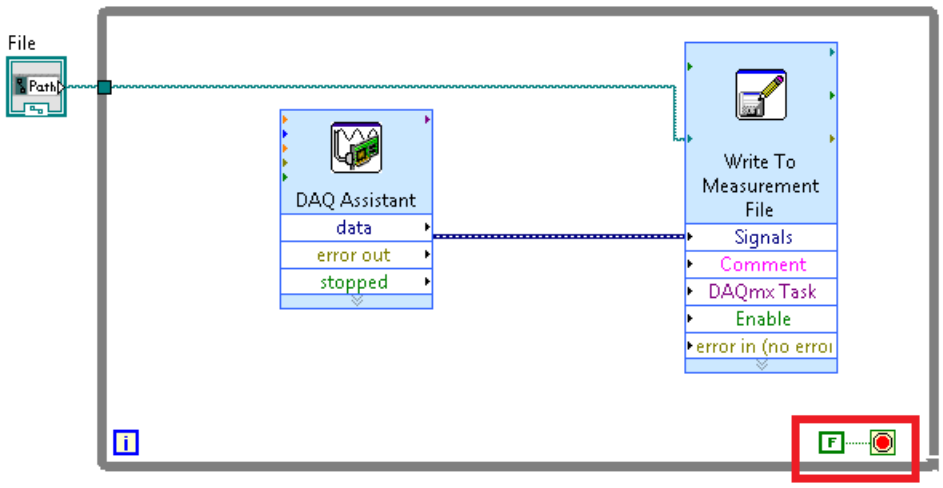

Figure 4.4: LabVIEW program containing an Infinite While Loop (smell is highlighted)

4.2.8 Non-Reentrant subVIs

When a subVI is created in LabVIEW, by default it is configured to be non-reentrant, meaning that only one instance of that subVI can run at a time. Basically, the subVI reserves only a single space in memory to store its data, so all instances of the subVI share that data space. However, if the subVI does not store data, then it can be configured for reentrant execution. Under this configuration, LabVIEW can execute all instances of the subVI simultaneously as LabVIEW allocates copies of the data space so that separate instances of the subVI have distinct locations to store their data.

Since LabVIEW is inherently parallel, if subVIs are not configured correctly they can cause issues with execution time and expected behavior. As an example, imagine there is a subVI that performs calculations on a number and returns the changed

value. Since this subVI is not storing data it can be configured for reentrant execution. If it is used only once in a program then there are no consequences. However, if this subVI is used multiple times in parallel loops, while still being configured as non-reentrant, it will slow the program down. When the instance of the subVI in one loop is running, no other instance can run, meaning that the other loop has to wait to execute the subVI, slowing the program down. Simply changing the configuration of the subVI to reentrant execution allows each instance to be run separately and the parallel loops to execute without a slowdown.

4.2.9 String Concatenation in Loop

In LabVIEW, strings behave very similarly to arrays, and the **`String Concatenation`** node is very similar to the **`Build Array`** node. When string concatenation occurs inside of a loop, it builds a new copy every time the loop iterates. Several participants mentioned that this was not as large of an impact when compared to ***Build Array in Loop***, but there are certain cases when this can cause poor performance, particularly when large strings are being concatenated together. This can eventually cause the same memory issues as a Build Array node.

Figure 4.5 shows a snippet of LabVIEW code containing this smell. In this example there are two **`String Concatenation`** nodes which are both highlighted. The node on the left is actually a false positive as the string that it creates will be removed from memory after every iteration since it is not connected to a shift register. The right one, however, will gradually accumulate memory for as long as execution

continues, without bound.

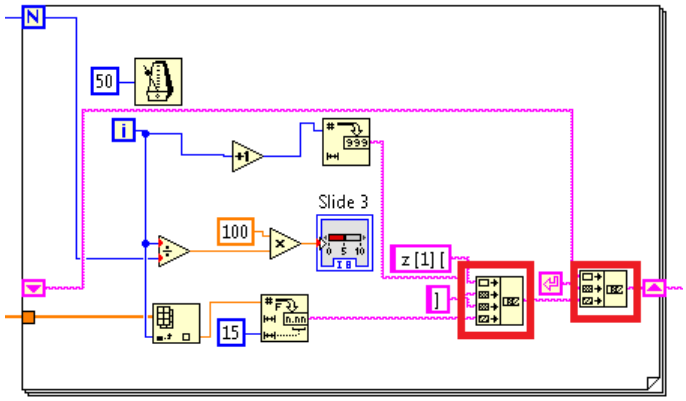

Figure 4.5: LabVIEW snippet containing two string concatenation nodes in a loop. Both nodes are highlighted

4.2.10 Sequence structure

The use of a **`Sequence Structure`** is often due to a user's desire to write LabVIEW code that is similar to how textual code is written, as sequence structures force a program to perform actions sequentially. This removes the inherent parallelism of LabVIEW. While there are cases where sequence structures make sense, several participants mentioned that gratuitous use lends to less parallelism and poorer performance.

Figure 4.6 shows a LabVIEW program containing a sequence structure. There is data being passed between the frames, and without a **`Sequence Structure`**, the frames would all operate in parallel.

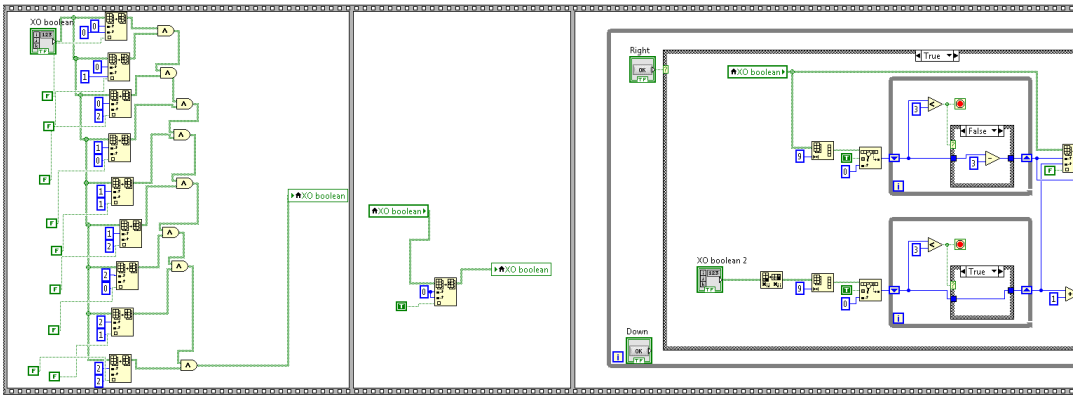

Figure 4.6: LabVIEW snippet containing a sequence structure. Only the first three frames are shown

4.2.11 Uninitialized Shift Registers

In LabVIEW, a shift register exists to help pass data between loop iterations. They can hold many types of data including, single values, arrays, and even clusters. Storing data in shift registers allow users to easily access and modify it inside of a loop.

Figure 4.7 shows a small example program containing a shift register, the nodes with the arrows on the left and right border of the for loop. In this example, the **`Shift Register`** begins with the value zero as that is what it is initialized to, it is then incremented in every iteration of the loop. Since this loop will iterate five times, the final value that shows up in the Numeric terminal is five.

This simple example shows the basic concept of a **`Shift Register`**, and also introduces initialization. As mentioned above, the **`Shift Register`** is initialized to a value when an element is wired to it. In the case above, that element was a constant integer with the value 0. This initialization sets the value of the **`Shift Register`** when the program starts to run.

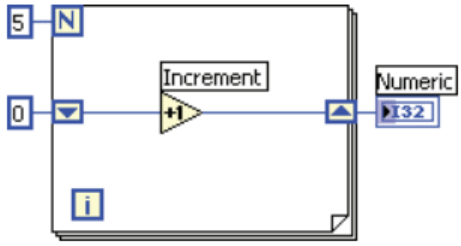

Figure 4.7: LabVIEW Example containing an initialized Shift Register (arrowed nodes on the side of the loop

If it is not initialized then it will keep the value stored in it from the previous run. For example, if the **`Shift Register`** in Figure 4.7 was not initialized then after one execution of the program it would hold the value 5 and would not be reset. If the program was run again, the final value would be 10. A third run would result in the value of 15, and so on.

While this example only contains a single value, if an uninitialized shift register holds an array, it can quickly increase in size over multiple executions, causing the program to use up more resources and the user to notice performance problems. If the shift register in Figure 4.7 was holding an array, each execution of the program would result in five additional elements being added to the array. This can quickly add up and result in unnecessary memory usage and slow programs.

4.2.12 Terminals Inside Structures

In LabVIEW, terminals refer to both controls and indicators, which are used for input and output respectively. Controls allow users to pass data to their programs

and change that data during execution, while indicators display data generated by the program. Controls can be objects such as buttons, sliders, or doubles, while indicators include objects like graphs, arrays, or gauges.

When a terminal is inside of a structure in a LabVIEW program, it causes the UI thread to be blocked frequently. This can be especially problematic when there are many terminals, and can cause a significant impact on the overall performance. The primary reason this smell causes poor performance is when a terminal, such as an array or a graph, is located inside of a loop. In these instances, every time the loop iterates the front panel will have to be updated to show the new values of the terminal. Moving these elements outside of the loop remove the ability to see them update live but can greatly increase the execution speed of the program.

One thing that should be noted about this smell is that to fix it entirely, all of the terminals have to be removed from the structure. So if there are cases where not all of the terminals can be moved, then the performance problems will not be completely fixed. There are cases were the terminals cannot be moved outside of the structure, mainly if they are required for algorithmic reasons. However, moving the worst offending terminals, such as graphs and charts could provide significant improvement.

Figure 4.8 shows an LabVIEW program with terminals inside of a structure, in this case a **For Loop**. The terminals are all individually highlighted. All of these terminals are outputs and represent two arrays and a graph. Terminals such as these are problematic in loops because they require the front panel to be updated every iteration to display the new values.

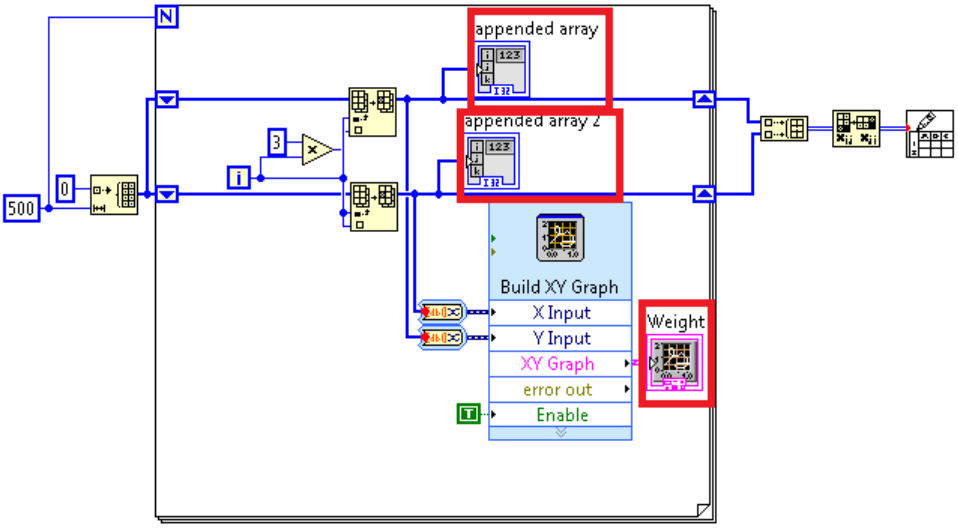

Figure 4.8: LabVIEW program containing three terminals inside of a loop (highlighted)

4.2.13 Smells unlikely to affect performance

The AE Specialists listed three smells that may have a minor impact on overall performance, but are more likely to impact program comprehension or cause the program to behave erratically. In a sense , they are more like the classic maintainability smells handled by many existing tools (Chapter 2). Because they are not relevant to the topic of this thesis, i.e. performance, these smells are not caught by SDPA and are only discussed here briefly.

The first smell is deeply nested loops, which tend to obscure the control flow of a program. This can make it difficult to diagnose performance problems, or to understand what the algorithm is designed to do. Deeply nested loops can sometimes be replaced with a simpler more efficient algorithm. Programs that contain this smell

often exhibit poor performance, but it is mostly due to poor algorithm design.

The second of these smell occurs when elements on the front panel are not connected to anything on the block diagram. This can cause a minor hit in performance because of unnecessary elements, but it is primarily a problem because the program fails to display data that the user is expecting.

The final smell is a loop that only iterates one time. This unnecessary structure creates clutter in the program and can reduce program readability. This is often a leftover remnant from a previous program that the user simply did not remove during code maintenance.

4.3 Smell-Driven Performance Analysis (SDPA)

The formative study revealed that many performance problems do result from end-user programmers' code. Below, are the technical implementation details of the LabVIEW prototype for finding such structures, as well as the design leading to this tool.

4.3.1 Paper Prototyping

To design a new kind of tool for detecting performance smells in graphical programs, an informal paper prototype study was run with a four LabVIEW users to discuss preliminary tool designs and to obtain feedback useful for refining the designs. In addition to gathering information about the types of interactions that were desired,

this study helped solidify the design of the tool, and it guided the creation of the actual prototype.

4.3.1.1 Participants and Methodology

To recruit participants for this study, LabVIEW programmers from Oregon State were contacted via email, focussing on professors and students of LabVIEW in classes at Oregon State. In total, four participants were interviewed: two professors, one staff member who used LabVIEW professional in a lab setting, and one graduate student who used LabVIEW for research purposes.

During a 30 minute one-on-one interview session, the participants were shown a LabVIEW program on a piece of paper. They were then told there they was a performance problem with the program, and the participant had to debug that issue. The participants were then showed several possible interface designs to determine what they thought worked best or what if anything they found annoying. The user could interact with these potential designs, and the researcher used the paper prototype to describe how each tool would respond. The participants' opinions were then gathered along with any potential ideas of there own. The participants' views on the feedback management, particularly Interruption Style and Criticism Type, were of particular interest to determine an effective way to give users feedback about a program.

4.3.1.2 Results

Through the process of showing the participants several potential designs, it was quickly apparent that they would really like to avoid any tool that dominated too much of the screen or too much of their attention. While they thought that it would be useful to have a tool that could notify them of potential performance problems, they did not want the tool's information to dominate the programming window. They felt like that space was already limited, and crowding it with a lot of extra icons or information would be very problematic.

Consequently, of the potential designs that were presented, most participants far preferred the ones that had small icons on the nodes indicating that there was a specific problem on a node. In terms of the potential designs, the users were split about how they wanted to be notified of a problem. Two users thought that having an icon show up in the IDE somewhere would be good, while the other two did not want any icon to show up unless they initiated the tool themselves. One of the tested methods involved highlighting each affected node with a squiggly red line or box similar to how a spell checker might work. None participants liked this approach as they felt it was too distracting.

The participants were also asked how they would prefer the see descriptions of problem details. There was no uniform answer, but they did generally say that it would be nice if the results from the tool could be displayed in a similar manner to current LabVIEW errors and warnings. In addition, they liked the idea of having this information within the LabVIEW IDE. In contrast, VI Analyzer and other "add-on"

LabVIEW tools are not part of the IDE requiring users to flip their attention back and forth between separate windows.

The interruption style that was preferred by all users was negotiated-style interruptions. This style uses interruptions that inform the user of a pending message but do not force them to acknowledge it immediately [62]. This method was also used in the Surprise-Explain-Reward research [96] and have been shown to make users more effective at debugging [81].

In terms of the criticism type, users thought that showing both bad (focusing on areas that have problems) and good (focusing on patterns that follow good performance) areas of code resulted in too much information that they had to sift through. They said only seeing feedback about the good ones was not helpful to the debugging process. By contrast, they felt that just seeing the bad information was the best form to aid in debugging.

Overall, the idea of a new kind of tool was well-received, and concrete guidance was obtained for how to design such a tool for reporting performance problems.

4.3.2 Basic SDPA Prototype

A prototype tool has been implemented to apply this technique on LabVIEW code. This tool is fully integrated with the version of the LabVIEW environment currently under development at National Instruments. This integration allowed easy evaluation of the effectiveness of this technique in a realistic end-user programming environment. Even though LabVIEW was used for this work, there is no reason why similar tools

could not be created for other visual dataflow languages, although presumably the specific heuristics needed would vary.

The prototype begins by acquiring a data structure representing the LabVIEW program at hand. This is done by taking advantage of the existing DFIR graph, which is LabVIEW's hierarchical, graph-based internal representation of a program's code [40]. The DFIR graph of a program is very similar to the program's visual representation on the screen, except (1) the DFIR graph decomposes some program elements that appear as single icons in the visual program, and (2) the DFIR graph includes optimizations achieved by minor program transformations (e.g., dead code elimination). In all other ways, however, a DFIR graph is to LabVIEW what an abstract syntax tree is to Java or a similar textual language.

Next, the prototype runs bad performance smell detectors on the program. Each detector checks for one of the performance problems by implementing a heuristic for finding that problem. The detector does this by visiting each node of the DFIR graph relevant to a particular smell, then computing which nodes (if any) meet the heuristic. For example, our detector finds instances of ***Build Array in Loop*** smells by visiting all **`Build Array`** nodes in the DFIR graph, then for each **`Build Array`**, recursively working outward in the DFIR graph (analogous to working up an abstract syntax tree) to check if that **`Build Array`** is nested inside a **`Loop`**; in this case, the heuristic is simply checking whether a **`Build Array`** node is anywhere inside a **`Loop`**. Likewise, the ***Too Many Variables*** smell detector visits all variable-declaration nodes and counts to check if more than four are in any one VI.

The prototype alerts the user to detected smells by annotating program elements

with small triangle icons, as shown in Figure 4.9[1]. Since smells usually involve more than one node, there is the question of which nodes should receive these alert annotations. For the prototype an alert is place on each node in the program that an end-user programmer would need to modify to fix the smell; usually, this is one program element or a handful of elements. For example, when ***Build Array in Loop*** is detected, SDPA places an alert on the **`Build Array`** node inside of the loop to indicate that this node is causing the issue, however for ***No Wait in Loop***, an alert will be added to the **`loop`** that is missing the node.

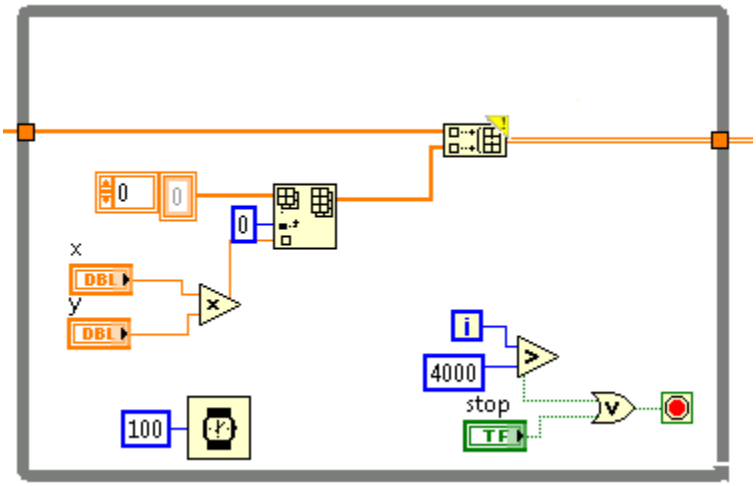

Figure 4.9: LabVIEW example containing an instance of Build Array in Loop that is marked with an icon to indicate that it has been detected

In addition to the alerts that appear on the visual code itself, the prototype also shows a tabular list of all smells found in the program at the user's request. This table succinctly explains each detected problem and its cause. Clicking on an alert icon in

[1]This figure is a mock up of the interface as the actual interface was implemented in an unreleased version of LabVIEW, kept confidential at NI's request.

the program causes the corresponding entry in the table to be highlighted (and vice versa), thereby supporting direct navigation between an alert and its explanation. If the user alters the code to fix a smell, the corresponding icon and table entry automatically disappear.

4.4 SDPA with Smell-Aided Profiling

While the tool described in the above section is useful for helping end-users detect performance problems, the information that it provides has the possibility of being a false positive.

For example, there may be a ***Build Array in Loop***, but if that loop only iterates a few times, it likely will have very little impact on the overall performance of the program. Similarly, a ***Multiple Array Copies*** that involves an array with only 5 elements that is copied only once through a wire split will have a very minor effect on performance. The basic SDPA prototype, above, does not have the intelligence to filter these cases out. This may lead to users seeing messages that have little to no effect on the performance and may cause mistrust in the tool.

One way to filter out false positives is to profile the program to determine which parts of the code take the longest to execute. The assumption is that the returned are the hot spots of the program and all debugging should be focused there. While profiling is a common technique done in performance debugging, adding profiling to a code can often change the execution and alter the timing enough that the correct hot spots may not always be discovered.

To address these challenges, the prototype also incorporates Smell-Aided Profiling, a new form of profiling to try and mitigate the effect of profiling the entire program. Since SDPA can detect smells that cause bad performance, it is likely that most performance problems will be contained in the areas affected by those smells. Smell-Aided Profiling treats these areas as potential hot spots and only profiles those areas. If one of those hot spots, which already contains a smell, is found to be poorly performing, then it is marked and the user is informed. By only profiling the areas that contain smells, the tool limits the adverse affects of profiling and is ideally able to determine the areas that are causing performance problems, while filtering out those that are not out of the tool results.

4.4.1 Profiling Smells

The first area of implementation that needed to be completed was to set up a way to profile each smell inside a LabVIEW program. To do this, the idea of timing probes was implemented. LabVIEW currently has the ability to create probes that track data over wires, and this method was modified so that CPU and memory usage could be tracked instead of the data. While probes are usually created manually by the users, the timing probes used for Smell-Aided Profiling are inserted automatically by the prototype tool.

As an example, take the program in Figure 4.10. It shows a **`For Loop`** in LabVIEW that contains a **`Build Array`** node inside it. As this program shows, the loops is only going to execute 3 times, meaning the impact of the overall performance

is likely to be minimal. However, without the ability to profile the code, SDPA will still notify users that a ***Build Array in Loop*** smell has been detected.

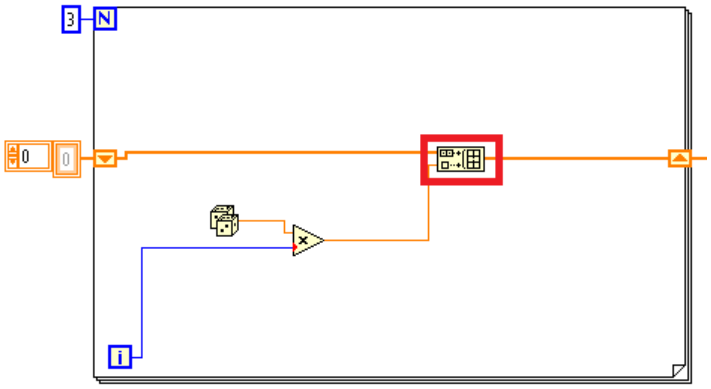

Figure 4.10: LabVIEW program showing a Build Array node in a Loop (highlighted in red). Due to loop iterations this smell is unlikely to cause a performance problem.

When the tool detects a smell, it automatically create two timing probes. The first probe is created on the wire leading into the smell and the second is created on the wire leading out of the smell. If a program is run with Smell-Aided Profiling turned on, it will take a snapshot of current memory usage as well as mark the current time when the first timing probe is hit (during execution). When the second timing probe is hit, it takes another snapshot of the memory usage and the current time. It then subtracts the memory usage and time that was gathered by the first timing probe from the data of the second. This gives the change in memory as well as the total execution time of the area containing the smell.

Consider again the program in Figure 4.10. When it is detected by the tool, timing probes will be dropped on the wire leading into the loop and the wire that is leaving the loop. When the program is executed, it can now collect data on the smell and

determine if it is causing performance issues in the program. If the timing probes are hit several times, that is, if the smell is inside a loop, the data collected will be summed for each run to gather the overall impact.

Since each smell manifests differently in a LabVIEW program, the timing probe configuration varies for certain smells. Table 4.2 details where each timing probe is set up for a specific smell.

Table 4.2: Timing probe wire locations for specific smells

Code Smell(s)	**Begin Timing Probe**	**End Timing Probe**
Build Array in Loop No Wait in Loop String Concatenation in Loop Infinite While Loop Uninitialized Shift Registers	Wire into Loop	Wire out of loop
Multiple Array Copies	Wire before split	Wire after split
Non-Reentrant subVIs	Wire into subVI	Wire out of subVI
Redundant Operations	Wire into operation	Wire out of operation
Sequence Structure	First wire in first frame	Last wire in last frame
Terminals in Structure	Wire into structure	Wire out of structure
No Queue Constraint	Wire into Enqueue	Wire out of Enqueue

There is one smell for which timing probes cannot collect adequate information to diagnose if they are a true positive or a false positive. For ***Too Many Variables***, which manifests over the entire program and is not relegated to a single spot that is profitable, no profiling is used to filter alerts.

With the timing probes in place, an initial snapshot of the program will be taken, providing the current level of memory use and the start time. This allows Smell-Aided profiling to determine the total memory usage and execution of the program by comparing it to a similar snapshot at the end of execution. During execution, if a

timing probe is hit a snapshot of memory is taken and the current time is recorded. When the corresponding ending timing probe is hit, another snapshot will be taken gathering the new memory information and current time. The tool compare the values to determine the change in memory usage and the execution time of the profiled area. If the timing probes are hit multiple times then the information for each execution will be averaged to see if the memory usage or execution time increases for each specific run.

The results will then be filtered based on which smells have been determined to be actual problems and which do not meet the requirements to counted as a smell. How this is done is discussed in the next section.

4.4.2 Determining False Positives

The final step of the profiler is use the timing and memory information gathered from the timing probes to determine which smells are actually causing performance problems and which detected smells are false positives. This will allow the tool to filter the smells shown to users.

The algorithm to determine if a specific smell is a false positive is as follows:

1. Calculate Percent of Total Time Used (TP) by Smell Time/Program Time
2. Calculate Percent of Total Memory Used (MP) by Smell Memory/Program Memory
3. Calculate Percent of Total Nodes (NP) (Smell Nodes/Program Nodes)
4. If TP - NP $< 15\%$ AND If MP - NP $< 10\%$ then smell is a false positive

Digging into this algorithm, it first must calculate the time and memory used compared to the total time of the program. For example, if the time measured for a given smell is 50 ms, but the total time of the program is 5000 ms then the smell has very little impact on the overall performance. Similarly, if the total time is only 100 ms, then the smell is the cause of 50% of the total time.

While it seems likely that such a smell would be a true positive, there is one further check that must be completed. That is to determine what percent of the program the smell takes up. A smell instance that occupies a large fraction of a program's code should be entitled to use a large fraction of the program's memory and execution time. This is calculated, imperfectly, using the number of nodes that are present in the area that was identified for a smell. If the smell was in a structure, such as a loop, then all of the nodes in that loop are counted. Take the example in the paragraph above. Suppose that the smell that caused 50% of the total time also had 50% of the total nodes in the program. If that is the case, then it is not surprising for it to take that much time to execute.

The thresholds (15% and 10%) for determining if a smell was a false positive was arrived at by running the tool, on programs gathered from the LabVIEW forums, with

certain values to determine the best possible fit. Values higher than this removed too many smells, while numbers smaller than this filtered very few smells.

4.4.3 Interface Design

The final touch needed to complete the profiling enhancement to the tool is to add a way for the user to apply profiling. The ability to profile for performance problems should be tied closely to the user interface where detected smells are presented. In addition, how end-users use LabVIEW as a whole should not change, meaning that how they run a program should not change despite the fact that they want to apply Smell-Aided Profiling.

To determine the final interface, several designs were developed to imagine potential methods for changing the interface to suit the needs of the tool. The final interface change that was decided on was ultimately very simple. As detailed in Section 4.3.2 above, there are two places where end-users can view information about the smells, a table at the bottom that contains information about all of the detected smells in the program and a panel on the far right of the IDE that shows information only on a smell that has been highlighted by the user. In both of these places, a check box was added, that allows users to turn on performance profiling upon the next run of the program.

When a user mouses over these check boxes, a message is displayed that details what this check box would do. If it is checked, when the program is next run, by clicking on the green arrow, profiling will be turned on and the necessary data will

be collected.

The final change to the interface is in the information that is displayed. When profiling is turned on, the smells table at the bottom will be filtered so that only smells that are determined to be true positives are displayed. In addition, the smells in the table will be sorted so that the smells that have been profiled to have the highest impact on performance are displayed at the top.

In addition, the information that is given in the right panel is changed to include specific profiling information about the selected smell. It tells how much memory was used throughout the course of execution, as well as the execution time of the timing probes around the smell. Finally, it provides information about what percentage of the total memory usage and execution time were taken up by the selected smell. Figure 4.11 shows this. In this image, the **`For Loop`** was detected as a hot spot. The user can click on the hot spot entry in the table at the bottom of the screen. The relevant area is then highlighted in the code and information detailing the hot spot, including the smell inside of the hot spot, is discussed in the pane on the right side of the IDE. (This figure is a mock up of the interface as the actual interface was implemented in an unreleased version of LabVIEW, kept confidential at NI's request.)

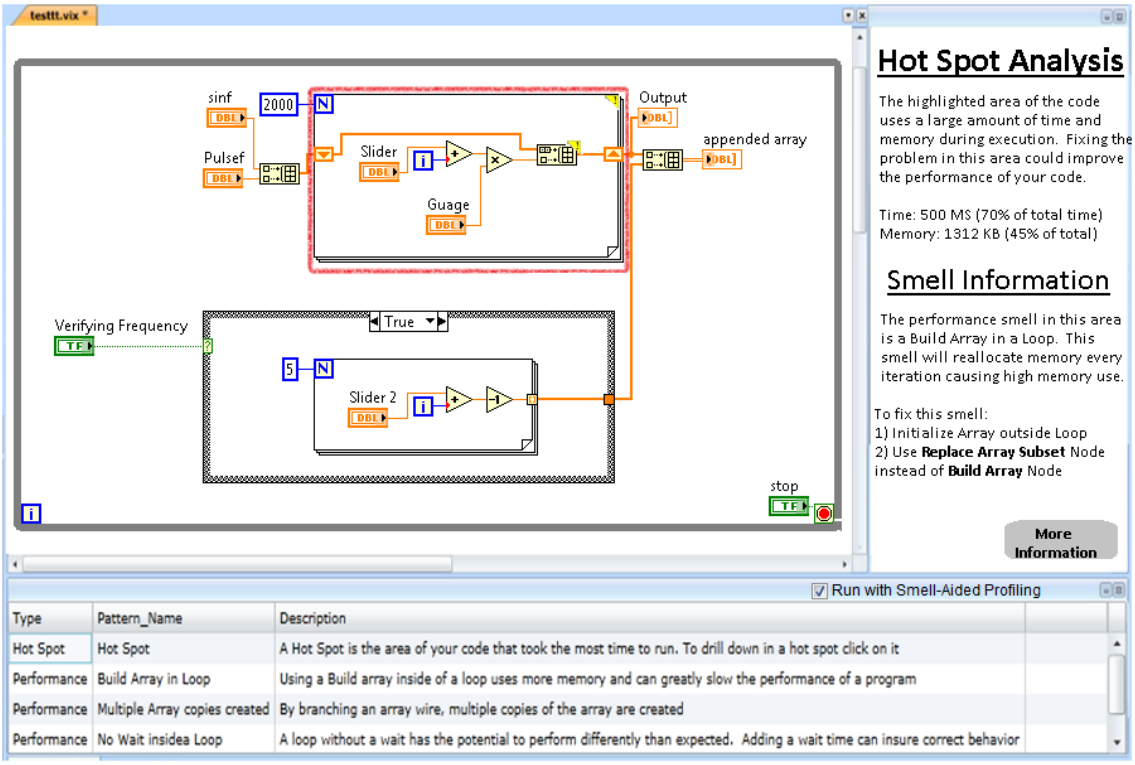

Figure 4.11: The SDPA user interface after Smell-Aided Profiling was run. The hotspot is highlight and information provided to user on the right side of the IDE

Chapter 5: Smell Guided Transformations

A user debugging a program has two primary goals. The first is to find the problem. Once that has been accomplished, the next goal is to determine how best to change the code to solve the identified problem.

Chapter 4 detailed SDPA and the Smell-Aided Profiling extension that help end-users find performance problems in their LabVIEW programs. These methods also provided basic advice on how a user could solve that problem. But in a user study (Section 6.2), it was found that this information was not always enough for the users to determine a solution. A few participants in this study mentioned that it would be helpful to have more information about ways to fix the issue. Among the suggestions were to link to white papers or websites about the smells and the potential solutions.

While these suggestions might be useful to end-users, what they really reveal is a need for help with fixing problems by eliminating smells. This chapter describes an approach for meeting this need by providing end-users with the ability to apply fully automatic or interactive code transformations that fix the performance problems detected by SDPA.

5.1 Methodology for Determining Smell Transformations

Before the transformations could be implemented for LabVIEW code, a set of transformations had to be determined for every smell detected by the tool. To accomplish this, AE Specialists from National Instruments were interviewed about how they would fix each smell. This was done to draw on the expertise of LabVIEW experts who routinely fix problems such as these.

To recruit participants, an email was sent to all of the AE Specialists at National Instruments describing the idea of smells and why transformations were required for the next stage of the prototype tool. Seven individuals agreed to participate in the interview process and were each scheduled for a half hour session.

During the interview, the participants were shown example programs for most of the detectable smells. These example programs were gathered from the National Instrument LabVIEW forums[1]. These programs were briefly tested to verify that they showed some performance problems. In some cases the programs were snippets that contained only the relevant part, but this was done in a limited fashion as often it is better to look at the whole program while debugging.

For each smell, the AE Specialists where asked to describe how they would fix the given program to solve the performance problem. After this information was gathered, they were asked in detail about their solution and why it would fix the problem. They were also asked if this solution would work for every instance of a smell, and if it did not work for a specific instance, to describe why, and a better

[1] http://forums.ni.com/t5/LabVIEW/bd-p/170

solution.

From these interviews, a set of transformations was created to handle the smells detailed in the previous chapter. In some cases, the participants could not determine a specific automatable transformation that would always work due to the difficulty of the problem. As such, three categories of transformations arose. The first, Full Transformations, address smells that can be completely fixed automatically by simply clicking on a button to apply a transformation. The second category, Partial Transformations address smells that can be applied fully with the exception of a few minor details that must be provided by the user. The final category, Wizard Transformations, address smells that do not have a fully automated fix but rather offer a wizard guiding the user to a solution. These categories will be talked about in detail in the following subsections.

Unlike refactoring, many of these transformations will change the semantics of the program. The problems that are detected by SDPA are code structures, and the problems that they cause cannot be fixed without changing the semantics of the program. In addition, the timing of when outputs occur from the program are also part of the semantics. Fixing performance issues will most likely change when a program outputs results and thus change the semantics.

5.2 Full Transformations

The first category of transformations contains those that fix smells that can be fixed semi-automatically with the click of a button. This is the simplest category of smells

since the user does not have to do much work to solve performance problems. The tool also provides information as to why this transformation will solve the problem in the hopes that the user will learn the cause of the problem and can avoid it in the future. The rest of this sub section will detail these transformations, as well as discuss what the AE Specialists said about each transformation.

5.2.1 No Wait in Loop

Having no timing structure inside of a loop can usually be solved with a rather simple fix. All of the AE Specialists said that what is needed is to add timing operation. For the majority of cases, this is as simple as adding a **Wait** to the loop. There are rare cases where a more detailed timing structure is required, but these examples had to do with complex programs such as those that would control an FPGA and would require specific timing requirements.

There are two primary types of waits in LabVIEW, but most of the AE Specialists preferred **Wait Until next ms Multiple**, since it is generally considered to have more precise timing, which is important for real time applications. For the examples that they were shown during the interviews, all stated that the type of wait that was added probably would not make much difference to the program and that most end-users would probably understand using the simple **Wait** node. A few of the participants mentioned that this could be one way to help instruct users of the differences between the two types of wait nodes by describing the cases when each should be used one this smell is detected.

The final step of determining the transformation was to decide on the best wait time. This is, of course, highly variable and depends on the loop itself as well as the computer that the program is running on. Several of the participants noted that they will often look at CPU usage and loop rate and determine the best time. Others stated that they usually pick a generic time, such as 10 ms or 100 ms and then modify that number if the program is not behaving as they would prefer. Finally, several mentioned that even adding a **`Wait`** with a wait time of 0 or 1 ms would cause the loop to pause and allow other structures or the UI Thread to be accessed. In the case of the programs that they were shown, they determined that the user may not want to wait a long time to see their results and that they would try a wait with a small number so that the program wouldn't run much longer than it did before the wait was added.

The final transformation for this smell is:

For a loop with no `Wait` node:

1. Add a **`Wait`** node to the loop
2. Add a integer constant with the value of 1 ms
3. Connect the constant to the **`Wait`**

Figure 5.1 shows the loop with the transformation for this smell applied. The transformation (adding a **`Wait`**) has been highlighted in red. The before program (Figure 4.3 can be found in Section 4.2.4.

In addition, the tool provides information (that was gathered from the AE Specialists) about why a **`Wait`** node should be inside of a loop and when someone might

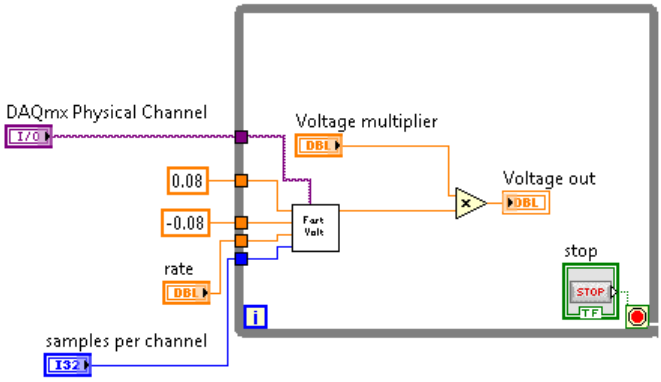

Figure 5.1: LabVIEW program showing the transformation for a While Loop with no Wait Node (transformation is highlighted)

want to use the different timing structures. It provides this information in the pane where users apply the transformation. This pane also describes that though the wait time of one millisecond is likely to be adequate to improve the performance, a longer wait may be preferable for some programs.

5.2.2 Sequence Structure

Using sequence structures effectively can get tricky, particularly if data is shared between frames in the sequence structure. If that is the case, then a **`Shift Register`** needs to be created on the loop to hold that information. This can result in many shift registers being created, which may make the code more confusing. When asked about this, AE Specialists mentioned that passing data between frames in a **`Sequence Structure`**, particularly a **`Stacked Sequence Structure`** is even more confusing. Figure 5.2 shows an example of a **`Stacked Sequence Structure`** with

all of the cases shown at once. Normally, the user would have to hit the arrow at the top of the structure to move between frames. The bottom of each frame contains feedback nodes that allow data to be transferred between the frames.

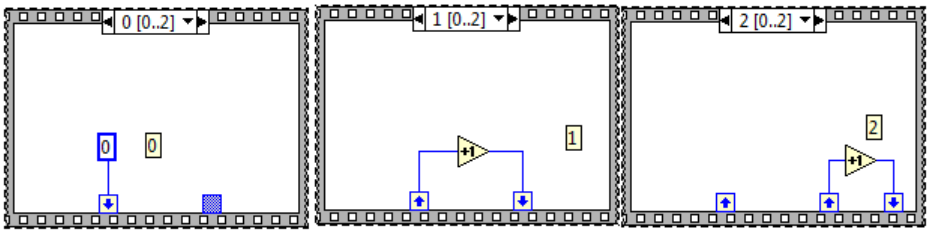

Figure 5.2: Stacked Sequence with each frame shown, feedback nodes (boxes at the bottom of each frame) pass data between frames

While this example contains only a few feedback nodes, several AE Specialists noted that with more complex uses, a **Stacked Sequence Structure** can have many of these nodes which can be very confusing. In some cases this can even cause data to be used incorrectly. For example, one participant had seen a program that was using more memory than expected and was slowing the execution speed to a crawl. After investigation, the AE Specialist discovered that the user was passing an array between frames using the feedback nodes incorrectly, which caused multiple copies of the array to be created every time the sequence structure was executed. Due to this, a few participants thought that replacing a stacked sequence structure might alleviate some of these issues as all of the data that is used in a state machine is visible at once, and could fix some of the performance problems that was caused by the incorrect usage of the feedback nodes.

In general, however, the AE Specialists who were interviewed were split about the

best transformation for the ***Sequence Structure*** smell, but most of them agreed that regardless of the transformation, replacing a **`sequence structure`** with anything was unlikely to change the performance very much for better or worse.

The majority did agree that users often add sequence structures to their programs because they are trying to replicate the behavior of a textual programming language. A **`Sequence Structure`** allows users to create programs that are guaranteed to operate in a specific order (frame by frame). Users who are learning LabVIEW can sometimes be unsure about when certain parts of the program will execute because they do not fully understand the dataflow nature of the language.

The transformation arrived at was to replace the **`Sequence Structure`** with a state machine, which in the case of LabVIEW can be created using a **`For Loop`** with a **`Case Structure`** inside of it. Each element in the **`Case Structure`** would contain one frame from the sequence and would be accessed by the loop count. Some participants were skeptical that this would be of much help, but they did not have a better solution.

One item of importance that was discussed is that users will often use a **`Sequence Structure`** to perform timing of a snippet or program. This generally takes the form of a three frame sequence structure in which the first frame will set up the timing, the second will contain the code the user wants to be timed, and the final frame will end the timing and display how long the code took to execute. Many of the participants also mentioned that sequence structures that are less than three frames don't really make sense to transform since they are so small anyway. For these reasons, the transformation will only be applied to sequence structures containing more than

three frames.

The complete transformation is:

For a sequence structure with X frames (X >3):

1. Create a for loop that executes X times
2. Create a case structure inside the loop
3. Wire the loop counter to the case structure
4. Copy the code in frame one of the sequence structure to the first element of the case structure
5. For any wires going from frame n to frame n+1, create a shift register on the loop
6. Connect wires to shift registers
7. Copy frame n+1 to the case structure
8. Connect relevant shift registers to elements in the case structure for frame n+1
9. Repeat steps 4-8 for the remaining frames in the sequence structure

Figure 5.3 shows the example program from Figure 4.5 (Section 4.2.10 after the transformation, with the sequence structure being replaced by a state machine. This figure has numbers corresponding to the steps of the transformation. The (1.) is located next to the for loop that is added in step 1, (2.) by the case, and (3) is next to the loop iteration count that is connected to the case structure. Steps 4-8 involved transferring the specific frames. In this Figure, frame 3 is shown, but clicking through the case structure would show the other frames.

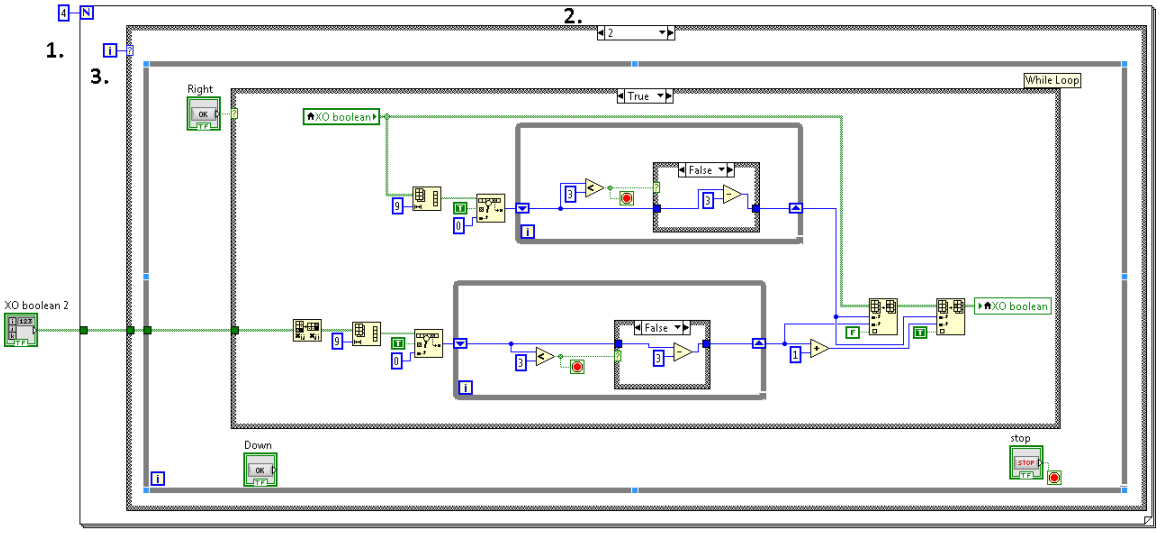

Figure 5.3: Sequence Structure after being transformed. The numbers mark the steps of the transformation.

5.2.3 Infinite While Loop

The problem with having an infinite **`While Loop`** inside of a program is that often the user expects the program to complete, but the loop will continue running. LabVIEW offers the ability to abort a program while it is running; however, this often causes problems with the behavior of the program. For example, if the program is collecting data inside the loop, it will not be saved when the program is aborted. Basically, the abort option simply ends the program wherever it is in the execution, which can be problematic, particularly for programs that are collecting data to be stored in a file after the loop is completed.

The participants in the interviews all came up with the same simple fix for this problem, which was to add a button the user could click on to end the loop. When the

button is clicked, the program will finish the execution of the current loop iteration and then exit the loop allowing the data inside of the loop to be saved and the rest of the program to finish executing. The complete transformation is below.

For a `While Loop`:

1. Create a button inside of the loop
2. Change the loop ending parameter to stop if true
3. Delete constant parameter
4. Connect the button to the parameter

Figure 5.4 demonstrates the above transformation. In this instance, the button has been created and wired to the loop ending parameter (highlighted in red). An image of the before program (Figure 4.4 can be in Section 4.2.7).

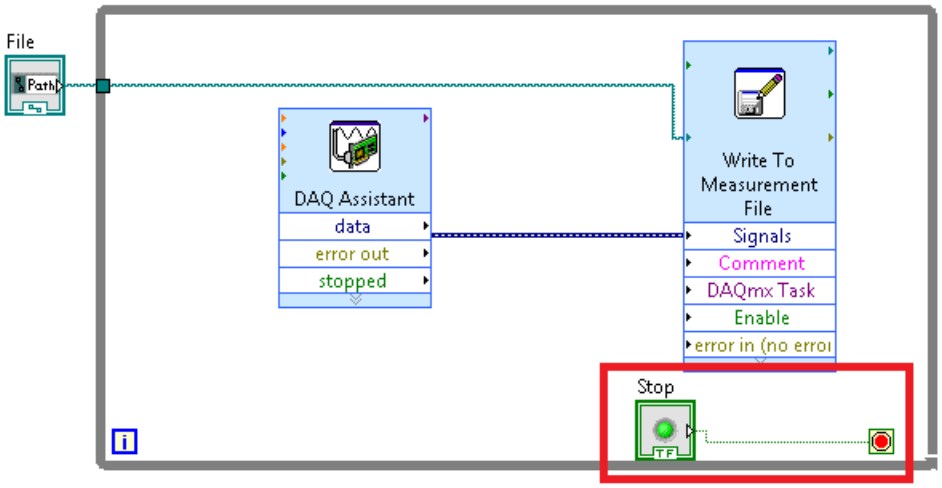

Figure 5.4: Transformed LabVIEW program no longer containing an Infinite While Loop (transformation is highlighted)

A few AE Specialists mentioned that there may be times when a more complex solution is needed to an ***Infinite While Loop***. Nonetheless, adding a stop button to the loop is the first thing that they would do, and many mentioned that the more complex solutions are dependant on algorithmic choices by the user. For example, a user may want a loop to run for a specific amount of time or until a specific amount of data has been gathered. Simply adding a stop button will not accomplish this.

5.2.4 Uninitialized Shift Registers

An uninitialized **`Shift Register`** causes problems because it causes a program to store data over multiple runs, which unnecessarily wastes memory. When this value is an array, that often means that an array can quickly grow unwieldy over several runs of the program. The correct way is to add an constant in front of the loop and connect it to the shift register. This will cause the value that is being held by this shift register to start fresh at each run of the program. An example of a shift register that is uninitialized and one that is initialized is shown below in Figure 5.5.

In this example, the **`Shift Registers`** are the nodes on the border of the loop with the down array. The **`Shift Register`** in the middle is initialized as it has a constant array wired to it, but the other three are uninitialized as they have nothing wired to them. There are several ways that the values for the shift register could be initialized, such as using an **`Initialize Array`** node to create a array that is the correct size for the algorithm, but these fixes may not also be correct for a given case.

The AE Specialists mentioned that this smell only causes performance problems

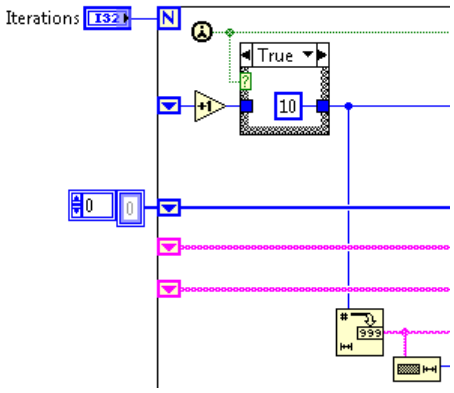

Figure 5.5: LabVIEW example showing initialized (second blue shift arrow node) and uninitialized (the remaining three) shift registers

if the **`Shift Registers`** are storing arrays. When an uninitialized shift register stores an array, it keeps all of the values that were added to it in the previous runs. However, if the value is a double, then it will only have a single value and uses the same amount of memory regardless (though it could cause the program to behave differently). Since SDPA marks all uninitialized shift registers, the transformation is created to repair all such cases.

This transformation will retain the behavior of a given program on its first execution, but in the cases when the user wanted the data to be kept over multiple runs, it should not be applied. This information, as well as why it can cause problems is supplied to the user by the tool.

For an uninitialized shift register:

1. Determine the type of the shift register.
2. Create a constant of that type outside the loop
3. Connect the constant to the shift register

Figure 5.6 shows the transformation applied all of the shift registers in Figure 5.5.

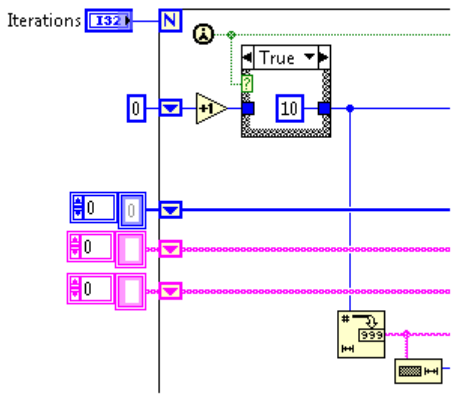

Figure 5.6: Transformed LabVIEW example showing all four shift registers initialized

5.2.4.1 Redundant Operations on Large Data

A redundant operation has a very simple transformation, which is to remove the redundant operation from the program. It is possible that some of these redundant operations might be necessary, but for the most part the participants mentioned that they are likely leftover code from something that was no longer needed. The primary problem is that the tool can only fix the redundant operations that can be

detected. In the prototype that was developed and described in the previous chapter, the range of redundant operations that was caught is relatively narrow. In all cases, the transformation is to take the redundant operation and remove it. Once this is completed, the wires must be connected so that the program can be executed.

5.3 Partial Transformations

Partial transformations are transfers that can mostly applied semi-automatically, but require the assistance of the user to complete them in some cases. In these instances, the user will be prompted for information that will be used in the transformation.

5.3.1 Build Array in Loop

The ***Build Array in Loop*** smell is often used as the primary example of a bad smell in LabVIEW, and the performance impact is usually fairly noticeable. However, the AE Specialists that were interviewed did not have a consensus about the correct solution for this smell. In fact, out of the interviews three possible solutions arose.

The first possible solution was to replace the **`Build Array`** node in the loop with array indexing, which is a LabVIEW construct that can be connected to loops that allows an array to be build every iteration of the loop. If an element is wired to the array indexing construct, it will be added to the ith element of the array, where i is the current loop iteration. This was a popular answer among AE Specialists, but upon testing was discovered not to improve the performance of the code at all.

The second solution was to wrap the **Build Array** with an **In-place Structure**. In LabVIEW this structure forces everything inside of it to be done in-place, meaning that no new copies of anything will be created. The participants who mentioned this solution were trying to mitigate the fact that the **Build Array** node inside of a **Loop** will use a lot of memory. This solution was moderately successful, and it could be implemented as a full transformation; however, the performance improvement was much less compared to the final option.

This final solution was to initialize the array outside of the loop. Then, inside of the loop, change the **Build Array** node to a **Replace Array Subset** node. This creates an array that is already as large as the user requires, and inside the loop the value at element i is replaced with the value that is generated inside the loop. This transformation cuts down on both the memory usage and the execution time, and of the three proposed solutions was clearly the best when tested on a handful of example programs.

However, the transformation is different depending on the loop that is part of the smell, thus necessitating user input. If the **Build Array** node is inside of a **For Loop**, then the transformation can be fully automated. The transformation will set up an array initialization node outside of the loop and the array size set to the number of loop iterations. In the case of a **While Loop**, the initialization cannot be fully automated because the number of loop iterations cannot be determined algorithmically. In this case, the transformation will be applied, but the user will be informed that the size of the array should be changed to reflect how many times the **While Loop** will iterate. The AE Specialists said that the default for an array in a while

loop should be to set the array size to an arbitrarily large number so that the array could continue to be populated without any problems.

The full transformation for this smell is:

1. Add **`Initialize Array`** node outside the loop
2. If **`For Loop`**: Wire loop count to array size, else create a large constant and wire to array size
3. Remove **`Build Array`**
4. Add **`Replace Array Subset`**
5. Wire loop iteration count to **`Replace Array Subset`** index input
6. Wire value to be added to **`Replace Array Subset`** value input

Figure 5.7 shows the program after the ***Build Array in Loop*** transformations has been applied. The highlighting on the left side of the Figure shows the **`Initialize Array`** Nodes. Since the two arrays are of the same type, the initialization can be used for both. The highlight on the inside of the loop shows how **`Replace Array Subset`** is used in place of **`Build Array`**. The before program (Figure 4.1) can be found in Section 4.2.2.

5.3.2 String Concatenation in Loop

The transformation for this smell is very similar to that for ***Build Array in Loop***. The best transformation that the AE Specialists had was to use an array to store the different strings being concatenated. This array is initialized outside of the loop

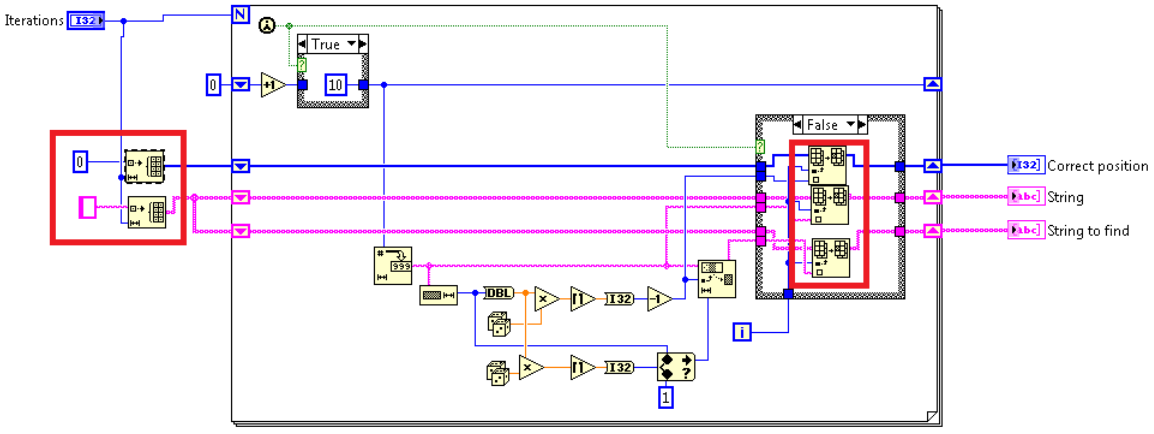

Figure 5.7: Transformed LabVIEW example showing all Build Array Nodes transformed. Left highlight shows initialization and right highlight shows Replace Array Subset

(requiring user involvement to set the array size) and the strings are added to the array using **`Replace Array Subset`**.

The one difference in this transformation occurs after the **`Loop`**. For the result of the program to remain the same, a **`String Concatenation`** node must be applied to the array of strings. This will create one large string, which was the result of the initial program.

The full transformation for this smell is:

1. Add **`Initialize Array`** outside the loop (input side)
2. If **`For Loop`**: Wire loop count to array size, else create a large constant and wire to array size
3. Remove **`String Concatenation`**
4. Add **`Replace Array Subset`**
5. Wire Loop iteration count to **`Replace Array Subset`** index input
6. Wire value to be added to **`Replace Array Subset`** value input
7. Add **`String Concatenation`** outside of loop (output side)
8. Wire output array to **`String Concatenation`**
9. Wire **`String Concatenation`** output to existing wire for the concatenated string

Figure 5.8 shows an example of this transformation after it has been applied to a LabVIEW program. The highlighting shows the relevant parts of the transformation. The key difference in this transformation when compared to ***Build Array in Loop*** is the **`String Concatenation`** node that is applied to the array. This is highlighted on the right side of the figure. The before program (Figure 4.5) can be found in Section 4.2.9. As mentioned in that section, the second **`String Concatenation`** node in the **`Loop`** is a false positive and thus has not been transformed.

As this transformation is nearly identical to the transformation for ***Build Array in Loop***, it has the same issue when the smell occurs in a **`While Loop`**. In

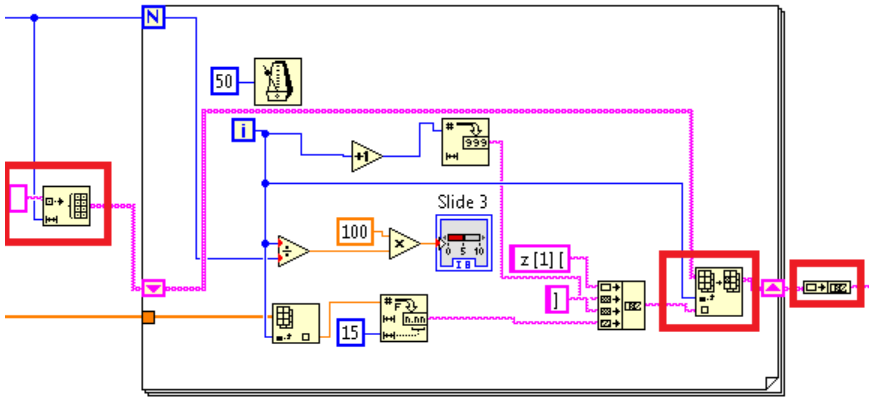

Figure 5.8: Transformed LabVIEW example removing String Concatenation Node in Loop transformed. Left highlight shows initialization and Middle highlight shows Replace Array Subset, Right highlight shows string concatenation

these cases, the initialization cannot be fully automated because the number of loop iterations cannot be determined algorithmically. Thus, the user is informed that the size of the array should be changed to reflect how many times the **`While Loop`** will iterate.

5.4 Wizard Transformations

The final set of transformations involve those that cannot be performed semi-automatically. This is either because they involve setting the properties of certain elements or because they require extensive effort from the user to continue. In these instances, the tool aims to guide the user through the process with suggestions, including a step-by-step guide to fix the problem.

5.4.1 Multiple Array Copies

The AE Specialists were unified about trying to avoid array copies, but they were rather split on the best way to solve this problem. In general, fixing this smell usually requires changing the algorithm or entirely refactoring the program. In the examples shown to the participants, the most common solution was to completely change how the program worked so that it did not require making multiple copies of the array. This is not something that can be reduced to a transformation, especially when it requires asking users specific questions about what they are trying to accomplish.

When pressed for a solution that could be turned into a transformation and applied to code, the AE Specialists' most-often-mentioned fix was to wrap the array copy inside of an **`In-place Structure`**, but they worried that this might be leading users down a path that is not optimal. Namely, they worried that users might start using these structures in other, more inappropriate situations. Another problem with the **`In-place Structure`** as a solution to this problem is that adding an **`In-place Structure`** around a wire split will not eliminate the array copy, and it will end up having no impact on the performance.

The majority of the AE Specialists said that the best way to go about fixing the problem would actually be to show the user every place where the memory is copied. Further tool guidance can then provide detailed advice. Since the most common answer was to entirely change the algorithm, the tool explains methods to reduce or eliminate these copies. There are a few different ways that an array copy can occur, so the program tries to tailor the advice based on the specific instance that

was recognized by the tool.

For example, one simple way for an array copy to occur is if the user wires an array into a structure two (or more) different times. Doing this will cause two copies to be created regardless of how they are used inside of the structure. By contrast, if the array is only wired once to the structure, multiple copies will not be created. When the tool detects this case, it suggests to change the wiring into the structure.

Another way copies are created is to branch the wire of an array to two different nodes that modify the contents of the array. While the branching can be detected and the user can be notified, removing this from the program is problematic. The tool describes why this might be a problem and explains why the program creates multiple copies of the same array. The suggestion to improve this is to try and refactor the program in such a way that the wire branch is not required. At times this may not be possible, as two arrays may actually be needed, but the suggestion is to change the program so that the program only uses one copy of the array, and to remove the branching.

In addition to these suggestions aimed at removing copies, the tool also offers the alternate suggestion of wrapping the offending code with an **`In-place Structure`**. This structure will try and perform all of the operations inside it in a manner that will reduce memory allocation if possible. The tool informs users that this works in some cases, but in many others it has little impact.

A final suggestion that the tool provides when appropriate is to move the array copies outside of loops if the user cannot remove them entirely. By removing them from a **`Loop`**, it will reduce the number of copies that are created. This can be enough

to improve the performance.

One of the main points that was brought up by the AE Specialists in relation to this smell is that they would like the ability to see how memory is allocated throughout a program. Having a tool that could do this would allow them to see where an array is created, where it is copied to create a new instance, or where it is simply modified. Many stated that having a tool that shows this information would greatly aid themselves (and likely other users) in fixing the problem, or at the very least help inefficient programs be more easily debugged.

5.4.2 Non-Reentrant subVI

Setting a subVI as reentrant or non-reentrant is a setting and can not be done programmatically. While it is a rather simple change, it is one that the tool has to guide the user through. In this case, the tool will instruct the user how to get to the proper menu and then how to set a subVI as reentrant.

The AE Specialists stated that most end-users are rather confused about this and there are only specific instances when this transformation should be recommended. Any subVI that contains data that is being stored on a physical resource should be reentrant.

They also mentioned that this is rarely a performance problem that they see end-users making, primarily because end-users do not often make code that is reusable, thus there is only one instance of a subVI being called in the program anyway. According to the interviewees, the most crucial case where reentrancy matters is when

programming for an FPGA. LabVIEW already has all subVIs set to be reentrant in those cases, meaning that the problem is mostly inconsequential in those instances.

Despite this, the wizard is set up to inform the user why having a non-reentrant subVI could cause performance problems. It will then give the steps on how to open the properties menu of the subVI and change it to reentrant execution. It also recommends that the user change the priority from normal to subVI priority to help with the timing of the program.

5.4.3 No Queue Constraint

Most of the AE Specialists felt like **Queues** were not used by end-users. One even stated that end-users often view **Queues** as scary or confusing and try to avoid them if at all possibly. In general, they did not think that this smell would occur often unless a user was saving the LabVIEW program to a real-time target that had limited memory. In those cases, the constraint on the queue and how the program handles hitting that constraint matter greatly.

Adding a constraint to a **Queue** is fairly straightforward in LabVIEW, but when doing so, there are several questions that must be dealt with by the user to set up the program correctly.

The key question is what the program should do when the constraint is hit. The AE Specialists mentioned two primary options. The first is to silently drop the data that the user is trying to enqueue. In general, the AE Specialists did not like this option. They much preferred displaying a runtime message stating that the queue

had reached the constraint limit and that the element could not be added to it. This would let the user know what is going on with the program and allow making a change.

When this smell is detected (and the user decides to address it), the tool will gather input from users on how big they want the queue to grow and what they want the program to do when the constraint is hit, to simply wait or to do a lossy enqueue and lose data. Once this data has been gathered the program will guide the user through steps on how to add this to their queue.

5.4.4 Terminals Inside a Structure

The transformation for fixing this smell is much more complex than just moving all **`Terminals`** outside of the structure, though that would seem like the obvious solution. Often those terminals are required for the correct execution of the VI. For example, there could be a boolean terminal inside of a **`Loop`** that is used to control a **`Case Structure`**. When the boolean is triggered on the front panel, a certain case will be run. In this instance, the terminal cannot be moved outside of the loop as it would mean that the boolean could never be triggered while the loop is running.

Figure 5.9 shows an example of this. In this instance, there is a boolean terminal, "Add to Array", that is controlling a **`Case Statement`** inside the **`While Loop`**. The user can interact with this terminal from the front panel while the program is running. When the user changes the value of this terminal to true, values will begin to be added to the array. If the terminal value is changed to false, then vales would

no longer be added to the array. This example also has a double terminal, "Value to Add", inside of the loop that is added to the array whenever Add to Array is true. Since it is located inside of the **`While Loop`**, the user can change this value during execution and have that value be added to the array.

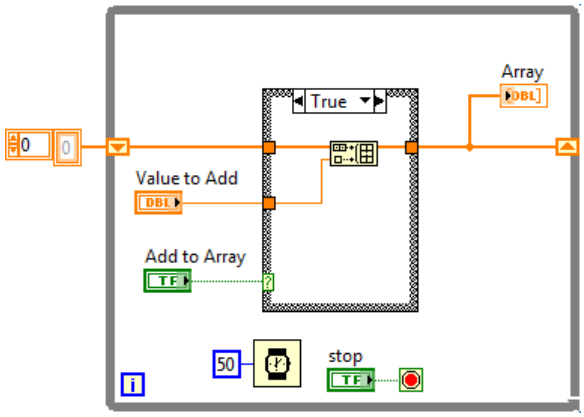

Figure 5.9: LabVIEW example demonstrating terminals (Add to Array Boolean) that must be included in a structure (while loop) for the program to execute correctly

Moving both of these terminals outside of the structure would completely change how this program behaves. If "Add to Array" is moved outside of the loop, then it can only have one value (read before the **`While Loop`**). So if it was set to false, then nothing would be added to the array, whereas if it was true, every value would be added. Similarly, if "Value to Add" was moved outside of the **`While Loop`**, then the same value would be added to the array every time the loop executed (providing that "Add to Array" was true).

It is instances like this that make this smell difficult to transform. The tool can advise the user on why having terminals inside of a structure are bad, but ultimately if they are required to be inside for algorithmic reasons, then they should not be

removed.

The tool handles this by prioritizing which **`Terminals`** to recommend for removal from structures. Often the largest cause of performance slowdown is a program being required to continuously update graphs or terminals inside of a loop as it causes the UI thread to be updated every iteration which greatly slows down the execution of the program. For this reason, the tool will ask the user if charts or graphs need to be updated inside of a **`Loop`**. If they can be moved outside of the structure, the program will execute more swiftly. It is up to the user to determine if they can afford to move **`Terminals`** or if the **`Terminals`** are required to be inside the structures.

Figure 5.10 demonstrates the results of this transformation, which involves moving the terminals outside of the loop. The before program is Figure 4.8 and can be found in Section 4.2.12.

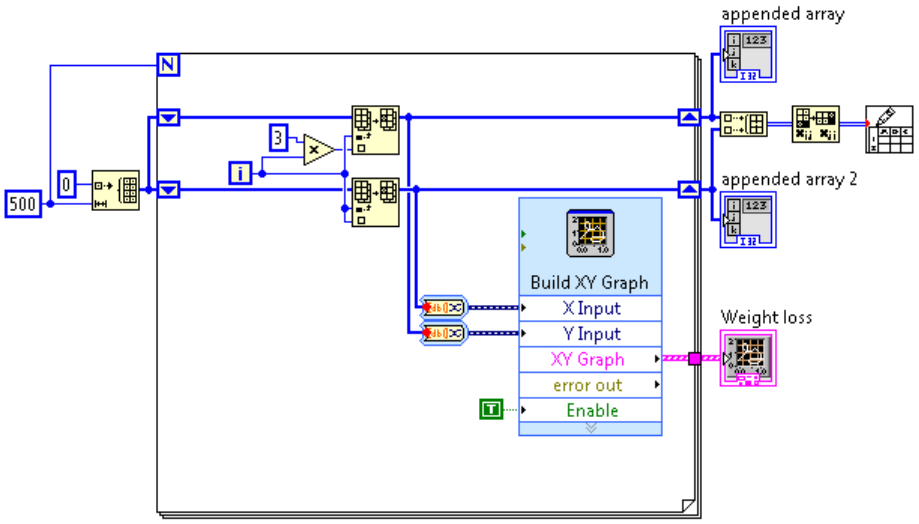

Figure 5.10: Transformed LabVIEW program showing three terminals moved outside of the loop

5.4.5 Too Many Variables

Too Many Variables had the widest range of possible solutions. However, all of those solutions involved getting rid of as many of the variables as possible. In general AE Specialists stated that users add variables to their LabVIEW code because they are trying to replicate the programming that they are used to in textual languages.

The AE Specialists boiled having too many variables down to one primary problem, which is the possibility that a variable might be written to at the same time due to the inherent parallelism of LabVIEW. This would result in a race condition and could disrupt the behavior and performance of the program.

Unfortunately, this problem is difficult to create a transformation for. While the tool can detect when there are multiple writes to a specific variable, the number of ways that they can appear in the code are varied, and fixing the problem usually requires reworking the code itself and changing the algorithm. For example, in the LabVIEW code that was shown to the AE Specialists during the interview, most of them mentioned that the easiest way to fix the problem would be to create a **`Shift Register`** on the **`Loop`** that would store the data. This would remove the variable entirely. However, this only works if all of the writes to that variable are inside the same loop. If they span multiple structures, then it gets much more complex.

Due to the multitude of ways that this problem can appear, a transformation could not be created. Instead the tool provides information about the kinds of problems that variables can cause, as well as ways to remove the variables. It makes a few different suggestions and explains how to use functional global variables, how to use

shift registers to store data in loops, and how to avoid using them entirely.

5.5 User Interface

The user interface for the transformations is based on the interface that was detailed in the previous chapter. When a user clicks on a smell in the table at the bottom of the LabVIEW IDE, new information populates the details pane located on the right.

With basic SDPA, this pane contained only information about the smell a brief description of how it could be fixed. The addition of transformations increases the information that this pane holds. For those smells that have a Full Transformation, a button is now located at the bottom of the pane, along with information that describes the transformation. This button will perform the transformation for the given smell. Partial Transformations contain a similar button, but once clicked the user will have to provide details to complete the transformation. Smells that only have Wizard Transformations contain a button that will open the wizard and guide the user through the process. Figure 5.11 shows an example of the Wizard for the Smell ***Terminals in Structure***. This figure is a mock up of the interface as the actual interface was implemented in an unreleased (confidential) version of LabVIEW.

Many of the smells and transformations contain more information than can be displayed in this pane. For those smells, another button was added to open a pop-up window that contains more information about the smell and the transformation. This pane can either be docked somewhere in the IDE or can be left to float above the code.

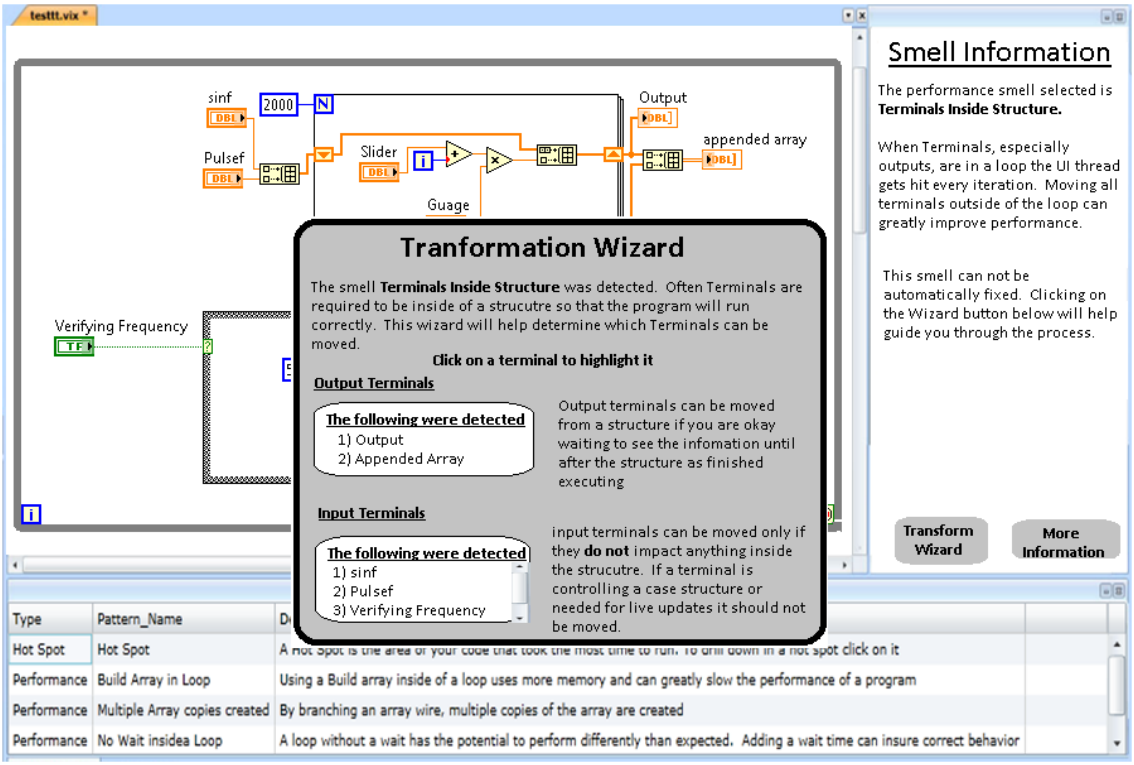

Figure 5.11: The Transformation Wizard for the smell Terminals in Structure.

An additional change that was made due to feedback from users, was to provide links to white papers or National Instruments web pages if applicable. While this information did not exist for every detectable smell, helpful links that could be found were added.

Chapter 6: Evaluations

The nine claims discussed in Chapter 3 were evaluated through a series of four studies. The first study evaluated the approach's effectiveness at finding and fixing performance problems on a corpus of "real world examples" retrieved from the LabVIEW forums. This study essentially assumed an ideal user, and real users might apply the approach differently than an ideal user. Therefore, two additional studies with real users examined how well the approach would help users to find and fix performance problems. Finally, a case study examined whether it would be possible to generalize the concepts to another data flow language (Chapter 7). Together, these four studies evaluated the nine key claims summarized in Table 3.1 (Section 3.2).

6.1 Real World Examples Study

The first study involved testing the tool on real world examples. The goal was to determine if detecting smells could accurately find real performance problems, resulting in a sizeable impact on the performance of programs. Several different analyses were performed, corresponding to the six claims evaluated by this study.

To study these issues, a corpus of real world LabVIEW programs was gathered that the tool could be run on. The source for these programs was the LabVIEW online forums[1], where users post questions about how to create or improve programs. While

[1] http://forums.ni.com/t5/LabVIEW/bd-p/170

there are many LabVIEW programs that are available on the forums, the parameters of this study required a focused search for programs that had performance problems. Investigating several claims required, in particular, comparing how the tool would improve programs to how experts would improve programs. So a corpus of "Before" and "After" programs was needed, namely, a set of pairs of programs, where the first in each pair was an uploaded program from a user that contained a performance problem before fixing by an expert, and the second was a version after fixing by an expert so the performance problem had been removed.

To gather this repository, the LabVIEW forum was searched for specific keywords (specifically, "slow program" and "poor performance"). The results were then filtered to only include forum posts that contained attachments. This set of posts was then read to find those that had poorly performing LabVIEW code as well as a solution that had been provided by an expert. In this case, an expert was considered to be a person with a high number of posts, a high proportion of posts that had been marked as correct solutions by other users, or a high level of activity on the forums. In some instances, the solution was accepted if it was provided by a non-expert who did not meet these criteria, but only if an expert subsequently stated in the forum thread that the non-expert's solution was correct and that is what they would have done, or if such an expert made a small suggestion about improving the non-expert's solution but did not upload new code. In these latter cases, if an expert stated that a modification should be made to the program without uploading a solution, then the change was performed manually by the researchers conducting this study, in order to reflect the change that the expert would have made. Forum posts were reviewed until

30 test programs with Before and After versions were obtained.

The six subsections below discuss analyses performed with this corpus to investigate each of the six claims related to this study.

6.1.1 Prevalence of Performance Smells

The first analysis examined the frequency of each smell in order to evaluate Claim 1, namely whether smells are common in end-user dataflow programs that have performance problems. The first step was to run the SDPA prototype tool on every Before program contained in the repository. This created a list with frequency data for each smell including how often smells occurred, both in terms of total number of instances as well as how many unique programs they were contained in. A few of the programs in the repository had subVIs associated with them. In those instances, the data for the subVIs are included in the information for the main program VI and does not show up as having a smell in two separate VIs, to avoid double-counting.

The analysis identified 150 smell instances, or an average of 5 smell instances per program. We found that every Before program in the corpus had at least one smell. These results confirm the claim that smells are common in end-user dataflow programs that have performance problems.

Table 6.1 shows the information that was gathered for each specific smell, including total number of instances and the number of programs affected. It is clear that not all smells are equally common. The first three smells listed below account for 70% of all the smell instances that occurred in these 30 programs, and the top 5 accounted

for 87% of all the smell instances.

Table 6.1: Prevalence of smells detected by SDPA in Real World Examples Corpus

	Total number of Instances	**Unique VI's with smell**
No Wait in Loop	37	19
Terminals in Structure	34	18
Build Array in Loop	34	12
Uninitialized Shift Registers	14	6
Multiple Array Copies	11	5
Sequence Structure	6	4
Non-reentrant subVI	3	2
Too Many Variables	4	3
Infinite While Loop	3	3
No Queue Constraint	2	2
String Concatenation in Loop	2	2
Redundant Operations on Large Data	0	0
Total	150	30

Several of the smells deserve additional discussion.

The total number of smell instances for the ***No Wait in Loop*** smell is high, in part, due to three programs that had 5, 5, and 7 instances of the smell, respectively. These instances were higher due to a large number of subVIs that all followed the same bad patterns, as if the smell instance had been copied and pasted, or as if the programmers did not know that this was a bad pattern and had manually repeated it several times.

The ***Build Array in Loop*** smell also had a particularly high average (2.58) number of instances per program. Manual examination of the programs showed that the most common usage of this smell was for a user to have several arrays being

constructed inside the same loop. Each array had a corresponding **`Build Array`** node inside of that loop.

Both ***Multiple Array Copies*** and ***Uninitialized Shift Registers*** also had a high number of instances per program (2.3 and 2.2, respectively). Reviewing the programs involved revealed that these, too, appeared to be situations where users either copy-pasted a mistake or manually recreated it many times. In particular, there were no cases where all of the shift registers were initialized but one: either the smell did not occur, or else it afflicted all of the shift register inputs.

The other smells all appear less frequently, but all (with the exception of ***Redundant Operations***) were detected by the tool in at least two programs.

6.1.2 Effectiveness of SDPA at Finding Problems Fixed by Experts

Claim 2 is that SDPA automatically identifies most performance problems that experts find. To test this claim, all of the Before programs created by users were reviewed and compared to the After programs created by experts. A list was made of all the locations where the Before programs were modified. Then, this list of locations was compared to the list of smell instances that SDPA had identified for the analysis in Section 6.1.1, above. It was then possible to compute what proportion of the expert-identified locations were identified by SDPA.

Experts made changes in only 8 locations other than the 150 smells identified by SDPA. Thus, SDPA successfully identified the vast majority (95%) of the performance problems identified by experts.

Table 6.2: Prevalence of optimizations performed by experts that are not detected by SDPA

	Total number of cases	Programs
Auto-Indexed Tunnels	4	3
Coercion Dots	2	2
Buffer Allocations	2	2

All of these uncaught optimizations are, to a certain extent, algorithm refactoring. There were several After programs in the corpus that were completely changed by the expert to make the code more efficient. One example of this was a program that had several nested loops. The expert stripped away all of the extra loops along with the necessary changes to insure the program still created the desired output. This not only fixed the performance problem the user was talking about, but it also made the program much easier to read and understand.

The 8 uncaught optimizations not identified by SDPA fell into three general categories, as shown in Table 6.2. The first change made by experts was to replace an **`auto-indexed tunnel`** with something more efficient. While this tunnel looks much cleaner than a **`Build Array`** node, it actually exhibits the same performance issues in a loop and, like the **`Build Array`**, causes many buffer allocations and potentially high performance problems. Nonetheless, tool was not designed to identify this problem because during the interviews, auto-indexing was not mentioned as problematic. In fact, when asked about the best transformation for the ***Build Array in Loop*** smell, some AE Specialists mentioned replacing it with an **`auto indexing tunnel`**. Likewise, auto-indexing tunnels would occasionally spark a discussion where forum members would argue about their merits and whether they performed the same

as a **Build Array** node. So even though removing **auto-indexing tunnels** could sometimes improve performance (and sometimes not), it is not clear that all experts or AE Specialists are aware of this fact. Figure 6.1 shows two example programs that will perform nearly the same. The top program contains the smell ***Build Array in Loop***, while the bottom program uses an **auto-Indexed Tunnel**.

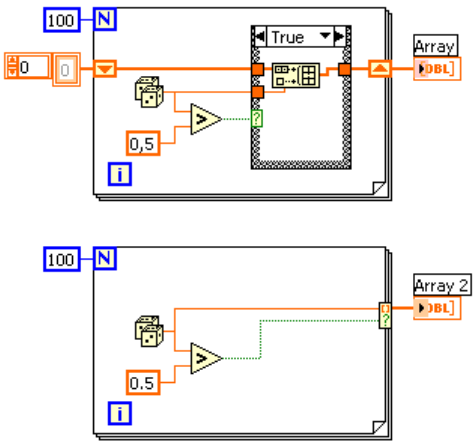

Figure 6.1: LabVIEW examples showing Build Array in Loop(top) and Auto-Indexing Tunnel(bottom). The performance of these programs is nearly identical.

Another optimization that experts identified but that went undetected by the tool was type coercion. LabVIEW will automatically change the type of a wire value if required for input to a specific node. For example, if a wire is an integer but is required to be a double to be an input to a node, LabVIEW will coerce the type automatically. Figure 6.2 shows a small code snippet that contains a coercion dot (the red dot on the **Add** node). A few expert solutions made a point of removing these dots with the claim that too many of them would affect memory usage. However, other expert LabVIEW programmers maintained that in the majority of cases, they

have little effect on memory. The cases from the corpus that had these changed by experts were tested to determine performance change. In one program, the effect was minimal, while the other showed a large improvement, though the solution was also heavily refactored elsewhere.

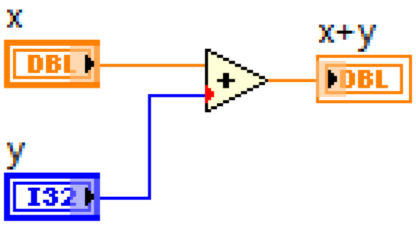

Figure 6.2: LabVIEW snippet showing type coercion (red dot on the blue wire input to the Add Node)

The final optimization that the experts looked for that was not caught by the tool was similar in nature to the coercion dot, but had much more impact on the performance. That was to show all of the buffer allocations of a program and to try and minimize them. This is similar in essence to program refactoring, but since it had the specific goal of removing one thing it warrants a discussion. Showing the buffer allocations can be particularly helpful when doing memory optimizations as it shows dots every place that LabVIEW compiler allocates memory. This usually means that the program is creating a copy of the value on the wire, and in the case of arrays, this can potentially cause high memory use, particularly when the buffer allocations occur inside loops. While some of the smells that the tool detects cause buffer allocations, namely ***Build Array in Loop***, the tool does not check explicitly check for buffer allocations. There were two programs in the corpus where the expert worked to remove as many buffer allocations as possible, which significantly improved

performance.

On a side note, many experts made a point of helping new LabVIEW users try to make their programs more understandable. Even for programs where the experts simply applied a simple fix, they would have describe ways that the user could make their program easier to understand. Based on the interviews with the AE Specialists, many users have problems creating code that can be read or understood by other people, and the experts seemed to be trying to remedy this by helping whenever they could.

Another thing that they were focused on was getting users to write code that was more in line with the visual language style and to try and move away from best coding practices of textual languages. This often involved explaining the best way to program something in LabVIEW, and while this did not always greatly affect the performance of the program, it did often have a minor effect.

In summary, the SDPA found almost all of the problems that experts fixed in corpus programs. One of the missed kinds of edits, **`auto-indexed tunnels`** could be added as a future smell. The other two kinds of edits involved substantially rewriting the program algorithm, such as for the purpose of eliminating buffer allocations.

6.1.3 True and False Positives of SDPA

The third claim is that smells identified by SDPA usually indicate real performance problems. Just because SDPA indicated the presence of a smell in a certain location, or just because an expert actually did perform an edit in that location, does not nec-

essarily imply that the smells actually caused real performance problems. Therefore, a further analysis was required to assess what proportion of the smell instances truly harmed performance.

This was accomplished in several steps. First, for each smell identified by SDPA in a Before program, a corresponding fix was created. This fix was taken from the After program if the expert had supplied a fix at that location in the program. Otherwise, a fix was generated using the transformation offered by the tool if applicable. Otherwise, for a few situations where the transformation offered by the tool could not be applied, then the Before version was used unaltered as the fixed version (implying that there was no good fix for the smell, and no performance improvement was possible). Second, the Before program was run and the fixed program was then run. In each case, the execution time and the memory usage were both measured. Third, the percent improvement between the fixed and the original program was noted, both in terms of impact on execution time and on memory usage. Fourth, a smell was categorized as a true positive if its impact was at least 15% on execution time or 10% on memory usage, or a false positive otherwise. Fifth, the overall numbers of true and false positives were computed.

Overall, this analysis revealed that most smells identified by SDPA did indicate an actual performance problem. Of the 150 smell instances, only 48 (32%) were false positives, while 102 (68%) were true positives.

Most of the false positives were due to the ***Terminals in a Structure*** smell. While this smell occurs frequently and does sometimes cause performance issues, there are cases where a terminal has to be inside of a structure due to the algorithm design.

In such a case, as noted above, neither the expert nor the tool could actually fix the problem, so the Before version of the program was used as the fixed form as well, leading to a 0% improvement. In addition, in some cases when terminals could be moved outside of loops, the impact still fell below the threshold to determine a false positive. This does not necessarily signal that this smell should be abandoned, as there are many cases where moving the terminals outside of a structure significantly improves performance.

Another interesting note is that every instance of the ***Sequence Structure*** smell was determined to be a false positive, as it had little impact on the actual performance of the program. Similarly, the ***Too Many Variables*** smell showed little impact on the programs in the corpus, and all but one instance was marked as a false positive.

One interesting set of false positives was a few instances of ***Build Array in Loop***. In these cases, either the **`Build Array`** node was in a loop that iterated very few times and did not have an impact on the memory, or it was in a conditional statement that executed very few times inside of the loop. While this smell in considered one of the worst offenders, as noted by AE Specialists in the formative study, the results of this analysis shows that there are certainly cases where it has little negative impact.

6.1.4 True and False Positives with SDPA and Smell-Aided Profiling

The analysis above was continued by turning on Smell-Aided Profiling, to evaluate Claim 4 that Smell-Aided Profiling improves the false positive rate of SDPA and negligibly impacts the true positive rate. As before, SDPA was run on all of the Before

programs to identify smell instances, but Smell-Aided Profiling was also activated in order to provide the execution data needed to filter down the list of smell instances to only include those that also corresponded to hotspots. This reduced the number of instances identified, which was expected to decrease the number of true and false positives. The question, of course, is whether Smell-Aided Profiling would successfully decrease the false positive rate much more than it decreased the true positive rate.

Enabling Smell-Aided Profiling(SAP) yielded a total of 112 smell instances, of which 24 were false positives and 88 were true positives. Thus, SAP decreased the total number of false positives by 50% while only decreasing the number of true positives by 13%. Table 6.3 shows the results obtained.

Table 6.3: True and false positives with SDPA and SAP

	True Positives	**False Positives**
Smell Detection	102	48
Smell-Aided Profiling	88	24

The primary reason there was not a larger decline in the false positive rate is due to how the smells are profiled. If there are multiple smells involving a structure, then the smell instances' timing probes extended to the structure node, so they are essentially profiled together by profiling the runtime within that structure. So if a loop has a terminal inside it that is not causing any performance issues, but it also has a **`Build Array`** node that is, both would be marked as a hotspot. This would enable the ***Terminals in Structure*** smell to slip through, even though it is not specifically causing problems. How smells are profiled could be refined in future versions by adding profiling inside of structures to pinpoint the exact areas that are

causing problems.

The primary reason for the decrease in the number of true positives is that the profiling was unable to measure the impact of ***Uninitialized Shift Registers*** as the impacts of that smell only show up when the program is run several times in a row, which the profiler does not check for. For example, one execution of a program that is profiled might use 5000 KB of memory for a 2 minute run. This might not be extreme (relative to the hotspot threshold in Chapter 4) and no significant performance problems would be detected. However, with the shift registers uninitialized, every single execution will cause the memory use to increase by around 5000 KB (depending on the run time). The second run would use 10,000 KB, the third 15,000. The profiler does not filter over multiple runs, and if it takes a long time for the problem to show up, the tool cannot determine that it is actually an issue. This problem could be addressed by allowing the profiler to aggregate over multiple runs, thereby enabling it to accumulate enough information to note that a certain smell is actually turning out to be a hotspot.

6.1.5 Impact of Fixing Smells

Claim 5 is that for most smells, tool-based transformation improves performance. This claim was evaluated applying all possible tool-based transformations to the SDPA-identified smell instances, then computing the average resulting impact on performance for each kind of smell over all instances of that smell. All smell instances were included, meaning that Smell-Aided Profiling was turned off. The overall average

impact of fixing a smell instance was computed, as well.

Table 6.4 summarizes the improvements in performance when transformations were applied on a per-smell instance. Several key points are evident in this table.

First, overall, fixing each individual smell instance led to an execution time improvement of 26% and a memory usage improvement of 12%. This result is important because it demonstrates that even fixing a single instance of a smell can have a noticeable impact on performance. As noted in Section 6.1.1, corpus Before programs contained an average of nearly 5 smells per program, so users could anticipate even greater benefits when applying multiple smells, which Section 6.1.6 explores in detail.

Second, for 7 out of the 11 smells that actually occurred in the corpus, applying the transformations improved execution time by at least 10%. This indicates, as expected, that tool-based transformation improves performance for most smells. For 3 of these 7 smells, memory usage also improved by at least 10%.

Third, however, the classic tradeoff between execution time and memory usage was apparent. Specifically, memory usage increased (negative improvement) for 4 of the 7 smells where execution time improved. The details of why this happened are discussed in subsections below.

Finally, the transformations achieved bigger impacts on the smells listed near the top of the table than they did for smells listed near the bottom of the table. As smells are shown in decreasing order of prevalence (as in Section 6.1.1), it is clear that the transformations help the most in the situations that users will likely encounter most often. Subsections below explore the impacts of the smell-based transformations in detail.

Table 6.4: Average impacts on performance of applying each tool-based transformation (per smell instance)

Smell	Execution Time Improvement	Memory Usage Improvement
No Wait in Loop	37%	-13%
Terminals in Structure	23%	-1%
Build Array in Loop	34%	37%
Uninitialized Shift Registers	22%	86%
Multiple Array Copies	25%	-16%
Sequence Structure	-2%	-1%
Non-reentrant subVI	13%	-2%
Too Many Variables	-1%	-1%
Infinite While Loop	0%	1%
No Queue Constraint	0%	0%
String Concatenation in Loop	16%	20%
Redundant Operations on Large Data	N/A	N/A
Average over all smell instances	26%	12%

6.1.5.1 No Wait in Loop

The transformation for ***No Wait in Loop*** is to simply add a **`Wait`** to the loop. This simple transformation greatly improved the execution time, while also slightly increasing the memory. The primary culprit to the memory gain seems to be due to the fact that many of the affected programs used very little memory to begin with, and small fluctuations show up as much larger percents. In addition, adding a **`Wait`** to a loop allows the inherent parallelism of LabVIEW to run and any other structures that can be run in parallel will be (including tother code that conserves memory). This is often enough to negatively affect the memory values used. However, this transformation in general shows impressive gains in execution time particularly for such a simple transformation.

6.1.5.2 Terminals inside a structure

For this smell, there are some cases where a terminal is required to be in a structure for algorithmic reasons, such as a button that controls a case statement or a stop button for a loop. For the transformation for this smell to be successful, all of the terminals must be moved outside of the structure. Due to this fact, instances where the terminal was required to be in the structure could not be transformed.

Nonetheless, where applicable, the transformation itself showed great improvement in terms of average impact on time, with negligible impact on memory usage. The best cases often showed a large improvement in both memory and time, and it may be worthwhile to focus the tool on those cases which involved moving large output terminals such as arrays and charts outside of the structure, which was usually a loop.

6.1.5.3 Build Array in Loop

As mentioned previously, the transformation for ***Build Array in Loop*** is not always simple, as the loop count has to be accounted for when looking at a **`While Loop`**. The transformation is straightforward for a **`For Loop`**, and often this slightly affected the overall impact on performance. Namely, removing a **`Build Array`** from a **`For Loop`** showed a greater improvement than removing it from a **`While Loop`** as it was usually much more precise when initializing the array outside of the loop.

There were two cases where the improvement did not help or showed little improvement. These cases both involved a **`While Loop`** and a miscalculation of the

loop counts. Basically, the array was initialized by the transformation to be much larger than was actually required, which caused more memory to be used and the program to slow down very slightly. It should also be noted that these cases were dealing with small arrays, and cases like these may get filtered out by the profiler.

6.1.5.4 Uninitialized Shift Register

This transformation by far shows the most improvement, both in terms of time and memory. This is also a simple transformation, simply adding a constant array to the **`Shift Register`**, but it showed a nice improvement in terms of time, with an average improvement of 22% on execution time and 86% in memory. Even the worst instance still shows a 75% improvement in terms of memory.

Since the issues due to this smell can manifest differently over multiple runs, the transformation was also separately tested over a series of 5 runs with the time and the memory recorded each time. Both the execution time and memory usage remained nearly constant for each run after the transformation was applied. This is encouraging, as it signals that the behavior of the program will remain the same over every execution once this transformation is applied to fix this smell.

6.1.5.5 Multiple Array Copies

The evaluation showed that this transformation improved execution time but harmed memory usage. The memory issue was because when there were multiple instances of

this smell in the same structure, then applying the fix to one smell instance allowed execution time to increase and the program to thereby invoke the other smell instances more often, which ended up consuming more memory overall. This is an indication that this smell requires all instances in a program or structure to be fixed to fully improve the program. Unfortunately, this is also one of the smells that requires the most manual work to fix because the tool uses a wizard to guide the user through the process of removing smell instances. While it is still beneficial to point this problem out to users, it is possible that the way the transformation is applied is not helpful and may need work. There may be a more straightforward way of dealing with it that would be simpler and would allow the user to solve their performance problem more quickly.

6.1.5.6 Sequence Structure

The ***Sequence Structure*** transformation is among the most complex, and it is also one that showed barely any impact in terms of performance. In general, it had very little impact on the total time of the program and memory. As AE Specialists noted in the formative study, gratuitous use of **`Sequence Structures`** can harm performance, but clearly none of the uses of this structure in the corpus were causes of poor performance.

6.1.5.7 Non-reentrant subVI

Two of the detected cases of this smell were unable to be transformed because the detected subVIs were part of a built-in LabVIEW module that did not allow access to the internal block diagram. This meant that there was only one instance that could actually be transformed and measured. The one instance that was able to be tested showed a slight improvement, with setting the subVI to reentrant execution, showing a 13% decrease in the execution time with a 1.5% increase in the amount of memory used.

6.1.5.8 Too Many Variables

The transformation for the ***Too Many Variables*** smell is complex, as generally the best fix requires refactoring the program in such a way that it no longer needs variables. While the wizard attempts to guide users through this process, the performance improvement is negligible, and in fact slightly increases both the execution time and memory usage. It could be that there exist cases where this transformation would be useful, but they were not found in the corpus. Of the four detected instances of this smell, two of them were using the variables correctly, i.e. only writing to them in one place. (While they had many variables, they also written in a way that avoided the possibility of race conditions, which was the other main concern of AE Specialists regarding this smell.)

6.1.5.9 Infinite While Loop

Compiling statistics for this smell was challenging. An infinite loop can only be stopped by using the abort button, which means that the rest of the program does not get executed. By comparison, adding the ability to end the program with a button means the rest of the program will execute, which will increase both the execution time of the program and the memory usage (how much depends on the program). So instead of compiling statistics for the entire program, a snapshot was taking right before either the program was stopped using the abort functionality, or the loop was stopped using the added button. The transformation had very little impact, positive or negative, on the overall performance.

Nonetheless, it is worth noting that all three of these programs worked completely after this transformation was applied. While it did not improve the performance per se, it did fix the performance problem expressed by the users in the forum, which in these three cases was: "Why is my program stuck in this loop the entire time?".

6.1.5.10 No Queue Constraint

Even before the transformations were applied, the queues never appeared to grow very large. When the constraints were added by the transformation, they were never hit, due to the constraint size being set rather high. If the constraint size was lowered dramatically, it did have an impact on the memory usage as it stopped adding elements once the queue had reached a certain size. However, reaching this point would not have been the end result of the transformation wizard and as such, the

statistics were not included. Because of this, the transformation had no impact on performance, other than a very slight increase in memory due to the extra nodes used to display a message if the constraint was ever hit.

6.1.5.11 String Concatenation in Loop

Concatenating a string inside of a loop is similar to using a **`Build Array`** node, and the transformation is nearly the same. However, the impact was not quite as significant.

6.1.5.12 Redundant Operations on Large Data Stats

There were no instances of this smell in the corpus so it was not possible to test the potential transformations on real world examples.

6.1.6 Tool-based Transformations versus Experts' Fixes

The final claim evaluated in this study, Claim 6, says that tool-based code transformations semi-automatically improve performance almost as well as experts do. This claim was tested by running all applicable transformations on Before programs and measuring the resulting performance impact, then also independently applying all expert-based fixes to the Before programs and measuring the resulting performance impact. As in the analyses above, performance impacts were expressed as percentages relative to the Before program measurements.

The Mann-Whitney U test was used to compare the performance impacts of tool-based transformations to the impacts of the experts' fixes. This test was performed once using the impacts on execution time measurements for the 30 programs, then again using the impacts on memory usage measurements for the 30 programs. Thus, two statistical tests were performed.

Table 6.5 summarizes the average impacts on performance. Overall, applying the available tool transformations to a program achieved, on average, an execution time improvement of 42% and a memory usage improvement of 20%. In comparison, experts achieved 46% and 28% improvements in these measures, respectively. The statistical tests showed that these differences were not statistically significant. (The P values were 0.57 for execution time and 0.25 for memory, with N=30 in each case. The U values were 291.5 for time and 266.5 for memory) These results confirm that the semi-automated tool-based code transformations did improve performance almost as well as the experts manually did.

Table 6.5: Average impacts on performance (per program)

	Execution Time Improvement	**Memory Usage Improvement**
Experts	46%	28%
Tool	42%	20%

Although the expert solutions were better on average, in half of the programs (15/30) the expert solution and the tool solution were within 5% of one another, in terms of execution time and memory usage. Furthermore, 7 of those 15 solutions were within 1%. These cases primarily were programs with the smells as ***Uninitialized Shift Registers*** or ***No Wait in Loop***, which have straightforward solutions.

There were four programs in which the tool solution was greatly superior (better than 20% difference) than the experts' fixes. In these cases, the experts did not notice instances of the ***Terminals in Structure*** smell. All of these programs had that smell occur frequently, and removing them created the difference in the results. These cases highlight the potential value of SDPA even to experts.

There were seven programs in which the expert solution was greatly superior (better than 20% difference) than the tool-generated solutions. Five of these cases were a case of the programs being completely refactored by the expert to be much more efficient, particularly by correcting buffer allocations, and coercion dots (i.e., situations discussed in Section 6.1.2).

To summarize, the results from the analysis in Sections 6.1.1-6.1.6 indicate that SDPA usually does an accurate job of finding performance problems, that Smell-Aided Profiling improves this accuracy, and that tool-based transformations improve performance nearly as well as experts do. The main limitations are related to the tool's inability to completely rewrite algorithms. In most other ways, the tools are nearly as good as having an expert at hand to assist with diagnosing and fixing performance problems. The sections below evaluate how well this tool-based assistance helps users in practice.

6.2 Smell-driven performance Analysis User Study

The first user study performed was a laboratory study to evaluate the effectiveness of the first SDPA prototype without Smell-Aided Profiling, as well as to uncover op-

portunities for future prototype enhancements. This within-subject study evaluated the seventh research claim, which is: "SDPA helps end-user programmers to more quickly and successfully find and diagnose performance problems."

6.2.1 Methodology

Participants were recruited by sending emails to Application Engineers at National Instruments. These individuals are the primary support staff for customers who have problems with LabVIEW. Participants were required to have at least a year of LabVIEW programming experience, with the intention of ensuring that participants would have a plausible chance of finding and diagnosing performance problems even without the prototype's help. Through this process we recruited 13 participants. The study occurred at National Instruments' Campus.

Every participant was given two program-troubleshooting tasks, each involving a different LabVIEW program. They were informed that each program had one performance issue, and that their task was to verbally identify the problem and the solution. They were not required to actually implement the fix, but could do so if they desired. To begin the study, a short tutorial about the prototype was given to each participant. The tutorial included information about what the prototype was designed to do and what kinds of information would be displayed to the user.

Participants had a total of 40 minutes to do both tasks, and they had access to the smell detection prototype tool, without hotspot analysis, for only one of the two tasks. For the other task, they used the basic LabVIEW IDE. The study was set

up with a within-subject design to control for between-participant skill differences. In addition, the program ordering and condition ordering (with/without prototype) was counter-balanced to control for any between-program differences in difficulty and between-task learning effects.

The two LabVIEW programs used for this study were based on real-world examples of LabVIEW programs that contained performance problems. These programs were found on the LabVIEW forum[2], where a user had asked for help with troubleshooting the problem. In order to increase the likelihood that the participants would complete tasks in the allotted time, even without access to the prototype, the posted programs were slightly modified so that the effects of the performance problem were more pronounced when the participants were debugging them. Specifically, these two programs as posted had demonstrated the ***Build Array in Loop*** and ***Multiple Array Copies*** smells, respectively, which eventually caused very slow execution within a few minutes. To cause the problem to be more evident the array size was increased in each program so that the problem occurred within seconds.

Altering the real-world programs in this fashion did not bias the study in favor of the prototype because the alterations did not affect the behavior of the smell detectors. If anything, they made it easier for participants to complete tasks without the use of the prototype, because the performance problems became more obvious.

After the participants had finished debugging the two programs, they were given a brief questionnaire to determine their thoughts on the tool in general and how helpful it was to the debugging process. In addition, they were asked for suggestions on how

[2]http://forums.ni.com/t5/LabVIEW/bd-p/170

the tool could be improved as well as any issues they had with usability. For the current study, qualitative questions were used that attempted to draw out explanations of what was good or bad about the prototype. For example, participants were asked "Did the tool ever intrude on your train of thought or distract you from the task at hand? If so, what was distracting?" Further questions asked for specific feedback about the alerts, the frequency of notifications, the kind of the alerts and advice given, the effectiveness at drawing attention to the performance problem, and the prototype's usefulness overall. These answers were analyzed with thematic analysis, followed by another researcher's inspection of all categorization results to assess face validity.

6.2.2 Quantitative evaluation

As participants attempted to find and verbally report each problem and its solution, four measures of success were manually recorded: whether participants found each problem, how long finding the problem took, whether they found the solution after finding the problem, and how long finding the solution required. Each program in this study had one performance problem, for which there was one clearly correct solution, which was based on forum posts uploaded by experts. When participants gave incorrect answers (e.g., stating that the problem was in a place other than the real problem), they were told that this was incorrect and to keep looking. The participants could give up and move on, if desired (e.g., due to frustration). It should be noted that the participants were actively trying to find the problem and were not

simply pointing at areas of code and asking if that was causing the problem.

Once the data was collected from all of the participants, these measures were analyzed statistically. Fisher's exact test was used to test two null hypotheses about success (H1 and H2, below) and the Mann-Whitney U test was used to test two null hypotheses about time taken by participants (H3 and H4, below). The Mann-Whitney U test was used instead of a t-test because of the small sample size (13 participants).

Null hypotheses, tested with one-tail statistical tests:

H1: Participants will be as likely to find performance problems without the tool as with the tool.

H2: Participants who find performance problems will be as likely to find solutions without the tool as with the tool.

H3: Participants who find a performance problem will do so at least as quickly without the tool as with the tool.

H4: Participants who find a solution (after finding a problem) will do so at least as quickly without the tool as with the tool.

Overall, the study showed Smell-driven performance analysis had a large impact on participants' ability to diagnose problems (Table 2). Participants found the problem in 100% of the tasks with the tool, compared to only 38% in tasks without the tool. This difference in success rates was statistically significant ($p < 0.001$). Moreover, when they found the problem, they took only 4.49 minutes with the tool compared to 6.85 minutes without it, although this difference was not statistically significant (at $p < 0.05$).

Participants were able to find the solution in 92% of the tasks with the tool and only 60% without (not significant at $p < 0.05$). These numbers do not include those tasks were the user was not able to find the problem to begin with. When they did successfully describe a solution, they were able to do so faster with the tool, taking only 0.83 minutes compared to 2.37 minutes, a statistically significant difference (Mann-Whitney U=32, $p < 0.001$). There were no significant differences between tasks. In short, participants were more likely to find problems with the tool, and having done so, they were faster at finding solutions.

Table 6.6: Effectiveness of participants with and without the prototype tool

	With Tool	**Without Tool**
Success Rate at finding problem	100% (13/13)	38% (5/13)
Average time for finding problem (Minutes)	4.49 (13 cases)	6.85 (5 cases)
Success Rate at finding solution	92% (12/13)	60% (3/5)
Average time for finding solution (Minutes)	0.83 (12 cases)	2.37 (3 cases)

6.2.3 Qualitative Feedback

In addition to compiling the times and success rates of the users, their opinions about the tool were also gathered in a questionnaire given after the study. These questions were qualitative in nature and were focused on gathering as much information as possible about the participants' views of the system and if it was helpful. For example, users were first asked if they found the feedback provided by the tool helpful, if they answered yes they were prompted to describe how it was useful. Similarly, if they said it was not useful, they were asked to describe why.

In terms of usefulness, 100% of participants said that the tool was useful in helping them complete the task. Eight of the thirteen participants said that the tool drew their focus immediately on the problem areas of the program, which was helpful and helped them speed debugging efforts. Three participants specifically mentioned that it alerted them to areas that they hadn't even considered to be problems, and two participants stated that the tips for improving the program and good programming practices were the most useful part. Two participants commented on the fact that they felt the tool would help most LabVIEW users write better code.

Finally, one participant commented that the tool might save development time. In particular, this person felt that many users create code that has memory issues, but they often do not notice these issues until much later in the development process, at which point it takes a lot of effort to go back and refactor the code to get rid of those issues. By having a tool that immediately pointed out areas in the program that might mismanage memory the issue could be solved immediately, which could save a lot of time and effort.

In addition to finding the tool useful, all of the participants stated that the tool made LabVIEW easier to use. In particular, eight of the participants stated that the tool made it easier for them to find the performance problems and that it allowed them to do much less analysis of the code and focus more quickly on the area that contained the problem. One participant mentioned that instead of having to run the code, the tool provided immediate feedback that made it simpler to narrow down the area of the code to focus on.

One unexpected finding was that for several users, the tool served as a way to

remember the training that they had already taken. Two participants stated that it was a quick reminder of problem areas that LabVIEW has and once they knew the exact issue they were able to remember ways to solve the issue in the task. One participant mentioned that it reminded them of good practices, but often forgot to implement in their programs.

While the tool was found to be useful and made LabVIEW easier to use, that did not mean that it was issue-free, as participants mentioned two issues that needed to be addressed. Four of participants said that more assistance in fixing the performance problem would be beneficial. In particular, the notification and explanation provided by the tool was too general and did not apply to the specific problem they were trying to solve. Most said that having links to white papers containing more information about the problems and more detailed ways to fix them would be helpful.

Several participants did not want to be forced to use the tool, with one even cautioning that making this tool pop up like Clippy would make the tool a hindrance. In total, eight of the participants mentioned that they would prefer to a way to opt in to the tool's notifications (and be able to also opt back out) or never have the tool explicitly notify them at all.

Finally, three participants mentioned that they were confused with some of the suggested fixes. The users understood where the issue was, but the suggestion provided was not enough for them to easily determine what the solution should be, so they had to do a lot of searching in the LabVIEW help to determine what the suggestion actually meant. Adding links to more information about the problem as well as examples might help clear up the confusion.

This feedback helped to guide several of the minor user interface improvements mentioned in Section 5.5. These included, for example, adding further details about why smells impact performance, how to fix smells and related transformations, and links to white papers.

6.3 Complete Tool Study

The second user study was designed to evaluate the entire system, including SDPA, Smell-Aided Profiling, and semi-automatic transformations. This study consisted of having the users first find problems in LabVIEW code and then attempt fix the problems. This tested both the find and the fix portion of the system, and was used to evaluate claim 8: "Smell-based performance tools (including SDPA, Hotspot, and transformations) together help end-user programmers to more quickly and successfully fix performance problems."

6.3.1 Methodology

Like the previous user study, this study was conducted at National Instruments headquarters and involved Application Engineers. They were again recruited by sending emails to all Application Engineers at National Instruments. Participants were required to have at least a year of LabVIEW programming experience, with the intention of ensuring that participants would have a plausible chance of finding and diagnosing performance problems even without the tool's help. For this study, 32

participants were recruited.

To begin the study, a short tutorial about the prototype was given to each participant. The tutorial included information about what the prototype was designed to do and what kinds of information would be displayed to the user. It also showed users how to run different parts of the tools and how to access all of the relevant information.

Every participant was given two program-troubleshooting tasks, and each task involved finding and fixing two LabVIEW programs. They were informed that each program had performance issues, and that each task was to find and fix the problems in the task's programs. For one task, the participant could use the IDE with SDPA. For the other, the participant only used the LabVIEW IDE without SDPA.

The four LabVIEW programs that were used for this study were real-world LabVIEW programs that contained performance problems that had been found on the LabVIEW forum[3]. Each participant was given a description of the task's two programs as well as any information that had been posted on the forum that described the symptoms of the performance problem that each program was experiencing. In the event that a participant had further questions about the programs, they had access to the forum post and any information that was contained in it.

Each task contained one easy program and one hard program, based on the amount of expected work needed to find and fix the problem. Specifically, one easy program contained the ***No Wait in Loop*** smell, while the other had the ***Uninitialized Shift Registers*** smell. One hard program had the ***Build Array in Loop*** smell,

[3]http://forums.ni.com/t5/LabVIEW/bd-p/170

while the other hard program contained the ***Terminals in Structure*** smell. (Note that these are four of the most common smells, as shown in Table 6.1.) In the case of these four programs, those smells were not the only ones in the program; however, those were the smells that caused the primary performance problems, and fixing the smells would fix the symptoms that the user asked about in the initial forum post.

The study was set up with a within-subject design to control for between-participant skill differences. The tasks were counterbalanced in terms of when they were received (first or second) and what treatment they were in (tool or no tool).

Participants had a total of 20 minutes to do each task (a total of 40 minutes). Participants were directed to begin with the easy program of the task. If at any time users wanted to stop debugging a program, they were allowed to do so. If they were on the first program of the task, they could then begin debugging the second program. They were not allowed to go back to the first program once they started the second program of a task. The task time limit was strict, and participants were told to stop debugging once the 20 minute time limit had been reached for each task. This meant that it was possible for them to run out of time in the middle of debugging. If this was the case, they had to stop and move on to the second task or end the study.

Success was measured on a per-task basis by giving 25 points when participants found the problem of a program and 25 points when they fixed the problem of a program. Each task therefore had a total of 100 possible points, 50 for each program. Partial credit (12.5 points) was given in cases where the participant implemented a fix that was not optimal, but that did improve performance; this list of acceptable improvements for partial credit was developed prior to the study based on the set of

suggestions that users had proposed on the forum for fixing the program. As in the first user study, the researcher did not provide any coaching to participants about how to fix the problems, but participants could ask if they had yet succeeded so they could determine whether they could move on.

After the participants had finished the study, they were given a brief questionnaire as in the first user study to gather their opinions on the tool, its usefulness, and any problems they experienced with it.

6.3.2 Quantitative Results

Once the data was collected from all of the participants, these measures were analyzed statistically. The one-tail Mann-Whitney U test was used for the null hypotheses on success, time, and efficiency:

Null hypotheses, tested with one-tail statistical tests:

H1: Participants will be at least as successful debugging performance problems without Smell-Driven Performance Analysis as with it.

H2: Participants will be at least as fast at debugging performance problems without SDPA as with it.

H3: Participants will be at least as efficient at debugging performance problems without SDPA as with it.

Looking at the primary measurement for success, the points given for finding and fixing performance problems, the prototype tool demonstrated a clear advantage over the normal LabVIEW IDE. The average number of points received by participants

using the tool was 97.66, while those who did not have access to the tool only scored 67.19 points on average. The difference between the points data was found to be statistically significant (Mann-Whitney U=181 $p < 0.001$). Because the total possible number of points was 100, these results indicate that participants were able to nearly perfectly complete every problem diagnosis and repair within the allotted time, provided that they had SDPA.

In addition, participants spent far less time on tasks with the tool, requiring only 5.66 minutes with the tool versus 13.28 minutes without. Because none of them ran into the 20-minute time limit when using the tool, but they hit the time limit four times when completing tasks without the tool, there would have been an even greater difference in task time if the study had permitted participants to take as long as they wanted on the tasks. The time taken to complete the total task was found to be statistically significant (Mann Whitney U=181 $p < 0.001$), indicating that the tool does help users debug more performance problems in less time.

Using these two statistics, points earned and total time, the efficiency (points/minute) for each participant was computed. Participants using the tool had an efficiency of 26.42 pts/min compared to only 7.14 pts/min when debugging without the tool. The efficiency by participant was found to be statistically significant (Mann Whitney U=31 $p < 0.001$), which indicates that the tool helped users become over three times (3.7) more efficient at debugging performance problems.

Table 6.7 highlights these differences in points scored and time taken, and it presents additional statistics that help to explain several factors that contributed to differences in points scored and time taken. Out of the 32 cases where participants

had to find a problem using the tool, they succeeded every time. In contrast, there were 19 occasions where they failed to find the problem without the tool. However, once a participant had found a problem, regardless of tool condition, there were only a few occasions where they were unable to fix the problem for partial or full credit.

Table 6.7: Summary of results

	With tool	Without tool
Summary statistics, averaged per task over all cases		
Points scored	97.66	67.19
Time spent (minutes)	5.66	13.28
Average Efficiency (Points/Minute)	26.42	7.14
Successes at finding and fixing problems		
# successes at finding problems	64	45
# successes at fixing problems (full or partial credit)	63	43
# successes at fixing problems (for full credit)	60	39
Time taken (minutes), averaged over full and partial successes		
Finding the problem	1.09	3.56
Fixing the problem	1.62	1.92

Most of the difference in average total time between the two conditions is attributable to the time taken for finding problems. The time taken to fix the problems does not appear noticeably different between conditions, but that appeared to be due to the fact that the participants usually spent time reading about the potential transformation before applying it. In almost all cases, it appears that they read the whole description of why the smell was a problem, what the transformation was, and why that transformation would be effective at solving the issue. It is plausible that once a user used a transformation a few times, that there would be no need and little desire

to repeatedly read over this information. Therefore, the difference in total task time would probably be even larger over the long run in practice than what was observed during this experiment.

Since the points given for partial credit was arbitrary, a sensitivity analysis was done on the data to ensure that did not impact the final results. To perform this two new data sets were created. In the first set all partial credit solutions were given one point and in the second all partial credit solutions were given 24 points. Analyzing each set then showed what would happen if the minimum or maximum points were given for partial credit. The points and efficiency statistics were still found to be statistically significant regardless of the points given for partial credit. Both points data sets (high and low) had the same result (Mann-Whitney U=181 $p < 0.001$). For the efficient data sets, both were statistically different with the high points data set (Mann-Whitney U=29 $p < 0.001$) having a slightly different U value than the low points data set (Mann-Whitney U=31 $p < 0.001$). This analysis reveals that regardless of the points given for partial credits the results are still statistically significant.

Tables 6.8 and 6.9, below, present statistics at an even finer level of granularity, breaking down the differences separately for the easy and the difficult performance problems. Of the 32 times that participants had to find an easy performance problem without the tool, they failed 10 times (for a 69% success rate). It was primarily at this point where the large gap opened up between the effectiveness of participants with the tool and without the tool, since their success rates at fixing the problem are comparable between conditions. (This success rate includes partial successes.)

Table 6.8: Easy LabVIEW programs, detailed breakdown of results

	With Tool	Without Tool
Success Rate at finding problem	100% (32/32)	69% (22/32)
Average time for finding problem (Minutes)	0.96 (32 cases)	3.53 (22 cases)
Success Rate at fixing problem at all	100% (32/32)	95% (21/22)
Average time for fixing problem at all (Minutes)	0.53 (32 cases)	1.00 (21 cases)
Average Efficiency (Point/Minutes)	46.50 (32 cases)	15.84 (32 cases)

All of the potential transformations for the easy programs were full transformations, and for the most part the participants used them. Out of the 32 cases where a semi-automatic transformation was available, they were applied 30 times. In the other two cases, the participant read the description and decided to implement a different solution based on the feedback provided by the tool.

Table 6.9 shows results for the difficult LabVIEW programs, which presents a similar story to the one with the easy problems. Because three participants used up their entire allotted time struggling with the easy problem of a without-tool task, there were only 29 cases where a participant even began to attempt the difficult program of the task. They were able to find the problem only 23 times (for a 79% success rate, compared to the 100% success rate with the tool). As with the easy problems, it was again at this point where the large gap opened up between the two conditions.

Investigating the transformations, there were 16 programs with a partial transformation(***Build Array in Loop***), and 16 with a wizard transformations(***Terminals in Structure***). The average time to fix the ***Build Array in Loop*** with the tool's partial transformation was 4.12 minutes, whereas without the tool it was 3.97 min-

Table 6.9: Difficult LabVIEW programs, detailed breakdown of results

	With Tool	Without Tool
Success Rate at finding problem	100% (32/32)	79% (23/29)
Average time for finding problem (Minutes)	1.22 (32 cases)	3.59 (23 cases)
Success Rate at fixing problem at all	97% (31/32)	96% (22/23)
Average time for fixing problem at all (Minutes)	2.71 (32 cases)	2.83 (22 cases)
Average Efficiency (Point/Minutes)	18.63 (32 cases)	9.24 (29 cases)

utes. This was often due to participants reading the information provided by the tool to determine the steps necessary to complete the transformation. The average time to fix the ***Terminals in Structure*** with the wizard was 1.25 minutes whereas without the tool the average was 1.53 minutes.

In summary, participants found and fixed more performance problems in less time with the tool than without. These differences were primarily attributable to the tool's support for finding problems. The without-tool condition did not match or outperform the with-tool condition on any of the statistics computed for any of the tables above, including those related to fixing problems.

Keeping in mind that participants in the study were Application Engineers and experienced LabVIEW programmers, it is reasonable to expect that less experienced programmers would struggle more than these study participants at finding and fixing performance problems. Thus, the tool is likely to be even more beneficial to users in practice than it was in this study.

6.3.3 Qualitative Results

Qualitative observations and user feedback revealed important difference between when participants had access to the tool and when they did not. Namely, because participants found the problem 100% of the time when they had access to the tool, this allowed them to move on to fixing the issue and debugging the next program in the task. But lacking access to the prototype, there were several participants who got so confused trying to find an issue that they took the entire 20 minutes simply trying to discover what was going on with the program and why is was displaying poor performance, never moving on to the second (harder) program. After the study, when they were told what had been causing the problem, they groaned and commented that they knew that it could cause performance problems, but never thought to look for it. This feedback echoes the findings from the earlier study, that SDPA helped some participants remember their performance-related training. Similarly, in many instances in this study where the participants had access to the tool, they looked at the results and commented that it was a good reminder of what to look for in LabVIEW code, but that they would not have thought of it right away without the tool.

One interesting observation that arose during the study, was how infrequently Smell-Aided profiling was used. While every participant had access to the ability to profile for performance, there was only one case where it was used. This may have been because the tool's SDPA features led participants to the problem already, without any real need to run profiling. In almost all cases, the participants saw these

SDPA output and immediately focused on which of the identified smells most likely to be the real problem. Specifically, there were only three instances where participants got confused and began trying to solve a smell that was in the program but was not the cause of the performance problems. In these instances, running the profiling tool would have filtered the smells and informed them where to focus. There was one case where the participant got so far off track that this person opted to start debugging from scratch. After looking at the results from the tool again, this person immediately identified the problem and fixed it. The low use of the profiling tool suggests that it might not be needed in practice, on relatively simple programs.

In terms of the questionnaire, 31 of the 32 participants completed it and answered questions about the prototype. Table 6.10 summarizes statistics related to usefulness, ease of use, and potential for distraction.

Table 6.10: Summary of qualitative feedback about the tool as a whole

Felt the tool was useful	97%
Felt it made LabVIEW easier to use	94%
Felt the tool was not distracting	90%

A total of 30 of the 31 respondents felt that the tool was useful. More specifically, nine of these users highlighted the overall feedback, both in terms of finding the problem and the transformation, as the most useful aspect of the tool. Six participants thought the ability to quickly see the areas in the program that had problems was useful, and five felt that the tool highlighted the best practices that should be used when programming in LabVIEW. Four felt the semi-automatic transformations and the suggested changes was the most useful aspect, and three thought it was simply how

much time they could save while debugging code. Three mentioned that they thought that this tool would reduce the number of support calls that National Instruments received. The one participant who said it was not useful stated it might not be useful, given this person's skill level, but that it "would be helpful for people not as familiar with LabVIEW or the programming constructs."

In addition, 29 of the respondents felt that the tool made LabVIEW easier to use. By far the most common aspect (13 participants) to cause LabVIEW to be easier to use was the ability of the tool to pinpoint areas causing the performance problems in code. Six participants felt that the ability to receive feedback about a program made LabVIEW easier to use and would help to reduce the number of customers that would have to call for support. Three felt that the transformations would reduce difficulty the most. Two thought that the education potential of the tool would over time help make LabVIEW easier to users as they learned how to avoid pitfalls. Finally, two felt that the ability to remind users of best practices would make it easier to program in LabVIEW. Of those that said it made it more difficult, one stated that at times he did not know what the tool was referring to and thought having a help file that came with the tool would have been good. The other participant actually thought that making it more difficult was a good thing. He stated "I think that forcing a user to think about this, even if it makes it more difficult, results in better code and better outcomes for our customers, so I don't see 'more difficult' as a bad thing."

Only three participants found the tool to be distracting. In all of these cases, they had complaints about the amount of data that was presented to the user and felt like it was too much at times or that it took up too much space on the screen. There were

several users who cautioned about the use of the tool. Much like in the first study, one participant explicitly asked to avoid a Clippy like interface, stating: "Please do not make a paper clip that tells my VI is inefficient." In total, five users were worried about the tool giving information that they might not want or need, and most (3/4) said that having the ability to turn off or hide the results was a necessary addition.

In terms of other possible improvements, many users indicated a desire for better highlighting of the problem. While this is currently a feature in the tool, it can be subtle and hard to miss. In total 10 participants thought that the highlighting of problem areas could be improved. Many thought that drawing a colored box around the area would be beneficial and make the tool easier to use.

In addition, the tool UI describing the smells was often not large enough to fit all of the information, particularly for the more complex smells. To see everything, the participants had to either scroll through the frame, or pop the frame out so that it hovered over the main IDE. Neither of these were optimal, and some participants struggled with how to best use this and how to see all the information that they wanted. This did not seem to deter them from using the tool, or from successfully debugging the program, but it is something that should be looked at in the future.

One final weakness in the system that was observed, and commented on by several participants, is that if there are multiple smells of the same type they are all shown to the user. In one program, there were four instances of the ***Uninitialized Shift Register*** smell. Several participants noted that it would be better to only see a single entry in the table that summarizes the smells detected in the program. Once the smell was clicked it could potentially highlight all instances of that smell in the

program (or open up like a tree control). In addition, they had to click on each smell to apply the transformation for that instance. Since all the transformations were the same, the participants preferred the option to be able to fix all instances of a smell at the same time.

Chapter 7: Conceptual Generalizability

The methods detailed in Chapters 4 and 5 were implemented in another graphical language in order to understand whether the ideas generalized beyond LabVIEW. When SDPA was implemented in LabVIEW, experts were interviewed to gather a set of patterns that frequently caused problems for end-user programmers. With another dataflow language oriented toward end-user programmers, Yahoo! Pipes [38], researchers have already taken the time to become experts at using the programming language and have published papers on ways to improve performance in programs. This existing research offered a basis for examining how well the smell-based approach described in this dissertation can conceptually encompass existing approaches. This existing work was used as a foundation to determine a set of problems that could be recognized as smells. Once this set of patterns was collected, a prototype were implemented to run on Yahoo! Pipes programs to detect the existence of smells and to apply transformations to fix them. The subsections below present some background on Yahoo! Pipes, the existing research on performance in this language, and the tools that were prototyped to find and fix smells that were defined based on this existing research.

7.1 Background on Yahoo! Pipes

Yahoo! Pipes enables users to create programs that can retrieve, transform, and/or combine web data sources to produce a feed of items [38][97]. For example, a typical Pipe might retrieve a list of articles from an online newspaper, retrieve another list of items from a second online newspaper, merge the lists, filter them to only include news that mentions Microsoft in the title, and then displays the result. Figure 7.1 shows a Yahoo! pipe that fetches many different feeds to gather information about the iPhones and iPod Touch and display it all in one output.

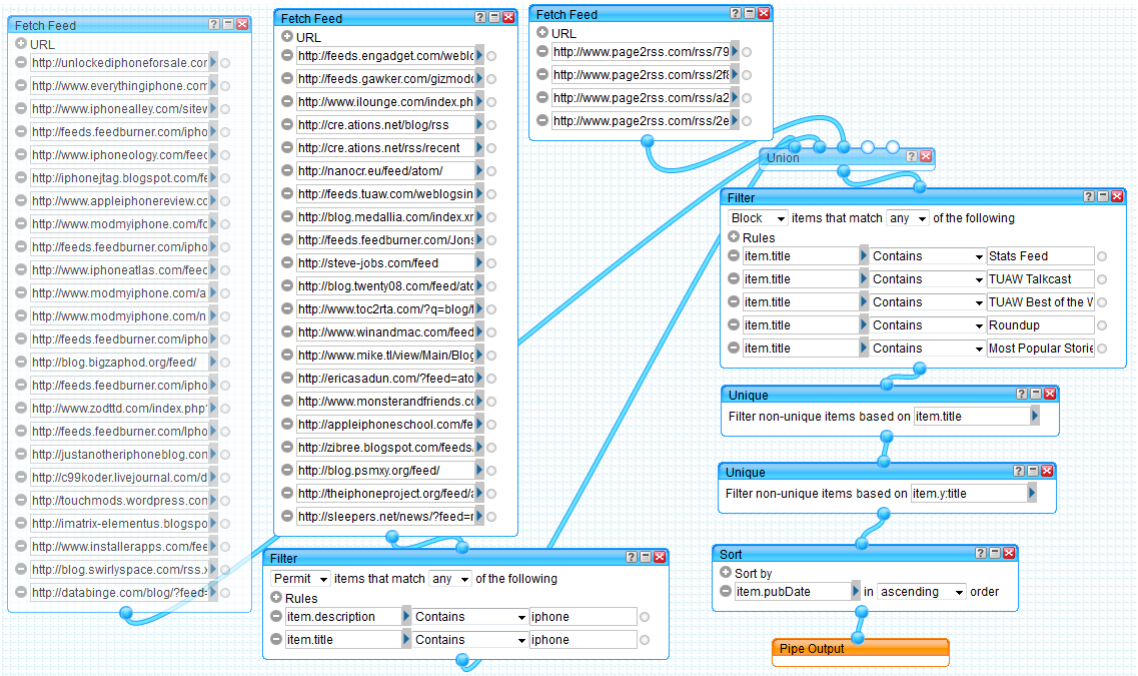

Figure 7.1: A Yahoo! Pipe example that gathers data from many feeds

The language offers a variety of primitives for operating on data, a few of which are shown in Figure 7.1. In this figure, the **`Union`** module combines all of the fetched

feeds into a single stream so that they operated on at the same time. Once this is done, the Pipe applies a **Filter** to the data to remove unwanted information. The **Filter** module can be configured to either block certain information as shown in Figure 7.1 after the **Union** or to only permit specific information, which happens in the **Filter** before the **Union** (bottom middle in the Figure). Another form of filtering, **Unique** is applied, which will remove duplicates. After all the filtering has been performed, the program sorts the data with a **Sort** module. This module will reorder the data by the specified argument, in this case publication date.

The other popular operators that are found in Yahoo! Pipes are **Reverse**, which will flip the order of the data, **Sub-Element** which will return only one element for every item in the feed, and **Regex**, which is used to find and replace strings in the feeds with a user specified replacement.

As with LabVIEW, performance can sometimes become a concern with Yahoo! Pipes. Recent research [58] analyzed mashups and found that very few of them will perform optimally. Previous research [90] has shown that 81% of pipes have a deficiency. While these deficiences are not necessarily related to performance, they do show that users struggle to create programs that execute optimally. Around 66% of these pipes are reused [90], and many pipes are designed to run hundreds (or more) times in a single day [59][100]. This means that existing performance problems can be exacerbated over many executions or perpetuated through reuse. Finally, many mashup systems have a limited amount of resources due to the online nature of their interface [34] meaning that even "small" performance problems can cause significant impact.

Yahoo! Pipes is a suitable context for testing the generalizability of SDPA and associated tools not only because of the concerns about Yahoo! Pipes performance, but also because the language is a visual dataflow language, like LabVIEW. Therefore, it potentially accommodates the general approach discussed in Chapter 3. Specifically, programs created in this language can be parsed into a structure that can be statically analyzed to identify specific combinations of program primitives, which could potentially correspond to performance smells. In addition, the parsed structure can be modified, potentially offering a means of semi-automatically eliminating smells. Finally, the visual representation of the code as a dataflow program offers locations on the screen where icons or other warnings could potentially be displayed.

7.1.1 Existing work on Yahoo! Pipes performance

Two specific projects have investigated performance in Yahoo! Pipes [34][100], while another [90] investigated maintainability smells that may influence performance. Another project [59] is designed to refactor specific patterns to optimize performance, but does not notify users as it simply performs the changes automatically.

The first of these projects [34] aims to improve performance by applying two heuristics. One heuristic works to remove common components or merge them together. The second checks a pipe to determine if it is in canonical form, which represents the best possible format for the optimal performance of specific modules. One example of this is that a filter module should always occur before a sort module to reduce the size of the feed. These heuristics were then applied to several data sets

that contained 2000-10000 mashups. The researchers found that these heuristics were able to reduce the delay during the execution, with the claim that delay was reduced by 2550 seconds over the entire set of mashups. The specific baseline and percent improvement were not given.

The second of these projects [90] focuses primarily on finding maintainability smells in Yahoo! Pipes, but some of these smells might have implications on performance. This work created a tool that can search a Pipe for a set of smells and then apply a transformation. However, no performance metrics were measured, so it is not clear if any of these smells impact performance. This work did show that by cleaning up the maintainability smells in a pipe, the program is easier to understand and read, which can be important due to the high number of pipes that are reused. Upon careful examination, several of the smells that this tool detected may impact performance, particularly the smells that focus on redundant code and combining operators. Overall, it was found that 81% of the investigated 8,051 Yahoo! Pipes had some form of deficiency (had a smell detected).

The third of these projects [100] focuses on creating a performance metric and scheduling framework in improve the memory utilization of mashups, such as Yahoo! Pipes. This approach first analyzes the original mashup for the memory and time used, and calculates the Product of Memory and Time (PMT). The scheduling framework, which implements lazy starts for modules, is run to try and find the best possible combination of starting times for specific modules so that PMT will be minimized (the program will use less time and/or memory). By picking the correct time to start certain modules, this research showed a performance improvement of 29% by

avoiding blocking time and wasteful memory consumptions.

The final project [59] aims to create the optimally performing version of a pipe by analyzing a program and refactoring it automatically to generate several different versions of the pipe. These versions are compared to determine the most optimal. The researchers introduce an algorithm, Optimal DataFlow Selection (ODFS), that will apply a set of six refactorings to the pipe and then analyze the resulting versions to determine the ones that executed optimally (best runtime). This was then tested on a set of mashups that they created containing different numbers of modules to test the system. The version that was selected as optimal was compared to the original to determine the difference in run time. In one example, they were able to show a 41% improvement, which may have partially been due to the feed size of 10,000 elements.

7.2 Smells for Yahoo! Pipes

The first two of the projects above [34][90] provided several insights about performance problems that could be detected with four smells discussed in detail below. The third approach above required modifying the scheduler [100], meaning that it aimed at improving the Yahoo! Pipes environment rather than the end-user programmers' code within that environment, so no smells were derived from that project. Because smells could be defined (below) that encompassed two out of three of these existing approaches, this indicates at least at a conceptual level that the smell-based approach can subsume some but not all of existing approaches aimed at improving performance in Yahoo! Pipes.

7.2.1 Delayed Filtering

One of the primary tasks in most Yahoo! Pipes is to gather data from feeds and modify the resulting data. Once the data is obtained, users can apply filters to it to reduce the data to contain only what they desire.

Hassan et al. [34] showed that filters can unnecessarily create performance problems when they are applied later than necessary. To take a simple example, suppose a news feed returns N items that are then passed through two other modules, A and B. Module A performs some operation on the items such as sorting, and Module B is a **`Filter`** that removes approximately 90% of the items. Then passing data through Module B and then Module A will offer much better performance than passing data through Module A and then Module B. The reason is that if B filters first, then the other module only needs to operate on 10% of the items. This could produce a sizable performance improvement, especially since most sorting algorithms are slower than O(N). This smell was also mentioned in [59] as a pattern that they recognized as problematic. They showed one instance where moving the **`Filter`** module earlier improved performance by 41%.

Figure 7.2 shows a slightly more complex example of a real-world program exhibiting a similar performance problem. In this case, the pipe is filtering the collected feeds by a publication date. However, the filtering is happening after the feeds have been joined together and sorted. By applying the **`Filter`** after the **`Union`** and the **`Sort`**, the pipe is being forced to apply those operations to the entire, unfiltered data set, which increases the work that it must perform.

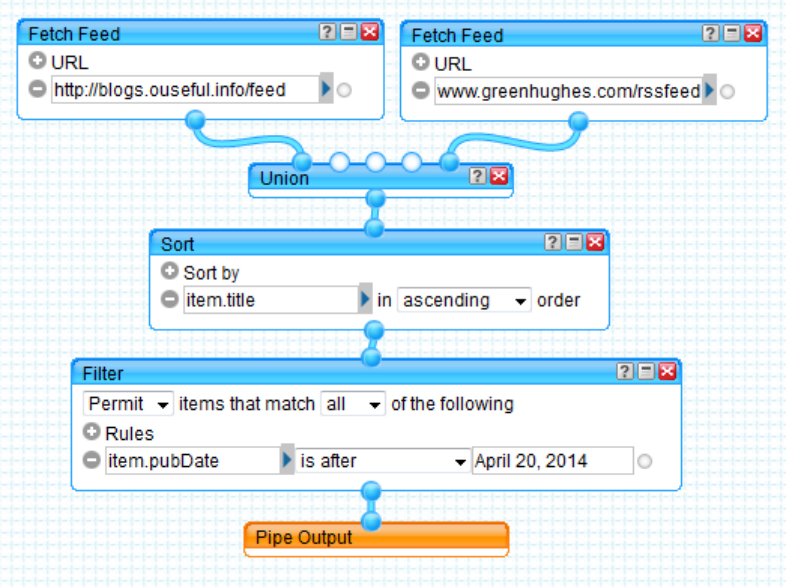

Figure 7.2: A Yahoo! Pipe containing a Filter module after a Union and a Sort module

In general, this smell can be recognized as any situation where a **`Filter`** module consumes data after one or more modules that operate upon items in a feed but that do not actually change any content that a filter might recognize. Based on the information given in [34], there are four modules that **`Filter`** should be moved in front of: **`Union`**, **`Sort`**, **`Reverse`**, and **`Sub-Element`**.

The transformation to fix an instance of the smell is to move the **`Filter`** module above these modules that may operate on the dataset. This may require cloning the filter module so that it can be applied to multiple incoming feeds. For example, Figure 7.3 shows the transformed program (from Figure 7.2) after the **`Filter`** module has been moved.

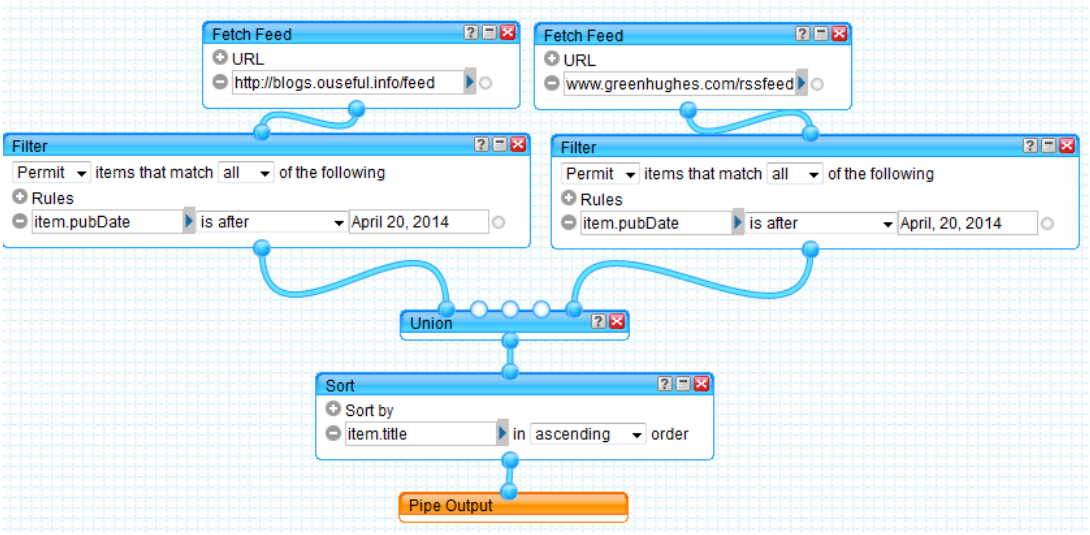

Figure 7.3: The transformed Yahoo! Pipe with the Filter module now located before the Union and Sort module

7.2.2 Duplicate Idempotent Operators

The second smell identified based on related work is having ***Duplicate Idempotent Operator*** modules. An idempotent function is any function that, if applied multiple times, produces the same result after the Nth application as it did after the first. Within Yahoo! Pipes, the idempotent operators are **`Sort`**, **`Union`**, and **`Filter`**. Having duplicates of these modules in a pipe can cause the program to take longer to execute, as it is running modules multiple times with no additional effect. This smell was motivated by [90], which reported that 23% of pipes have duplicate models.

As an example, Figure 7.4 shows a Yahoo! Pipe that has two **`Sort`** modules that are not required. In this instance, they are exactly the same and the second one can be removed. It should be noted that if the second **`Sort`** module was sorting by a different parameter, then both modules need to remain.

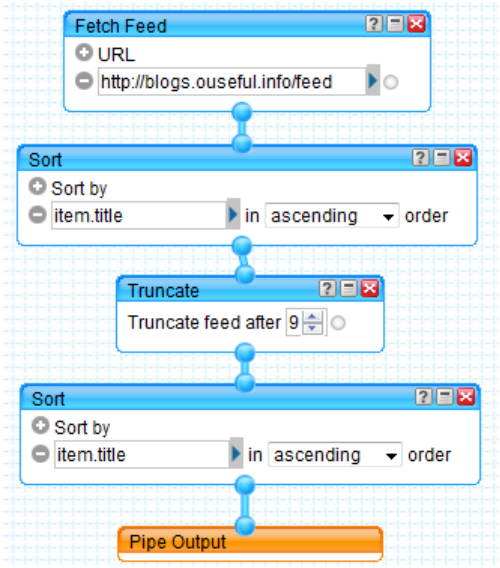

Figure 7.4: Yahoo! Pipe showing with subsequent sort operations. The second sort is not necessary

In general, this smell can be recognized as a case where the idempotent operators appear more than once in a given path within the dataflow graph. The transformation for this smell is to simply remove the second instance of the duplicate module.

It is important to note that this transformation could potentially alter the output of a program, if any module in between the duplicate idempotent operators have any side-effects that modify the effects of the second instance of the operator. For example, suppose that a program applies a **`Sort`**, then a **`Regex`** that alters the field used for sorting, then the same **`Sort`** again. Removing the last **`Sort`** operation would alter the program output. In general, most if not all of the operators in Yahoo! Pipes are not idempotent; therefore, if any of them are in between the two duplicate idempotent operators, it is possible that side-effects could occur that alter the program output.

Consequently, if a tool for this transformation were offered to users, it would not be offered in a fully automated form. Instead, it would be offered as a semi-automatic transformation that the user can decide whether or not to apply, depending on whether the resulting alterations to the output would be problematic. (This is very similar to the Wizard Transformations that were introduced for LabVIEW in Section 5.4, which address smells that do not have a fully automated fix but rather offer an interactive tool guiding the user to a possible performance solution.) For example, the tool could present a series of screens that explain the smell, suggest a transformation, show the effect of the transformation, and ask the user whether to retain the transformed code or instead to revert to the original, untransformed program.

7.2.3 Combinable Operators

A related smell is the redundant use of modules that can be combined. One example of this is two **`Regex`** modules being applied in a row. Instead they could be combined into one multi-rule module. This smell can occur with the **`Regex`**, **`Sort`**, and **`Filter`** modules. This smell was created based on the work done in [34] which detected common components and found that combining those components could decrease the delay experienced. This smell was also one that was refactored by ODFS [59].

Figure 7.5 shows a snippet of a Yahoo! pipe that contains the ***Combinable Operators*** smell. In this example, there are three **`Filter`** modules that operate back to back. All of these modules are blocking any element of the feed that matches

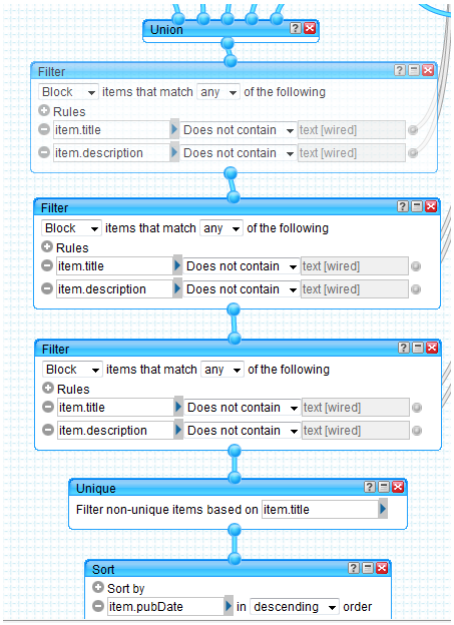

Figure 7.5: Yahoo! Pipe showing with combinable filter operations. All three filter modules can be combined

certain criteria, which are set when the pipe is executed.

The smell can be detected any time that the output of a combinable module of a particular type (**`Regex`**, **`Sort`**, or **`Filter`**) is fed directly into the input of a second module of the same type. For example, a **`Filter`** module that *blocks* any element that matches a criteria and **`Filter`** module that *permits* any element that matches a criteria cannot be combined. The transformation for this version of the smell is to combine the two redundant modules and use the arguments for both in the new version. Figure 7.6 shows the transformation applied to the above pipe.

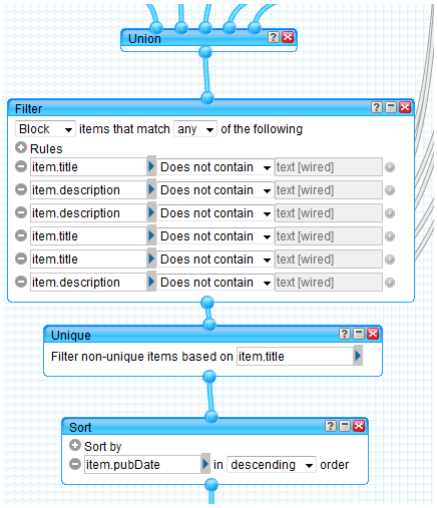

Figure 7.6: Yahoo! Pipe showing with combined filter operations. All three filter modules have been combined to one

7.2.4 Noisy Modules

The ***Noisy Modules*** smell involves checking the modules in a pipe for any unnecessary fields. This smell was mentioned in [90] and makes pipes harder to read and less efficient to execute. This smell is detectable by looking at the fields of a module and determining if they are 1) empty or 2) duplicated. Figure 7.7 shows an example of two modules that exhibit this smell. The **`Fetch Feed`** module (on top) contains two feed fields that are the same meaning that the feed is fetched twice. The **`URL Builder`** module (on bottom of image contains an empty field, in this case the field that says text indicating that text can be added if desired.

The transformation for this smell is to remove the useless field(s). For example, the **`Fetch Feed`** module in the above figure would only have the first two fields after the transformation and the **`URL Builder`** module would have no field under "Path

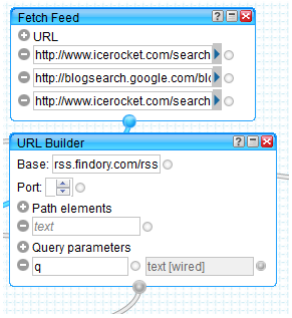

Figure 7.7: Yahoo! Pipe snipper showing two Noisy Modules. The Fetch Feed module (top) contains a duplicate field, while the URL Builder module (bottom) contains an empty field

Elements".

7.3 Prototype

A prototype tool was implemented that could find and fix the performance smells described above. To create the tool, the framework that was used in [90] was acquired[1] and modified to detect the performance smells discussed above.

This framework collects the JSON representation of a pipe by sending a *load pipe* request to the Yahoo! Servers for a specified Pipe ID. For this prototype, detectors were created for the four performance smells that would look through the JSON for specific patterns and print out which smells were detected and which modules were affected.

As an example, the ***Delayed Filter*** smell would be found by looking at a **`Filter`** Module and finding the parent. The module type of the parent is checked to determine

[1] $http://www.cs.iastate.edu/\ kstolee/Refactoring_Mashups.html$

if it is one of the four offending modules. If it is, then the a smell has been detected, otherwise, the program will check the parent of that module and continue checking until a source module, most likely a **`Fetch`**, is reached. The last module that is either a **`Union`**, **`Sort`**, **`Reverse`**, or **`Sub-Element`** will be noted for use in applying transformations.

For each of the four smells, a transformation was created that would remove the problem. Since ***Noisy Module*** was an already existing smell in this framework, the detection and transformation was not changed. Once the transformations have been applied, the JSON representation can be reloaded to the Yahoo! servers so that an actual pipe will be created. This pipe can then be run to determine the performance improvements from the transformations.

7.4 Running the Prototype

To verify that the prototype was able to successfully run on Yahoo! Pipes programs, 100 programs were collected to test from the "Popular Pipes" section of the Yahoo! Pipes repository[2]. To gather 100 valid pipes, 108 were retrieved, as 8 of the pipes contained modules that no longer existed and would not load.

7.4.1 Smell Prevalence

The prototype performed well on all of these test cases. The SDPA prototype was able run successfully on 100% of the pipes in the this collection, within which it detected

[2]http://pipes.yahoo.com/pipes/pipes.popular

97 smell instances. In total, 34 of the 100 Pipes had a smell that was detected. For each smell detected, the tool printed out the module that was affected as well as the smell that was detected. Table 7.1 details the prevalence results for each detectable smell.

Table 7.1: Prevalence of smells detected by SDPA in 100 Yahoo! Pipes

Smell	**Total Number of Instance**	**Unique Pipes with Smell**
Noisy Module	45	15
Delayed Filter	22	15
Combinable Operators	19	7
Duplicate Idempotent Operators	11	4
Total	97	34

Noisy Module was by far the most detected smell (46% of all smells), but the average number of instances per pipe (3) was high and thus it occurred in the same number of unique pipes as ***Delayed Filter***. In total, these two smells made up 69% of the total smells that were detected by the SDPA prototype.

Combinable Operators also had a high average number of instances per pipe (2.7), and it was rare to find this only once in a pipe. ***Duplicate Idempotent Operators*** was detected 11 times, though this high number was due to 7 instances occurring in one pipe.

7.4.2 Performance Impact of Transformations

Once the smells were detected, the applicable transformations were applied to fix each smell. This was done to each individual smells to determine the impact of each

specific transformation. The new pipe was then run to determine the impact of the transformation. For this evaluation, only execution time was gathered, since Yahoo! does not provide memory information for pipes.

Table 7.2 shows the average performance impact for each individual smell instance. ***Delayed Filter*** had the largest impact on execution time with nearly a 44% increase on average. ***Noisy Module*** also showed great improvement when the transformation was applied. The others had less of an impact, with ***Combinable Operators*** only improving performance by 9% on average.

Table 7.2: Performance Improvements of transformations on detected smells in 100 Yahoo! Pipes

Smell	**Average Performance Improvement**
Delayed Filter	44%
Noisy Module	34%
Duplicate Idempotent Operators	18%
Combinable Operators	9%
Average over all smell instances	29%

As mentioned in the previous subsection, the top two smells, ***Delated Filter*** and ***Noisy Module***, made up 69% of the detected smell instances. These smells were also the top two in terms of average performance improvement.

Combinable Operators improved performance by 9%, and the majority of the instances barely improved the performance at all.

7.4.3 True and False Positives

The smell instances detected by the Pipes SDPA prototype were also analyzed to determine how many were false positives and did not actually impact the performance of the pipe. A detected smell was said to be a false positive if it had less than a 15% impact on the execution time of the pipe. Table 7.3 shows the results of this analysis.

Table 7.3: False Positive smells detected by SDPA in 100 Yahoo! Pipes

Smell	**Total Instances Detected**	**False Positives**	**False Positive Rate**
Delayed Filter	22	7	32%
Noisy Module	45	12	27%
Duplicate Idempotent Operators	11	4	36%
Combinable Operators	19	14	74%
Total False Positives	97	37	38%
Total False Positives without Combinable Operators	78	23	29%

The top three smells in this table all had fairly similar numbers in terms of false positive rate, however the final smell, ***Combinable Operators*** did not fare well. 14 of the 19 detected instances were false positives, and 12 impacted the overall performance by 3% or less. Combining operators still required the pipe to perform the actions, and evidently the time saved by removing a module was minimal.

The overall false positive rate was 38%. It is possible that this number could be reduced further by smarter smell detection. For example, the false positives for ***Delayed Filter*** were primarily caused by **`Filter`** modules that were using the "permit" argument instead of "block". It appeared that these modules did not block

many elements in the feed and so moving the filter has less impact on the overall performance.

7.4.4 Evaluation Summary

The successful application of smell-based analysis and transformations, to detect program structures known from earlier research to be problematic from a performance standpoint, shows that the methods described in this dissertation can encompass these existing approaches for detecting and correcting performance problems in end-user dataflow programs.

The SDPA prototype for Yahoo! Pipes was able to detect performance smells and apply transformations to quickly fix the issues. Overall, the tool was able to improve the pipes by an average of 29%, which is comparable to the improvement in [100] (28.65%). However, the SDPA prototype does this without requiring the scheduler to be rewritten and puts the power in the hands of the users by allowing them to choose which smells should be fixed.

Chapter 8: Conclusions and Future Work

Performance problems in data flow code are very difficult to debug for end-user programmers who lack the experience and knowledge to correctly find and fix the problems. These people have few options for assistance other than expensive customer support or posting on their questions on forums that have a low rate of solutions and a highly variable response time. This thesis has described Smell-Driven Performance Analysis, which detects common performance smells and gives end-users the ability to quickly and efficiently fix the issues.

This work modified the idea of maintainability smells and, drawing upon the knowledge of expert LabVIEW programmers, developed the idea of performance smells. These smells were used to detect common problems and to aid end-user programmers in finding regions of the code that may be causing poor performance. Once these smells have been detected, semi-automatic transformations were provided that help users fix these issues.

Several evaluations performed on real world examples showed that the smells detected by SDPA are prevalent in real LabVIEW code and that SDPA missed very few of the optimizations performed by experts. In addition, SDPA showed a false positive rate of only 32%, and the number of false positives was reduced by 50% with the introduction of Smell-Aided Profiling.

Real world examples were also used to evaluate the transformations that are sup-

plied by SDPA and the performance improvements that they create. The transformations on average improved the execution time by 42% and the memory use by 20%. This was comparable to the 46% and 28% improvements shown by the optimizations performed by experts. This indicates that SDPA can achieve nearly the same performance improvements as experts, without requiring an expert to look at the code.

SDPA was also evaluated through two user studies to determine how well it helped actual users find and fix performance problems. Participants using the tool were able to find problems quickly, and most noted that the ability to pinpoint potential problems saved time. In terms of the total debugging process, participants using the tool were over 3 times more efficient when debugging performance problems then those using the normal LabVIEW IDE.

Finally, it was shown that SDPA is also applicable in other dataflow languages, as smell detection and transformations were implemented in Yahoo! Pipes. This case study showed that the ideas described in this thesis are generalizable and can be implemented in languages similar to LabVIEW.

8.1 Future Work

There are two types of work that could be pursued based on the research presented in this thesis. The first type is incremental changes to the tool or methods, while the second is more ambitious and looks at different ways this research could be applied to other languages or in other situations.

8.1.1 Incremental Improvements

There are several small areas that could be done to improve the tool and the methods. While these suggestions are incremental improvements on the existing work, they are based on discoveries made during the evaluations and would improve the tool and likely make it more effective at helping End-users.

8.1.1.1 Studying Adoption

One aspect of creating new tools is that users may not adopt them. Users often have techniques to debug their programs and it may be hard to get them to try new methods that they aren't sure of. A further study could investigate if users adopt the tool into their daily work practices, as well as finding ways to aid in the adoption process.

To investigate this, a field study could be performed where the LabVIEW IDE with SDPA is given to users for a month. During this time, users actions would be tracked to determine when they used the tool as well as how helpful they found the results to be. Once the study is completed, the data could be analyzed to determine if it was adopted by the participants. In addition, participants could be asked about the tool, and better ways to make the results noticeable and easy to understand.

This is particularly important since there were cases in the user studies where participants did not use the tools, especially Smell-Aided Profiling. This study would definitively show when certain parts of the tool are used as well as cases when they would be beneficial but are not used. This information could improve the user in-

terface by revealing better ways to notify users or which notification methods do not work.

8.1.1.2 Does this aid Learning?

During the complete tool study, it was clear that most of the participants took the time to read the information the tool provided on a given smell, as well as the possible transformation. One hypothesis is that providing this information to users will help them learn about the potential pitfalls while programming and help avoid these problems in the future. This could be evaluated with a field study that tracks users over time (perhaps one month) to see if the use of known bad smells decreases over time after exposure to the information provided by the tool.

8.1.1.3 Improved User Interface

In the user studies, there were several suggestions from participants to improve the highlighting of the smells. Currently, when a user clicks on the smell instance in the table it flashes briefly in the program and then remains selected (indicated by a dashed line around the node). Several participants felt that this was not enough and that it could be made easier to find the smells. One suggestion was to draw a colored, slightly opaque box around the node or nodes causing the problem; another was to make the current action last longer and be more pronounced so that it stood out. While this did not hinder participants from finding the smells, it could make it

easier for users to notice the smells in a given program.

Another improvement that could be made is to the smells table. In the prototype all the instances of a smell were given to the user. In some cases this meant seeing several identical messages, all for a different node. Clicking on one of the messages would highlight one node, and to find the location of all such smells the messages had to be handled separately. Participants felt that this could be streamlined by having one message for all instances of a certain smell. Clicking on that message would highlight every location of that smell in the program. The message could be expanded to select a single instance if desired. In addition, participants mentioned that having the ability to apply a transformation to all instances of a smell would decrease the amount of time spent fixing the problem, so this could also be a worthwhile addition.

8.1.1.4 Better Profiling

While profiling was not widely used in the study, there is still a place for it, particularly for programs that will be run on real time systems or FPGAs where the amount of memory is limited and every little bit matters. There are improvements that could be made to the profiler that may improve its effectiveness. One issue is caused by LabVIEW's inherent parallelism. Consider the example of two loops that execute in parallel. If one of the loops contains a smell, the data gathered by the timing probes will include timing and memory information for the execution of both loops (since they operate in parallel).

Another profiler improvement that could be implemented is to add profiling within

structures. Currently, the prototype does not profile inside of structures. This means that if a structure has two smells, one that causes a problem and one does not, the profiler will not be able to distinguish that one of those smells is a false positive and will report both to the user. Profiling inside the structure could indicate the actual smell that caused the problem, further reducing the false positives detected.

8.1.1.5 User Added Smells

One aspect that could be very beneficial would be allowing users to create their own smells and transformations. This would allow experts to create pattern/transformation combinations that could be installed by novice users. This would allow the tool to be highly flexible in the types of problems that could be detected and fixed.

8.1.1.6 Integrating Transformations with the Compiler

It is possible that some of the transformations performed by the tool may be more applicable if they were automatically applied by the compiler. This is particularly the case for the transformations that didn't require any input from the user to be applied. For example, when ***Redundant Operations*** are detected, it may be simpler to have the compiler remove them from the execution rather than inform the user.

It would be worth investigating what percentage of transformations could be automatically applied by the compiler and what extent this changes the performance of the program. It is also possible that this removes too much say from the user. Many

of the transformations are not always wanted or needed. For example, a user may want to run through an array as fast as possible with a loop. Having the compiler automatically add a wait would mean that would no longer be possible. Future research could investigate which smells could be automatically transformed with the compiler and which require a users decision to be applied to the code.

8.1.2 Further Research Applications

In addition to the incremental extensions described above, there are several applications for this research that could prove fruitful.

8.1.2.1 Extension to other Graphical Languages

While the prototype for this research was created in LabVIEW, the generalizability evaluation showed that these methods can be applied to another graphical language. Future work could look at the best way to implement these ideas in those languages. Creating tools that could help end-users find and fix performance problems in a variety of languages would have the potential to help end-users create programs that run more efficiently and give them the ability to do their jobs quickly and easily without having to worry about their programming skills.

8.1.2.2 Checking for Additional Types of Problems

In addition to extended this research to other languages, it could also be extended to other types of problems. Performance problems was one area where little research had been done and was ripe to be investigated, and there are several similar areas.

The first is checking for misuse of Dataflow concepts. Many LabVIEW users reportedly try to create programs that are similar in style to textual code. This frequently means that the programs they create do not take advantage of the unique aspects of the language and make them harder for understand and read. The tool could be extended to check for a set of patterns that indicate poor dataflow programming. To a certain extent, the existing tool checks for this as the smells ***Too Many Variables*** and ***Sequence Structures*** are often found when a user is trying to code in a manner similar to textual languages. Talking to AE Specialists could reveal more patterns that violate dataflow concepts or decrease readability or understandability. These patterns could then be detected using the methods in this thesis and transformed at the users' behest.

A second type of problem that could be checked for using this method is unnecessary code. On the National Instruments LabVIEW forum, experts often poke fun at "Rube Goldberg Code", which are programs that they have found that use LabVIEW very inefficiently. They will show an example program and then provide a much simpler solution that has the same end result. It may be possible to mine these cases for additional patterns to detect. If some of the "Rube Goldberg Code" cases could be generalized, they might be a powerful way of showing novice users how not

to program in LabVIEW.

Finally, this research could be extended to look at LabVIEW code designed to run on an FPGA. The AE Specialists would often mention that some smells were far more important for code designed to run on a real time systems or a FPGA because the available memory is much lower. It is likely that modifying the tool to check for specific patterns that are problematic in FPGA code could be very useful.

In some ways, broadening the types of problems that are caught also broadens the number of users who may find a tool like this useful. Checking for dataflow concepts is more likely to be helpful for novice LabVIEW programmers, while looking for patterns that are problematic for FPGA code would mostly be helpful for more advanced users. This presents challenges in terms of designing an interface that all users would feel comfortable with, and one that would be helpful without being intrusive or Clippy-like.

8.1.2.3 Aiding Reuse

While the main goal of this research was to find performance problems in an individuals code, there is no reason why it could not be extended to find performance problems in other code available to LabVIEW users. In particular, if a user wants to reuse a LabVIEW program, or a snippet, they could first check it for smells to determine if it should be added. This could be used as a guide to help users find code that would perform well if reused. In addition, the tool could be applied to any large snippet that is pasted or imported. It could quickly be run to notify the user if they

have added (or pasted) code that will likely perform poorly.

This idea could easily be extended and applied to an entire repository of LabVIEW programs. When the programs are uploaded to be shared they could be checked by the tool for any smells that might exist. This would help to determine the quality of a given program and help users pick programs that would not perform poorly. Currently, there are no online repositories that LabVIEW users can access, so creating one that has the ability to define the quality of a program as it is uploaded could be very beneficial.

Finally, it may be possible that the methods could be modified, after talking to experts, to detect areas in the code where it would make sense to reuse other code or create subVIs. There may be code patterns that experts recognize that could be replaced by existing design patterns or snippets. Once detected, the tool could suggest a semi-automatic transformation that would replace the existing code with a snippet that has been designed by National Instruments and is known to behave properly and perform optimally.

Bibliography

[1] Walid Abdelmoez, Essam Kosba, and Ali Falah Iesa. Risk-based code smells detection tool. In *The International Conference on Computing Technology and Information Management*, pages 148–159, 2014.

[2] Atipol Asavametha. Detecting bad smells in spreadsheets. Master's thesis, Oregon State University, 2012.

[3] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 307–320, 2012.

[4] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, et al. Analyzing parallel programs with pin. *Computer*, 43(3):34–41, 2010.

[5] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[6] Don Batory, Christian Lengauer, Martin Odersky, and Charles Consel. Domain-specific program generation. 2004.

[7] Barry W Boehm. Software engineering economics. In *Pioneers and Their Contributions to Software Engineering*, pages 99–150. Springer, 2001.

[8] Michael D Bond, Katherine E Coons, and Kathryn S McKinley. Pacer: proportional detection of data races. *ACM Sigplan Notices*, 45(6):255–268, 2010.

[9] William J Brown, Hays W McCormick, Thomas J Mowbray, and Raphael C Malveau. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley New York, 1998.

[10] Margaret Burnett, Andrei Sheretov, Bing Ren, and Gregg Rothermel. Testing homogeneous spreadsheet grids with the" what you see is what you test" methodology. *IEEE Transactions on Software Engineering*, 28(6):576–594, 2002.

[11] Xing Cai, Hans Petter Langtangen, and Halvard Moe. On the performance of the python programming language for serial and parallel scientific computations. *Scientific Programming*, 13(1):31–56, 2005.

[12] Jeffrey C Carver, Richard P Kendall, Susan E Squires, and Douglass E Post. Software development environments for scientific and engineering software: A series of case studies. In *Proceedings of the 29th International Conference on Software Engineering*, pages 550–559, 2007.

[13] Chris Chambers and Martin Erwig. Reasoning about spreadsheets with labels and dimensions. *Journal of Visual Languages & Computing*, 21(5):249–262, 2010.

[14] Chris Chambers, Martin Erwig, and Markus Luckey. Sheetdiff: A tool for identifying changes in spreadsheets. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 85–92, 2010.

[15] Chris Chambers and Christopher Scaffidi. Struggling to excel: A field study of challenges faced by spreadsheet users. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 187–194, 2010.

[16] Chris Chambers, Zachary Sommers, and Christopher Scaffidi. A study of help requested online by spreadsheet users. *Journal of Organizational and End User Computing (JOEUC)*, 24(4):41–53, 2012.

[17] Christopher Chambers and Christopher Scaffidi. Smell-driven performance analysis for end-user programmers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 159–166, 2013.

[18] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed N Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1001–1012, 2014.

[19] Keith D Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 2002.

[20] Jácome Cunha, João P Fernandes, Hugo Ribeiro, and João Saraiva. Towards a catalog of spreadsheet smells. In *Computational Science and Its Applications*, pages 202–216. Springer, 2012.

[21] Jácome Cunha, Joao Paulo Fernandes, Pedro Martins, Jorge Mendes, and Joao Saraiva. Smellsheet detective: A tool for detecting bad smells in spreadsheets. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 243–244, 2012.

[22] Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. Relooper: refactoring for loop parallelism in java. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 793–794, 2009.

[23] Timothy J Duesing and John R Diamant. Codeadvisor: Rule-based c++ defect detection using a static database. *HEWLETT PACKARD JOURNAL*, 48:19–21, 1997.

[24] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, volume 10, pages 1–16, 2010.

[25] Martin Erwig, Robin Abraham, Steve Kollmansberger, and Irene Cooperstein. Gencel: a program generator for correct spreadsheets. *Journal of Functional Programming*, 16(3):293–325, 2006.

[26] Gerhard Fischer. A critic for lisp. In *Proceedings of the 10th international joint conference on Artificial intelligence-Volume 1*, pages 177–184, 1987.

[27] Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. *Communications of the ACM*, 53(11):93–101, 2010.

[28] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[29] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[30] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[31] Scott Grant and James R Cordy. An interactive interface for refactoring using source transformation. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects*, pages 30–33, 2003.

[32] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, pages 145–155, 2012.

[33] Anna Harutyunyan, Glencora Borradaile, Chris Chambers, and Christopher Scaffidi. Planted-model evaluation of algorithms for identifying differences between spreadsheets. In *Proceedings of the IEEE Symposium onVisual Languages and Human-Centric Computing*, pages 7–14, 2012.

[34] OA-H Hassan, Lakshmish Ramaswamy, and John A Miller. Enhancing scalability and performance of mashups through merging and operator reordering. In *Proceedings of the IEEE International Conference on Web Services*, pages 171–178, 2010.

[35] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 27–38, 2013.

[36] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 441–451, 2012.

[37] Shan-Hsi Huang and Jan M Rabaey. Maximizing the throughput of high performance dsp applications using behavioral transformations. In *Proceedings of the European Conference on Design Automation*, pages 25–30, 1994.

[38] Yahoo Inc. Yahoo! pipes. `http://pipes.yahoo.com`. Accessed: 06-1-2014.

[39] National Instruments. Getting started with LabVIEW virtual instruments. `http://www.ni.com/white-paper/7001/en`, 2008. Accessed: 02-16-2013.

[40] National Instruments. The LabVIEW compiler - under the hood. `http://zone.ni.com/devzone/cda/pub/p/id/1177#toc1`, 2010. Accessed: 02-04-2013.

[41] National Instruments. Advanced LabVIEW debugging: Profiling VI execution with the LabVIEW desktop execution trace. `http://www.ni.com/white-paper/8083/en`, 2012. Accessed: 01-31-2013.

[42] National Instruments. NI LabVIEW VI Analyzer toolkit overview. `http://www.ni.com/white-paper/3588/en`, 2012. Accessed: 01-31-2013.

[43] National Instruments. NI LabVIEW full development system for Windows. `http://sine.ni.com/nips/cds/view/p/lang/en/nid/2454`, 2014. Accessed: 06-24-2014.

[44] Gary W Johnson. *LabVIEW graphical programming: practical applications in instrumentation and control*. McGraw-Hill School Education Group, 1997.

[45] S. C. Johnson. Lint, a C program checker. *Computing Science TR*, 65, 1977.

[46] Michael Jones and Christopher Scaffidi. Obstacles and opportunities with using visual and domain-specific languages in scientific programming. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 9–16, 2011.

[47] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *ACM SIGPLAN Notices*, volume 46, pages 155–170, 2011.

[48] H Katagiri, K Furukawa, and S Anami. Rf monitoring system in the injector linac. In *Proceedings of the International Conference on Accelerator and Large Experimental Physics Control Systems*, volume 99, pages 69–71, 1999.

[49] Elijah Kerry and Derrick Snyder. Comparing LabVIEW graphical code to text-based alternatives for use in test applications. In *2010 IEEE AUTOTESTCON*, pages 1–2, 2010.

[50] Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and Ali Ouni. Design defects detection and correction by example. In *Proceedings of the 19th International Conference on Program Comprehension*, pages 81–90, 2011.

[51] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 17–26, 2010.

[52] Toru Kisuki, P Knijnenburg, M OBoyle, and H Wijshoff. Iterative compilation in program optimization. In *Proceedings of Compilers for Parallel Computers*, pages 35–44, 2000.

[53] Andrew J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):1–44, 2011.

[54] Andrew Koenig. Patterns and antipatterns. *The patterns handbook: techniques, strategies, and applications*, pages 383–390, 1998.

[55] K Krippendorff. *Content Analysis: An Introduction to Its Methodology*. Sage Publications, Cambridge, MA, 2012.

[56] Chidamber Kulkarni, Francky Catthoor, and Hugo De Man. Code transformations for low power caching in embedded multimedia processors. In *Proceedings of the First Merged IPPS/ SPDP Symposium on Parallel and Distributed Processing*, pages 292–297, 1998.

[57] Susan Leah. *Increasing Customer Satisfaction Through Employee Satisfaction in a Call Center Environment*. PhD thesis, University of Wisconsin, 2005.

[58] Hailun Lin, Cheng Zhang, and Peng Zhang. An optimization strategy for mashups performance based on relational algebra. In *Web Technologies and Applications*, pages 366–375. Springer, 2012.

[59] Jie Liu, Jun Wei, Dan Ye, and Tao Huang. A new approach to performance optimization of mashups via data flow refactoring. In *Proceedings of the Second Asia-Pacific Symposium on Internetware*, pages 1–8, 2010.

[60] Usman Mansoor, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. Code-smells detection using good and bad software design examples. Technical report, Technical Report, 2013.

[61] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359, 2004.

[62] Daniel McFarlane. Comparison of four primary methods for coordinating the interruption of people in human-computer interaction. *Human-Computer Interaction*, 17(1):63–139, 2002.

[63] Erica Mealy, David Carrington, Paul Strooper, and Peta Wyeth. Improving usability of software refactoring tools. In *Proceedings of the 18th Australian Software Engineering Conference*, pages 307–318, 2007.

[64] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and A Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[65] Emerson Murphy-Hill and Andrew P Black. Breaking the barriers to successful refactoring. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering*, pages 421–430, 2008.

[66] Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, pages 5–14, 2010.

[67] Karthik Nagaraj, Charles Edwin Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 353–366, 2012.

[68] Bonnie A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT press, 1993.

[69] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 35th International Conference on Software Engineering*, pages 562–571, 2013.

[70] Vesa Norilo and Mikael Laurson. A method of generic programming for high performance dsp. In *Proceedings of DAFx-10*, pages 65–68, 2010.

[71] Xinghao Pan, Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Ganesha: blackbox diagnosis of mapreduce systems. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):8–13, 2010.

[72] Raymond R Panko. Applying code inspection to spreadsheet testing. *Journal of Management Information Systems*, pages 159–176, 1999.

[73] Trevor Parsons and John Murphy. Detecting performance antipatterns in component based enterprise systems. *Journal of Object Technology*, 7(3):55–90, 2008.

[74] Helmuth Partsch and Ralf Steinbrüggen. Program transformation systems. *ACM Computing Surveys (CSUR)*, 15(3):199–236, 1983.

[75] Aleksey Pesterev, Nickolai Zeldovich, and Robert T Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*, pages 335–348, 2010.

[76] Apan Qasem, Guohua Jin, and John Mellor-crummey. Improving performance with integrated program transformations. Technical report, In manuscript, 2003.

[77] Lin Qiu and Christopher K Riesbeck. Making critiquing practical: incremental development of educational critiquing systems. In *Proceedings of the 9th international conference on Intelligent user interfaces*, pages 304–306, 2004.

[78] Deon Rademeyer. *A benefits model for the call centre strategy.* PhD thesis, University of Johannesburg, 2014.

[79] Kamalasen Rajalingham, David Chadwick, Brian Knight, and Dilwyn Edwards. Quality control in spreadsheets: a software engineering-based approach to spreadsheet development. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, pages 10–19, 2000.

[80] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, volume 6, pages 115–128, 2006.

[81] TJ Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R Ruthruff, Laura Beckwith, and Amit Phalgune. Impact of interruption style on end-user debugging. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 287–294, 2004.

[82] Rebecca Sanders and Diane Kelly. Dealing with risk in scientific software development. *IEEE software*, 25(4):21–28, 2008.

[83] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.

[84] Stefan Slinger. Code smell detection in eclipse. Master's thesis, Delft University of Technology, 2005.

[85] Yannis Smaragdakis and Christoph Csallner. Combining static and dynamic reasoning for bug detection. In *Proceedings of the 1st international conference on Tests and proofs*, pages 1–16, 2007.

[86] Trevor J Smedley. Using pictorial and object oriented programming for computer algebra. In *Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing: technological challenges of the 1990's*, pages 1243–1247, 1992.

[87] Connie U Smith and Lloyd G Williams. Software performance engineering: A case study including performance comparison with design alternatives. *IEEE Transactions on software engineering*, 19(7):720–741, 1993.

[88] Connie U Smith and Lloyd G Williams. Software performance antipatterns. In *Workshop on Software and Performance*, pages 127–136, 2000.

[89] Connie U Smith and Lloyd G Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, pages 717–725, 2003.

[90] Kathryn T Stolee and Sebastian Elbaum. Refactoring pipe-like mashups for end-user programmers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 81–90, 2011.

[91] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Visual, log-based causal tracing for performance debugging of mapreduce systems. In *Proceedings of the 30th International Conference on Distributed Computing Systems*, pages 795–806, 2010.

[92] Catia Trubiani and Anne Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. In *ACM SIGSOFT Software Engineering Notes*, volume 36, pages 19–30, 2011.

[93] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of 9th Working Conference onReverse Engineering*, pages 97–106, 2002.

[94] Alexander Wert, Jens Happe, and Lucia Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments.

In *Proceedings of the 2013 International Conference on Software Engineering*, pages 552–561, 2013.

[95] Kirsten N. Whitley and Alan F Blackwell. Visual programming in the wild: A survey of LabVIEW programmers. *Journal of Visual Languages & Computing*, 12(4):435–472, 2001.

[96] Aaron Wilson, Margaret Burnett, Laura Beckwith, Orion Granatir, Ledah Casburn, Curtis Cook, Mike Durham, and Gregg Rothermel. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 305–312, 2003.

[97] Jeffrey Wong and Jason Hong. What do we mashup when we make mashups? In *Proceedings of the 4th international workshop on End-user software engineering*, pages 35–39, 2008.

[98] Guoqing Xu, Michael D Bond, Feng Qin, and Atanas Rountev. Leakchaser: helping programmers narrow down causes of memory leaks. *ACM SIGPLAN Notices*, 46(6):270–282, 2011.

[99] Jing Xu. Rule-based automatic software performance diagnosis and improvement. *Performance Evaluation*, 69(11):525–550, 2012.

[100] Jingbo Xu, Hailong Sun, Xu Wang, Xudong Liu, and Richong Zhang. Optimizing pipe-like mashup execution for improving resource utilization. In *Proceedings of the 5th IEEE International Conference on Service-Oriented Computing and Applications*, pages 1–4, 2012.

[101] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*, pages 93–102, 2011.

[102] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 199–208, 2012.

[103] Eyal Zimran and David Butchart. Performance engineering throughout the product life cycle. In *Proceedings of Computers in Design, Manufacturing, and Production*, pages 344–349, 1993.