Impact and utility of smell-driven performance tuning for end-user programmers

Citation	Chambers, C., & Scaffidi, C. (2015). Impact and utility of smell-driven performance tuning for end-user programmers. Journal of Visual Languages & Computing, 28, 176-194. doi:10.1016/j.jvlc.2015.01.002
DOI	10.1016/j.jvlc.2015.01.002
Publisher	Elsevier
Version	Accepted Manuscript
Terms of Use	http://cdss.library.oregonstate.edu/sa-termsofuse

Impact and utility of smell-driven performance tuning for end-user programmers

Christopher Chambers, Christopher Scaffidi*
{ chamberc, scaffidc} @onid.orst.edu

School of Electrical Engineering and Computer Science
Oregon State University
1148 Kelley Engineering Center
Corvallis, OR 97331

*=corresponding author, 541-737-5572

Abstract

This paper proposes a technique, called Smell-driven performance tuning (SDPT), which semi-automatically assists end-user programmers with fixing performance problems in visual dataflow programming languages. A within-subjects laboratory experiment showed SDPT increased end-user programmers' success rate and decreased the time they required. Another study, based on using SDPT to analyze a corpus of example end-user programs, demonstrated that applying all available SDPT transformations achieved an execution time improvement of 42% and a memory usage improvement of 20%, comparable to improvements that expert programmers historically had manually achieved on the same programs. These results indicate that SDPT is an effective method for helping end-user programmers to fix performance problems.

Keywords: end-user programming; performance; visual language

1 Introduction

Performance problems are among the trickiest types of bugs to debug. Studies have shown performance bugs take 32% longer to debug than non-performance bugs [23], resulting in substantial programmer frustration [24]. Worsening this problem, many programmers postpone fixing performance problems for as long as possible [21] even though the cost of fixing performance problems increases over time [1][25]. Consequently, programmers need a fast and easy way to find and fix performance problems early on.

Although most related research has focused on helping professional programmers with code performance, these problems also afflict end-user programmers, who are people who write code for their own use [12]. For example, scientists are concerned with the performance of code for data acquisition and analysis [10]. Other end-user programmers face similar issues, including engineers who for example encounter performance problems when creating prototype systems with the LabVIEW dataflow programming language [4]. For instance, a LabVIEW program (called a "VI") might need to read a signal every 100 milliseconds to control a scientific instrument or a robot. While the code might be correct, it could be written inefficiently and execute every 250 milliseconds, missing the desired read time, reducing the usefulness of the data. Researchers have noted the need for helping end-user programmers with performance problems in a variety of dataflow languages in addition to LabVIEW [9][11][13][20]. Much of the research on end-user programmers' code quality has focused on creating tools that aid in testing and debugging, particularly in spreadsheets (e.g., [2][3][5][17][18]), leaving end-user programmers little help with finding and fixing performance problems.

The chief prior work in this area is Smell-driven performance analysis (SDPA), which automatically provides situated explanations within a visual dataflow language IDE to help end-user programmers overcome performance problems without leaving the visual dataflow paradigm [4]. This technique analyzes end-user programmers' dataflow code to spot potential performance problems, based on inefficient structures called "smells" that appear in the code. A study showed that an SDPA-based tool for the LabVIEW programming language enabled end-user programmers to diagnose performance problems more quickly.

However, a limitation of that work so far is that SDPA only finds errors: it does not actually help people to fix those errors. Given a problematic code structure, end-user programmers are still left to determine a more efficient, replacement code structure.

We address this limitation through a new extension of SDPA called Smell-driven performance tuning (SDPT), which semi-automatically transforms end-user dataflow code to fix inefficient code structures. To test this approach, we implemented a tool for LabVIEW to investigate how well SDPT might work in practice. Specifically, for each smell, the tool offers the programmer one or more option for removing the smell. In some cases, the removal can be accomplished automatically at the programmer's command; in many other cases, due to the need for human input and judgment, the tool walks the programmer through the process of fixing the smell. A study using a test corpus of LabVIEW programs showed the tool-based code transformations improved performance almost as well as expert LabVIEW programmers do. An experiment showed the tool was useful for helping end-user programmers to more efficiently and successfully fix performance problems. Together, these studies show that SDPT is an effective technique for helping end-user programmers fix performance problems in dataflow code.

The remainder of this paper is organized as follows. Section 2 provides background on SDPA and summarizes other related work. Section 3 presents the SDPT technique in general, as well as the prototype that we implemented for helping LabVIEW programmers to apply this technique. Section 4 describes the study investigating the impact on performance of applying SDPT on test programs, and Section 5 presents the second study examining the usefulness of the technique for helping end-user programmers. Section 6 summarizes threats to validity, while Section 7 presents conclusions and opportunities for future work.

2 Smell-driven performance analysis

2.1 Background: Finding smells

When the idea of code smells [7] was proposed, the smells were designed to aid programmers in the refactoring of their code to ease understanding and maintenance. The most well-developed tools in this area,

which have already made it into industrial practice, incorporate heuristics for finding potential maintainability problems in object-oriented code (e.g., [16][19][22]) Such heuristics analyze the code to find code structures that usually (though not always) signal the presence of a problem. For example, 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 [22], via a dialog window within the IDE [19], or via annotations inserted alongside the code [15].

Smell-driven performance analysis (SDPA) adapts the concept of smells to the concern of performance in end-user programmers' dataflow code [4]. Given this shift in focus, it uses different heuristics than those used to find maintainability problems in object-oriented code. During the development of SDPA, we interviewed LabVIEW technical support personnel and reviewed LabVIEW training materials, enabling us to construct a catalog of 12 performance-related smells for LabVIEW, each of which consisted of a particular code structure (Table 1). For example, the No Wait in Loop smell was recognizable by the presence of a loop that ran without any wait primitive, which means that the loop would run unrestricted and starve any other threads for CPU time. Another smell, called Build Array in Loop, was recognizable by a loop containing the LabVIEW primitive that performed a memory reallocation and copy (to grow the array), which could cause repetitive, wasteful memory copying. SDPA also includes an optional profiling operation: the program in question is executed once, and a putatively problematic code structure is only flagged as a smell if a large proportion of the execution time occurs within that part of the code, thereby helping to reduce false positives. We tested SDPA's accuracy at finding real performance problems by downloading a corpus of 30 test programs with known performance problems, and we found that only 22% of the smells identified were false positives. Moreover, we conducted a laboratory study that showed end-user programmers were 2.6 times as likely to succeed at finding a performance problem in example code when they had SDPA at their disposal.

The foregoing work highlights the value that smells have in not only finding maintainability problems, but also in finding performance problems in end-user programmers' dataflow code. SDPA, in particular, provides a foundation for our current effort. Our new work enables end-user not only to find performance but also to fix such problems.

2.2 Related work: Fixing smells

Given the longstanding work on maintainability-related smells, there are many tools that help professional programmers refactor their code to fix smells [7]. "Refactoring" refers to code transformation that leaves the semantics of the code unchanged (specifically, the inputs and/or outputs of the program, and their mathematical relationship to one another), but which improves its quality. Different tools offer different interaction styles to the programmer. For example, in [16] the tool acts as a guide that walks the programmer through a mostly-manual refactoring process. This interaction style is designed to help teach the programmer about refactoring, with the dual intent of education about object-oriented maintainability and avoiding similar problems in the future. Two other tools [8][14] apply automatic refactorings on code or suggest a menu of automated refactorings rather than walking programmers through a manual process of improving maintainability, while others (e.g., [9]) are focused on aiding similar tasks with functional programming code. Such existing tools apply to various textual languages in non-dataflow paradigm; in contrast, our current work focuses on visual dataflow code.

Although performance in dataflow code has received less attention than maintainability of object-oriented code, two tools are available to improve the performance of dataflow code through refactoring. The first applies refactorings to Yahoo! Pipes mashups to minimize the redundant operations that are in a program [9]. 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. This method showed a slight decrease in execution time (1-5%). The second tool [13] enhances the performance of a Yahoo! Pipes mashup by automatically refactoring the structure of the data flow. It does this by generating a large set of semantically equivalent programs, applies heuristics to estimate a projected execution time for each, and recommends that the programmer use the new version that has the lowest estimated execution time. A study showed that execution time decreased 22% on average over a test corpus. In contrast, our approach achieves a higher level of performance improvement by considering a broader range of transformations (including those that are non-semantics-preserving).

In our view, the key limitation of these existing approaches is not just the small improvement effected by the first tool, nor the second tool's reliance on brute force generate-and-search, but rather the fact that these tools only apply refactoring transformations, which would not fix many of the performance problems we have found in our preliminary work. For example, fixing the No Wait in Loop smell mentioned above requires inserting a pause into the loop so that other threads can get more CPU time. For instance, if a program's data-acquisition loop A is starving a data-acquisition loop B of CPU time because A is running unbounded, then the fix is to decrease the sampling rate of A so that the sampling rate of B can increase. This modifies what input the program acquires, which may indirectly affect its outputs, as well. The semantics are altered--a transformation that refactoring cannot achieve, by definition. Because of examples like these, transformations other than refactorings may be required in general to address the broad range of performance problems that end-user programmers encounter in practice. Our goal is therefore not only to fix performance problems by refactoring, but also to support appropriate non-semantics-preserving transformations.

3 Smell-driven performance tuning

While the programmer is working with a dataflow program in an IDE, SDPA identifies a set of locations in the program containing structures that could potentially impact performance [4]. If the programmer runs the program, SDPA can track whether those portions of the code consume a large proportion of the CPU time or memory, to help filter out false positives. The memory is tracked because excessive memory use has been observed to cause thrashing to disk, depending on how much memory happens to be available when the program is run, which is hard to predict in advance [4]. SDPA inserts icons into the dataflow program to indicate the portions of code that should probably be fixed, and it provides a table of explanations about these problems.

Smell-driven performance tuning (SDPT) extends SDPA by augmenting the table of explanations with a menu of options (displayed as buttons) for restructuring each identified portion of the code. For example, for each loop that demonstrates a Build Array in Loop smell, SDPT offers an option to fix that instance of the smell. In addition, a separate pane is provided where the programmer can view information about each transformation. This pane can either be docked somewhere in the IDE or can be left to float above the code. This pane additionally provides links to white papers or web pages if applicable.

SDPT allows for a range of different user interaction styles, depending on the level of automation that can reliably be performed for a given smell. Some smells can be fixed by a fully automated process (which we call a Full Transformation), if the programmer chooses to do so. Other smells can be fixed reliably only if the programmer provides additional information required to perform the transformation (which we call a Partial Transformation). Still other smells cannot be fixed automatically, so the tool instead guides the programmer through a manual process (which we call a Wizard Transformation).

Figure 1 summarizes the process above, as a flowchart.

3.1 Methodology for determining smell transformations

Before we could implement a prototype, we had to determine appropriate transformations for each smell that we had uncovered in prior work. We accomplished this by interviewing AE Specialists, who are the technical support personnel at National Instruments that help LabVIEW programmers to fix performance problems and are all experienced LabVIEW programmers.

We recruited participants by sending emails that described 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.

We showed participants example programs for smells, which we had retrieved from the National Instrument LabVIEW forums. Beforehand, we briefly tested these programs to check that they showed the reported performance problems. For each smell, we asked the AE Specialists to describe how they would fix the given program to solve the performance problem. To verify that this proposed fix was a reasoned solution (and not a guess), we then asked in detail about the solution and why it would fix the problem.

Based on these interviews, we implemented a set of transformations. We describe these transformations in the subsections below, then the evaluation of their impact and utility in Sections 4 and 5.

3.2 Full Transformations

The first category of transformations contains those that fix smells automatically with the click of a button. This is the simplest category of transformation to apply because the programmer does not have to do much work to solve the detected performance problems. This does not mean that the transformation is a refactoring--in fact, the transformations typically alter program's inputs, outputs, or user interface. Rather, our conversations with AE Specialists convinced us that the transformation would be reliable enough not to harm program behavior in practice, as long as the programmer was informed of what the transformation would do to the program. Therefore, our prototype also provides information about what the transformation does, so that the programmer can evaluate whether to apply it. In addition, it explains why this transformation will solve the problem in the hopes that the programmer will learn the cause of the problem and can avoid it in the future.

3.2.1 No Wait in Loop

Having no timing structure inside of a loop can usually be solved by adding a wait primitive. 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, so the AE Specialists counseled to focus on the situation where adding a wait operation is sufficient.

There are two primary types of waits in LabVIEW. Most of the AE Specialists preferred the "Wait Until Next ms Multiple" node, since it is generally considered to have more precise timing, which is particularly important for real-time applications. The final step of determining the transformation was to decide on the best wait time. Several participants noted that they will often look at CPU usage and loop rate to guess an effective wait time. Others stated that they usually pick a generic time, such as 10 ms and then increase 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 1 ms would cause the loop to pause, solving many performance problems by allowing other loops to run (since LabVIEW, internally, does not implement cooperative multi-threading). Our conclusion from these discussions was that starting with a small wait time would likely solve many problems, and the programmer could increase this time if desired.

Figure 2 summarizes the transformation that we implemented. Figures 3 and 4 show a program before and after applying this transformation, with the affected code highlighted in red within Figure 4. In addition to this highlighting, the tool provides a separate pane of information (gathered from the AE Specialists) about why a Wait node should be inside of a loop and when someone might want to use other available timing structures. 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.

3.2.2 Sequence Structure

Sequence Structures force two portions of LabVIEW code to run one after the other, but using such structures effectively can be tricky, particularly if data is shared between frames in the sequence structure. If that is the case, then a Shift Register (essentially a variable) needs to be created. This can result in many Shift Registers being created, which may make the code more confusing and, according to interviewees in our preliminary work [4], can lead to performance problems due to inefficient passing of data among sequence frames.

Figure 5, already somewhat confusing, is only a simple example of a structure; several AE Specialists noted that with more complex uses, inefficient code is quite common. For example, one participant had seen a program that was using more memory than expected and was slowing the execution speed to a crawl (due to disk swapping). After investigation, the AE Specialist discovered that the programmer was passing an array between frames incorrectly, which caused multiple copies of the array to be created every time the sequence structure was executed.

Due to such problems, a few participants thought that replacing a Stacked Sequence Structure with an equivalent state machine might alleviate some of these issues, as all of the data that is used in a state machine is visible at once. Other AE Specialists did not have a suggestion about the best transformation to replace a Sequence Structure. Most of them agreed that regardless of the transformation, replacing a Sequence Structure with anything equivalent was unlikely to change the performance much for the worse.

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.

One item of importance that was discussed is that programmers will often use a Sequence Structure to time how long a snippet of code takes to execute. This generally takes the form of a three frame sequence structure in which the first frame will set up the timing (usually by recording a timestamp), the second will contain the code the programmer wants to time, and the final frame will end the timing (usually by recording a second timestamp) and display how long the snippet took to execute. Many of the participants also mentioned that sequence structures that are less than three frames rarely make sense to transform since they are so small anyway. For these reasons, the transformation that we implemented only affects sequence structures containing more than three frames. The complete transformation is shown in Figure 6, while Figures 7 and 8 illustrate its effect on an example program.

3.2.3 Infinite While Loop

In our prior study, AE Specialists associated some perceived performance problems with having an infinite While Loop, because often the programmer 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 that data will not be saved when the program is aborted as the abort option simply ends the program wherever it is in the execution. This can be problematic, particularly for programs that are collecting data to be stored in a file after the loop is completed.

All interviewees in our current study came up with the same fix for this problem, which was to add a button the programmer 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. Figure 9 defines this transformation, Figure 10 shows an example of a program with this smell, and Figure 11 shows the program after correction. In this instance, the button has been created and wired to the loop ending parameter (highlighted in red). This fix alters the user interface of the program, whose visual layout is displayed in a separate window (not shown in Figure 11).

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 dependent on algorithmic choices by the programmer. For example, a programmer 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, so our prototype's informational pane explains the issues to the programmer in case if further code alterations are desired.

3.2.4 Uninitialized Shift Registers

A Shift Register provides users a way to store and then reuse data in the context of specific structures. In a Sequence Structure, it allows the programmer to pass data between frames, while in a loop it makes the data available during each iteration. An uninitialized Shift Register can result in performance problems because it causes a program to store data over multiple runs, which unnecessarily wastes memory and can become significant in the case of large arrays. The example program in Figure 12 has four Shift Registers, only one of which is initialized.

AE Specialists explained that the correct way to fix this is to add a 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. Figure 13 defines the transformation.

This transformation will retain the behavior of a given program on its first execution but alter the behavior of subsequent executions. Therefore, as with other transformations, programmer judgment is required about whether to apply this transformation. When programmers intend for data to be retained over multiple runs, they are unlikely to need this transformation.

3.2.5 Redundant Operations on Large Data

The primary examples of this smell deals with unnecessary math being performed. 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 add 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.

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 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. Figure 14 shows this transformation.

3.3 Partial Transformations

Partial Transformations can be applied semi-automatically, but they require programmer input to complete them reliably. In other words, whereas Full Transformations merely require judgment from the programmer about whether to apply a transformation, Partial Transformations also require information from the programmer. Put another way, Full Transformations are reliable enough that if they are performed, the resulting program will not crash (though it perhaps might not speed up much in some cases). In contrast, Partial Transformations would sometimes cause some programs to crash in the absence of information from a programmer.

3.3.1 Build Array in Loop

The Build Array in Loop smell is a common 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.

One 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 built every iteration of the loop. Specifically, each iteration of the restructured loop would output a single array element, which LabVIEW then assigns to the i^{th} element of the array, where i is the current loop iteration.

The second solution was to wrap the Build Array node with an In-place Structure. In LabVIEW, this structure forces everything inside of it to be done memory-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 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 pre-allocates an array large enough to contain all the data, and inside the loop the value at element i is replaced with the value that is generated inside the loop.

Given the disagreement among interviewees, we implemented the three and ran them on the test programs from the interview to see which transformation would be most promising. The first purported solution had no effect on performance at all (presumably because, internally, LabVIEW still is building a growable array). The second had a small effect, while the third had the most, so we chose to retain the third as our transformation.

However, the transformation is different depending on the type of loop that is part of the smell, thus necessitating programmer input. If the Build Array node is inside of a For Loop that only initializes one array item per iteration, 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 programmer will be informed that the transformation thus applied is not reliable until after the programmer changes the size of the array to reflect how many times the While Loop will iterate. If the programmer fails to provide the necessary information about the array size (by changing the array size from the default large constant inserted by the tool), then the program will not behave as expected and may crash with an out-of-memory exception.

Figure 15 defines the transformation. Figure 16 shows a LabVIEW program containing three Build Array nodes in a For Loop and Figure 17 shows the program after the transformation 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, one 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.

3.3.2 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. The arrays are then concatenated in one step after the loop. The transformation for this smell is shown in Figure 18.

Figure 19 shows an example program with this smell, and Figure 20 shows the result after transformation. 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 after the loop terminates, highlighted on the right side of the figure. Note that the program contains two String Concatenation nodes. 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 these cases, the initialization cannot be reliably automated because the number of loop iterations cannot be determined algorithmically. If the programmer does not input the needed array size after the transformation, then the program will crash. So the programmer is informed that the size of the array should be changed to reflect how many times the While Loop will iterate.

3.4 Wizard Transformations

The final set of transformations involves those that cannot be automated much at all. This is either because they require manually setting the properties of certain elements or because they require extensive effort from the programmer at multiple steps during the transformation. In these instances, the tool aims to guide the programmer through the process with suggestions, including a step-by-step guide to fix the problem.

3.4.1 Multiple Array Copies

Copying large arrays can consume substantial CPU time, so AE Specialists were unified about trying to avoid array copies whenever possible, but they were split on the best way to eliminate unnecessary copying. In general, fixing this smell usually requires changing the algorithm or entirely refactoring the program. This is not something that can be reduced to an automated transformation, especially since it would require asking programmers 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 programmers down a path that is not optimal. Namely, they worried that programmers 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 (where two dataflow operators that modify an array both consume the same input array) 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 programmer 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 programmer wires an array into a given structure two (or more) different times. For instance, a While Loop might consume a certain array; if the programmer defines two inputs to the loop and assigns the array to both inputs, then two copies of the array are created. By contrast, if the array is only wired once to the loop, and then branched internally within the loop, multiple copies will not be created. When our tool detects this case, it suggests changing 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 programmer can be notified, removing this from the program is problematic, since the two different operations might both be needed for other parts of the programmer's algorithm. Our 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. This might improve performance due to reduced memory consumption and therefore reduced thrashing, but it also might marginally harm performance due to the fact that the resulting code may need to be executed by the LabVIEW runtime in sequence rather than in parallel. The tool informs programmers that this works in some cases, but in many others it is not beneficial.

A final suggestion that the tool provides, when appropriate, is to move the array copies outside of loops if the programmer cannot remove them entirely. By removing them from a loop, it will reduce the number of copies that are created. This can often be enough to improve the performance. The process for providing assistance for this smell is shown in Figure 21.

3.4.2 Non-Reentrant SubVI

Each LabVIEW program is called a VI ("Virtual Instrument"), so a subVI is a subprogram that serves as a callable function within a larger VI. Marking a subVI as reentrant or non-reentrant affects whether it is allowed to be run in single- or multi-threaded mode, but it is a program setting and cannot be done programmatically within the IDE. Therefore, the tool cannot perform it automatically, due to the limitations of the IDE's internal APIs. Instead, it instructs the programmer how to get to the proper menu and then how to set a subVI as reentrant. It also recommends that the programmer change the priority from normal to subVI priority to help with the timing of the program. The process for providing assistance for this smell is shown in Figure 22.

3.4.3 No Queue Constraint

LabVIEW operations can pass data asynchronously via queues, but unbounded queues can grow without bound and cause thrashing. AE Specialists mainly considered it a likely problem when running on 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 programmer 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 programmer 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 programmer know what is going on with the program and allow making a change.

When this smell is detected (and the programmer decides to address it), the tool will gather input from programmers 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 programmer through steps on how to add this to their Queue. The process for providing assistance for this smell is shown in Figure 23.

3.4.4 Terminals Inside a Structure

Programs can read and write data of user interface elements, which appear as terminal primitives in the dataflow program. However, due to the fact that user interface elements are not thread-safe, LabVIEW internally has only one distinguished thread that is allowed to access user interface elements. Other threads that need to interact with user input elements must internally pass a message to the distinguished thread,

block, wait for the distinguished thread to complete the read or write, and then continue. This block-and-wait behavior greatly decreases the speed of the waiting thread, especially if the distinguished thread happens to be very busy during a particular period of time. If the waiting thread is running over a While Loop or a similar structure, the program can slow to a crawl.

The transformation for fixing this smell is much more complex than just moving all terminals outside of the structure, since that means the terminal is only read (or written) a single time outside the structure. For example, there could be a Boolean terminal inside of a loop, meaning that the Boolean is read on each iteration. Moving it outside the loop would mean the input is read only once, prior to the loop. User input altering the Boolean during the loop would be completely ignored.

Figure 24 shows an example of this smell. In this instance, there is a Boolean terminal, "Add to Array", that is controlling a Case Statement inside the While Loop. The programmer can interact with this terminal from the front panel while the program is running. When the programmer changes the value of this terminal to true, data sampling will begin and values will be added to the array. If the terminal value is changed to false, then sampling will stop and new values will stop being added to the array. This example also has a 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 programmer can change this value during execution and have that value be added to the array. 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 advises the programmer on why having terminals inside of a structure are bad, and it suggests moving them outside the structure, but ultimately if they are required to be inside for algorithmic reasons, then they should not be moved.

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 time the loop iterates which greatly slows down the execution of the program. For this reason, the tool will ask the programmer 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 programmer to determine if they can afford to move terminals or if the terminals are required to be inside the structures.

Figure 25 demonstrates the process that the wizard takes to help programmers, while Figure 26 shows a LabVIEW program that contains three terminals inside of the loop (highlighted in red). Figure 27 demonstrates the result after the transformation has been applied, which involves moving the terminals outside of the loop.

3.4.5 Too Many Variables

AE Specialists in our earlier study [4] noted programs with many variables are often excessively complicated and therefore poorly performing. Variables are particularly problematic when they are written to and/or read in parallel, which can lead to race conditions and cause scheduling overhead, which slows the performance.

Fixing such problems usually requires substantially rewriting the algorithm of the user's program, which is a process that cannot be automated. Therefore, the tool gives guidance (Figure 28). Specifically, it explains three ways that the user might be able to remove or restructure variables. First, it explains that it might be possible to replace a variable with a Shift Register in situations when writes to a variable are all in the same loop. Second, the tool explains that it might be possible to rewrite variables as a special thread-safe type of variable supported by LabVIEW (called "Functional Global Variables") that is useful for safe and efficient scheduling when a variable is read and written in many places throughout the code. Third, the tool explains the general concept of refactoring, with an emphasis on how to avoid variables in dataflow code.

4 Evaluation of transformations' impact on performance

To assess how well the approach would accurately identify performance problems in general, a corpus of real world LabVIEW programs was gathered that the tool could be run on. This is the same corpus that we previously used to evaluate the effectiveness of SDPA for finding performance problems [4]. The corpus consisted of 30 pairs of programs, where the first in each pair was an uploaded program from a programmer that contained a performance problem before fixing, and the second was a version after fixing by an expert so the performance problem had been removed. This corpus thus enabled us to assess the impact of each transformation individually and to compare the overall impact of SDPT to that of experts.

4.1 Performance impact of individual SDPT transformations

We expected that SDPT would positively impact performance, in terms of other execution time and/or memory usage, for most smells and for smells overall. We tested this hypothesis by running SDPA to identify smell instances, then completing every SDPT transformation on every smell instance, performing manual operations as required for Partial and Wizard transformations when prompted by the tool.

Averaged over all smells, we found that each transformation yielded an average execution time improvement of 26%, in addition to 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 previous research [4] LabVIEW programs contained an average of nearly 5 smells per program, so programmers could anticipate even greater benefits when applying multiple smells, which Section 4.3 explores in detail.

Table 2 summarizes the per-smell improvements in performance. It is noteworthy that 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, it is clear that the transformations help the most in the situations that programmers will likely encounter most often.

The results confirm our expectation that SDPT transformations improve performance for most smells, since for 7 out of the 11 smells that actually occurred in the corpus, applying the transformations improved execution time by at least 10%. For 3 of these 7 smells, memory usage also improved by at least 10%. There were three smells (Too Many Variables, Infinite While Loop, and No Queue Constraint) where the transformation gave little benefit and one that did not occur (Redundant Operations), so these transformations could conceivably be removed to simplify our tool, with no resulting reduction in its impact on performance.

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:

No Wait in Loop: This transformation greatly improved execution time (37%), while slightly increasing memory usage (-13%). The primary culprit to the memory gain seems to be the fact that many affected programs used little memory to begin with, and small fluctuations show up as much larger percentages.

Terminals in Structure: The transformation itself showed great improvement in terms of average impact on time (23%), with negligible impact on memory usage (-1%).

Multiple Array Copies: This transformation improved execution time (25%) but harmed memory usage (-16%). When there were multiple instances of this smell in the same structure, applying the fix to one smell instance allowed execution speed to increase and the program to thereby invoke the other smell instances more often, which ended up consuming more memory overall. Thus, this negative result is an artifact of the fact that we only applied a single transformation at a time in this analysis, and it will not be an issue in practice if the programmer fixes all instances of the smell.

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 slight increase in memory used.

Overall, the impacts on execution time outweighed the impacts on memory usage, and increases in memory usage generally harm performance only in a memory-constrained execution environment, so SDPT's impact on performance is likely to be quite positive overall.

4.2 Comparison of SDPT impact to experts' impact on performance

To compare the tool transformations with those of experts, we applied all applicable transformations on the 30 programs and measured the resulting performance impact. As above, we manually performed all operations required by the Partial and Wizard transformations. We then also considered the expert-based fixes for each program in the corpus and measured the resulting performance impact. As in the analyses above, performance impacts were expressed as percentages relative to baseline.

We hypothesized that SDPT's impact on performance would be comparable to that of experts. Therefore, we used the two-tail Mann-Whitney U test to test this hypothesis. We performed the test twice: once with impact on execution time and once with impact on memory usage.

Table 3 summarizes the average impacts on performance. Averaged over all programs, applying all available SDPT transformations to a program achieved 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 at $p<0.05$. (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. These results also compare very favorably to the performance of the dataflow-refactoring tools discussed in related work (Section 2.2), which achieved improvements of 1-5% and 22%, respectively.

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%. Thus, end-user programmers relying on SDPT will often obtain a benefit comparable to what they would obtain from contacting an expert and waiting for advice.

There were 4 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 SDPT even to experts.

There were 7 programs in which the expert solution was greatly superior (better than 20% difference) than the tool-generated solutions. To achieve these improvements, the experts had to substantially modify the code, in 5 cases completely rewriting the program. Therefore, in the majority of cases, SDPT is an adequate replacement for getting expert help with performance problems, but there still is a non-trivial role for experts in aiding less experienced programmers with the most hard-to-solve problems.

5 Evaluation of SDPT in the hands of end-user programmers

We performed a laboratory study to evaluate the utility of SDPT, as well as to uncover opportunities for future prototype enhancements. During this within-subjects study, 32 people performed tasks with our tool, and the same 32 people performed comparable tasks without the tool.

5.1 Methodology

Participants: We recruited participants by sending emails to Application Engineers at National Instruments. (The study was conducted in National Instruments headquarters.) 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 our prototype's help. For this study, 32 participants were recruited.

Tasks: We gave every participant two program-troubleshooting tasks, and each task involved finding and fixing performance problems in two different LabVIEW programs (total of 4 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 SDPT. For the other, the participant only used the LabVIEW IDE without SDPT.

The four LabVIEW programs that we used for this study were real-world LabVIEW programs that contained performance problems that had been found on the LabVIEW forum. 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 (i.e., based on our own experience with fixing the problems). 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, 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 2.) 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 programmer 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) to cancel learning effects.

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 participants 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 programmers had proposed on the forum for fixing the program. 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 to gather their opinions on the tool, its usefulness, and any problems they experienced with it.

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 following null hypotheses on success, time, and efficiency:

H1: Participants will be at least as successful (score as many points) when debugging performance problems without SDPT as with it.

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

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

5.2 Quantitative Results

The prototype tool demonstrated a clear advantage over the normal LabVIEW IDE (Table 4). 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 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 SDPT.

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 programmers 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, a statistically significant difference (Mann Whitney $U=31$, $p < 0.001$). Overall, the tool helped programmers become over three times (3.7) more efficient at debugging performance problems.

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.

5.2.1 Qualitative Results

A total of 31 of the 32 participants opted to complete our questionnaire, which asked participants to state what parts of SDPT were helpful or not helpful.

A total of 30 out of 31 respondents mentioned at least one way in which the tool was useful. More specifically, nine of these participants 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 mentioned that the tool made LabVIEW easier to use. By far the most commonly stated aspect (13 participants) to cause LabVIEW to be easier to use was the ability to pinpoint causes of problems. Six participants felt that the ability to receive feedback about a program (pre- or post-transformation) 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 programmers as they learned how to avoid pitfalls. Finally, two felt that the ability to remind programmers 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."

A recurring theme, regarding opportunity for improvement, was that 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 programmer and felt like it was too much at times or that it took up too much space on the screen. There were several participants who cautioned about the use of the tool. One participant explicitly asked to avoid a Clippy-like interface, stating: "Please do not make a paper clip that tells me my VI is inefficient." In total, five participants 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 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 was 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 programmer. 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.

5.3 Threats to validity

One key threat to validity is the potential that the participants in our study might not be entirely representative of other end-user programmers who encounter performance problems in their code. We did recruit from a sample frame of very skilled programmers (AEs) who were qualified to solve these problems without the assistance of our technique. Most programmers probably have less skill than these participants but, nonetheless, even our participants benefited from our tool. Thus, it is reasonable to expect that other end-user programmers with lower skill levels would likely benefit even more from our approach than our test subjects did.

Another concern is the extent to which we can generalize from our results to other situations where end-user programmers need to fix performance problems. The user study alone would not address this threat to external validity, but the additional study involving the corpus of programs from the online forms (Section 4) provides evidence that the results would generalize over an ecologically valid range of LabVIEW programs.

However, we cannot say that the specific smells or transformations involved in our LabVIEW prototype are present in other end-user dataflow languages, such as Yahoo! Pipes. Nonetheless, such languages and their runtime environments do offer the key features required for SDPT, particularly a graph structure amenable to analysis and transformation. Moreover, the related work cited in Section 2 does provide evidence that some performance improvement can be achieved in such a language using refactoring, which is a special case of transformations, so it is reasonable to expect that equal or greater performance improvement could be achieved through structural transformations in general.

6 Conclusions and future work

In this paper, we have presented a technique to help guide end-user programmers through the process of debugging dataflow code for performance problems. Our technique has proven useful for improving the success and speed of end-user programmers, and it has achieved performance improvements almost as great as those achieved by expert programmers in a given language.

During the user 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. We would hope that providing this information to programmers will help them learn about the potential pitfalls while programming and help avoid these problems. In future work, this hypothesis could be evaluated with a field study that tracks programmers 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.

Another potentially beneficial area would be allowing programmers to create their own smells and transformations. This would allow experts to create smell/transformation combinations that could be installed by novice programmers. This would allow the tool to be highly flexible in the types of problems that could be detected and fixed.

While the main goal of this research was to find and fix performance problems in an individual's code, there is no reason why it could not be extended to find performance problems in code provided by other

LabVIEW programmers. Smell-drive performance analysis and Smell-driven performance tuning 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. The problem areas could be marked and transformations could be applied if desired. This would create a repository of programs that are likely to perform well. This might help to encourage reuse between programmers and help create more modular programs.

6.1.1.1.1 ACKNOWLEDGMENT

We thank National Instruments for funding this research, helping to recruit study participants, and providing access to the latest version of the LabVIEW development environment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of National Instruments.

6.1.1.1.2 REFERENCES

[1] Boehm, B. (2001) Software engineering economics. *Pioneers and Their Contributions to Software Engineering*. 99–150.

[2] Burnett, M., Sheretov, A., Ren, B., and Rothermel, G. (2002) Testing homogeneous spreadsheet grids with the "what you see is what you test" methodology. *IEEE Transactions on Software Engineering. 28(6),576–594.*

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

[4] Chambers, C. and Scaffidi, C. (2015) Utility and accuracy of smell-driven performance analysis for end-user programmers, *Journal of Visual Languages & Computing, 26*, 1-14, doi:10.1016/j.jvlc.2014.10.017.

[5] Erwig, M., Abraham, R., Kollmansberger, S., Cooperstein, I.. (2006) Gencel: a program generator for correct spreadsheets. *Journal of Functional Programming*. 16(3),293–325.

[6] Farmer, A., Gill, A., Komp, E., and Sculthorpe, N. (2012) The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. *ACM SIGPLAN Notices*, 1-12.

[7] Fowler, M., and Beck, K. (1999) *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, Boston, MA, USA.

[8] Grant, S. and Cordy, J. (2003) An interactive interface for refactoring using source transformation. *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects*. 30–33.

[9] Hassan, O., Ramaswamy, L., and Miller, J. (2010) Enhancing scalability and performance of mashups through merging and operator reordering. *2010 IEEE International Conference on Web Services* (ICWS), 171-178.

[10] Jones, M., and Scaffidi, C. (2011) Obstacles and opportunities with using visual and domain-specific languages in scientific programming. *IEEE Symp. on Visual Languages and Human-Centric Computin.*, 9-16.

[11] Katagiri, H., Furukawa, K., and Anami, S. (1999) RF monitoring system in the Injector Linac. *Intl. Conf. on Accelerator and Large Experimental Physics Control Sys.*, 69-71.

[12] Ko, A., Abraham, R., Beckwith, L., Blackwell, A., Burnett, B., Erwig, M., Scaffidi, C., Lawrence, J., Lieberman, H., Myers, B., Rosson, M., Rothermel, G., Shaw, M., and Wiedenbeck, S. (2011). The State of the Art in End-User Software Engineering. *ACM Computing Surveys, 43*, 3, 1-44.

[13] Liu, J., Wei, J., Ye, D., and Huang, T. (2010) A new approach to performance optimization of mashups via data flow refactoring. *Proceedings of the Second Asia-Pacific Symposium on Internetware*, 6:1-6:8.

[14] Mealy, E., Carrington, D., Strooper, P., and Wyeth, P. (2007) Improving usability of software refactoring tools. *Proceedings of the 18th Australian Software Engineering Conference*. 307–318.

[15] Murphy-Hill, E. and Black, A. (2008) Breaking the barriers to successful refactoring. *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering*. 421–430.

[16] Murphy-Hill, E. and Black, A. (2010) An interactive ambient visualization for code smells. *Proceedings of the 5th international symposium on Software visualization*. 5–14.

[17] Panko, R. (1999) Applying code inspection to spreadsheet testing. *Journal of Management Information Systems*. 159–176.

[18] Rajalingham, K., Chadwick, D., Knight, B., and Edwards, D. (2000) Quality control in spreadsheets: a software engineering-based approach to spreadsheet development. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*. 10–19.

[19] Slinger, S (2005). Code smell detection in eclipse. Master's thesis, Delft University of Technology.

[20] Smedley, T. (1992) Using pictorial and object-oriented programming for computer algebra. *ACM Symp. on Applied Computing*, 1243-1247.

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

[22] Van Emden, E. and Moonen, L. (2002) Java quality assurance by detecting code smells. *Proceedings of 9th Working Conference onReverse Engineering*. 97–106.

[23] Zaman, S., Adams, B., and Hassan, A. (2011) Security versus performance bugs: a case study on firefox. *Proceedings of the 8th working conference on mining software repositories*, 93–102.

[24] Zaman, S., Adams, B., and Hassan, A. (2012) A qualitative study on performance bugs. *Proceedings of the 8th working conference on mining software repositories*, 199–208.

[25] Zimran, E. and Butchart, D. (1993) Performance engineering throughout the product life cycle. *Proceedings of Computers in Design, Manufacturing, and Production*. 344–349.