Table 1: Performance-related smells that our prior interviews of AE Specialists and reviews of training materials previously revealed as being common with LabVIEW users [4]

Problem	Description
No Wait in Loop	If there is no wait inside of a loop, it could cause synchronization issues and also usually makes front panel elements unresponsive
Terminals in Structure	When a terminal is inside of a structure in a LabVIEW program it causes the UI thread to be hit frequently.
Build Array in Loop	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.
Uninitialized Shift Registers	An uninitialized Shift Register causes a VI to store data over multiple runs, which unnecessarily wastes memory.
Multiple Array Copies	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
Sequence Structure	Sequence structures remove a lot of the power of LabVIEW, limiting the compiler's optimization options and potentially reducing readability.
Too Many Variables	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 use.
Infinite While Loop	A loop that runs infinitely (sometimes due to the lack of any rule for termination) may cause issues with expected behavior and execution time.
No Queue Constraint	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.
Non-reentrant subVI	Non-reentrant VIs have a data space shared between multiple calls. If a non-reentrant VI is in a structure that can be parallelized, blocking causes poor execution speed.
String Concatenation in Loop	When string concatenation occurs inside of a loop, it builds a new copy every iteration. This can cause slow performance and memory issues.
Redundant Operations	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)

Table 2: Average impacts on performance of applying each tool-based transformation (per smell instance)

Smell	Execution Time Improvement	Memory Usage Improvement	Total Number of Instances
No Wait in Loop	37%	-13%	37
Terminals in Structure	23%	-1%	34
Build Array in Loop	34%	37%	34
Uninitialized Shift Registers	22%	86%	14
Multiple Array Copies	25%	-16%	11
Sequence Structure	-2%	-1%	6
Non-reentrant subVI	13%	-2%	3
Too Many Variables	-1%	-1%	4
Infinite While Loop	0%	1%	3
No Queue Constraint	0%	0%	2
String Concatenation in Loop	16%	20%	2
Redundant Operations on Large Data	N/A	N/A	0
Average over all smell instances	26%	12%	150

Table 3: Average impacts on performance (per program)

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

Table 4: Summary statistics, averaged per task over all cases

	With tool	Without tool
Progress made (points)	97.66	67.19
Time spent (minutes)	5.66	13.28
Average Efficiency (Points/Minute)	26.42	7.14

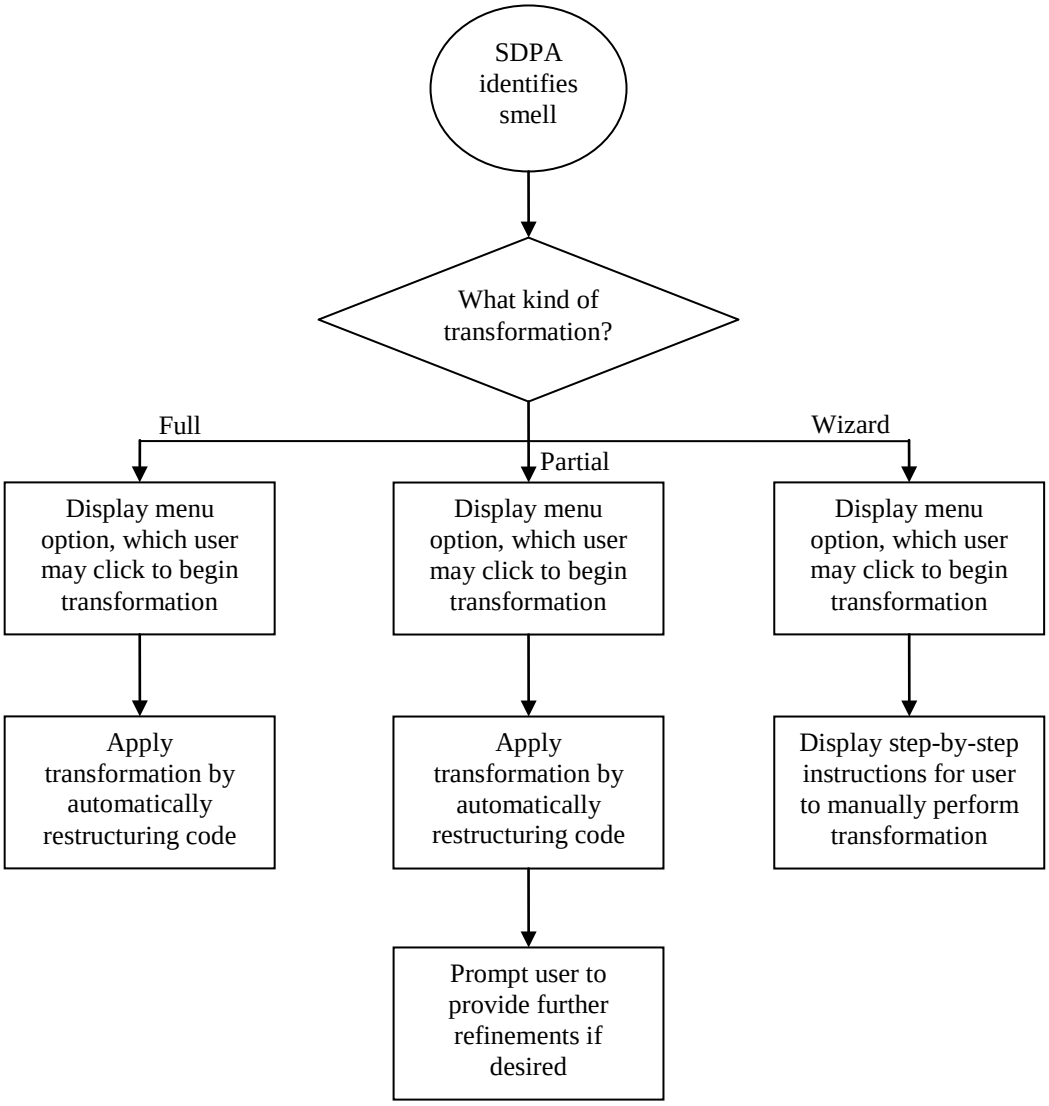

Figure 1: Process flowchart for SDPT

For a loop with no Wait node:

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

Figure 2: No Wait Node Transformation

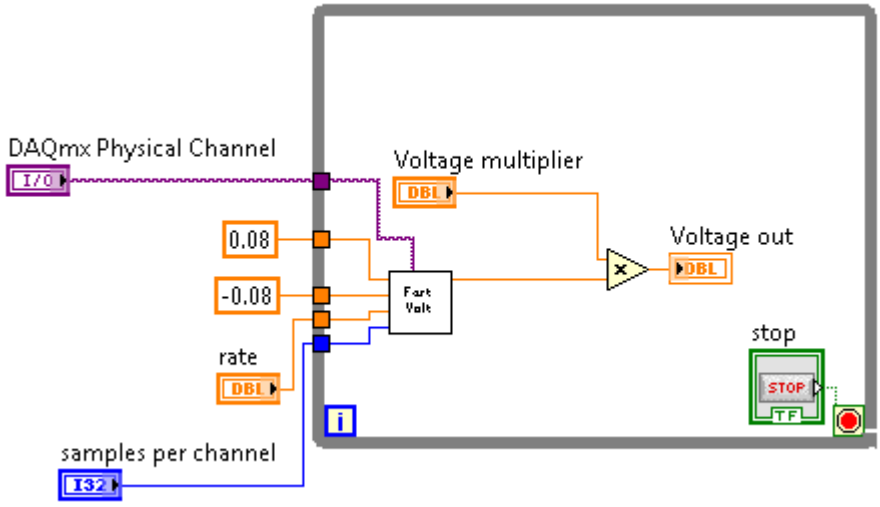

Figure 3: LabVIEW Program demonstrating a No Wait in Loop

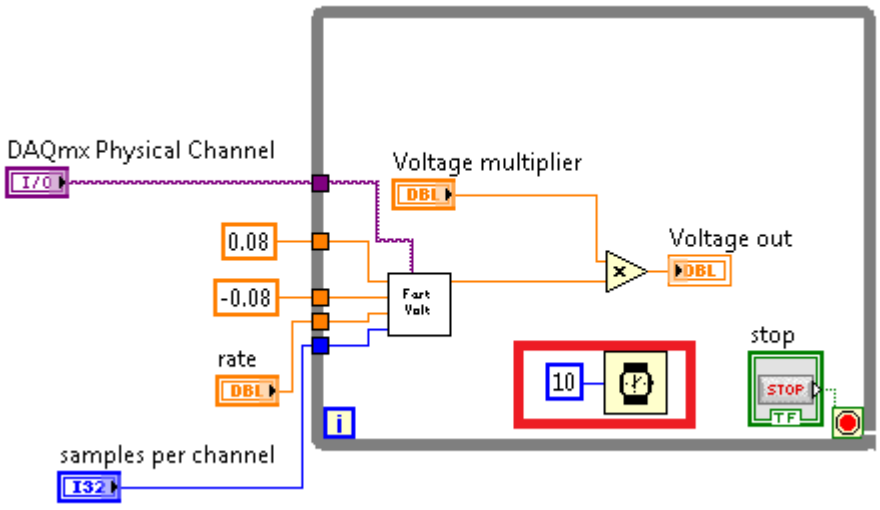

Figure 4: LabVIEW Program after a transformation has been applied to fix No Wait in Loop. (Transformation is highlighted)

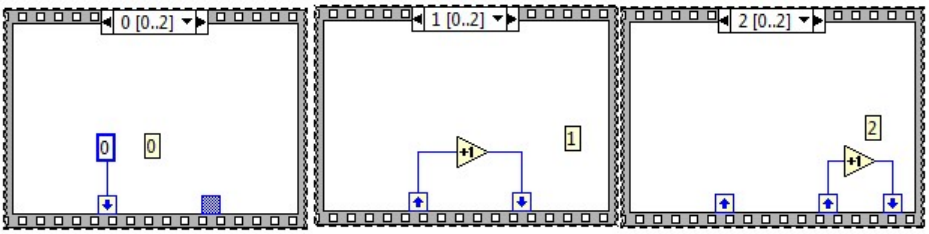

Figure 5: Three pieces of code, run one after another, in a sequence structure. The IDE actually shows only one of these three at a time. To move among the three frames, the programmer clicks on the arrows at the top of the structure. Data values are passed in and out of frames via shift registers, shown as arrows near the bottom of the structure.

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 6: Transformation for Sequence Structure

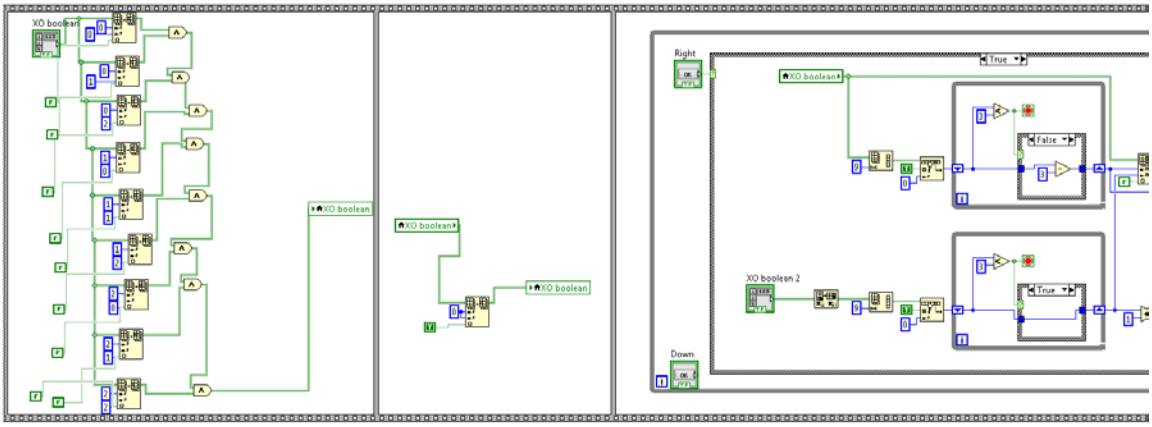

Figure 7: Sequence Structure in a program before transformation. Only the first three frames are shown

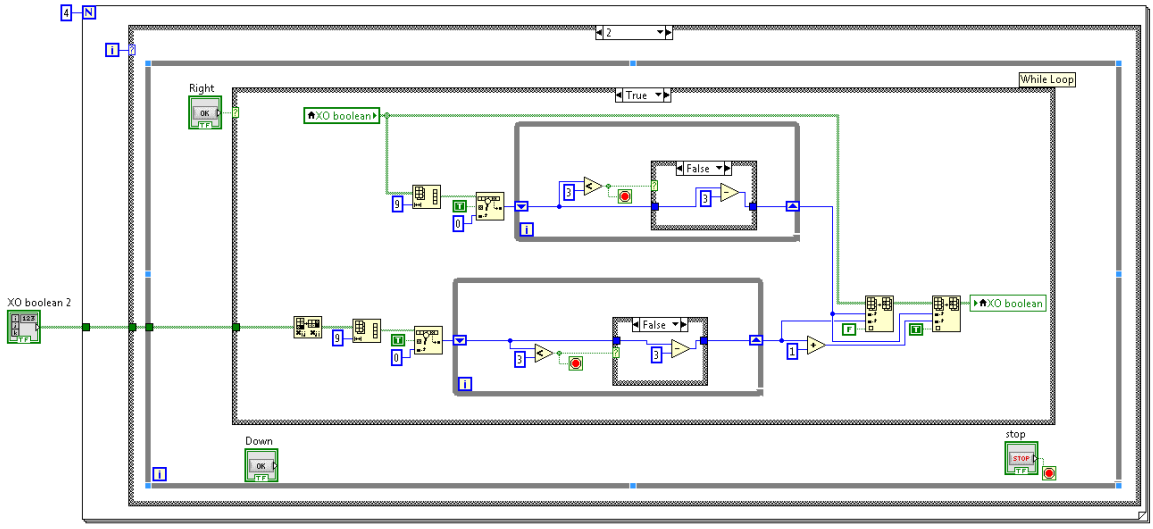

Figure 8: Sequence Structure after being transformed

For an Infinite 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 9: Transformation for Infinite While Loop

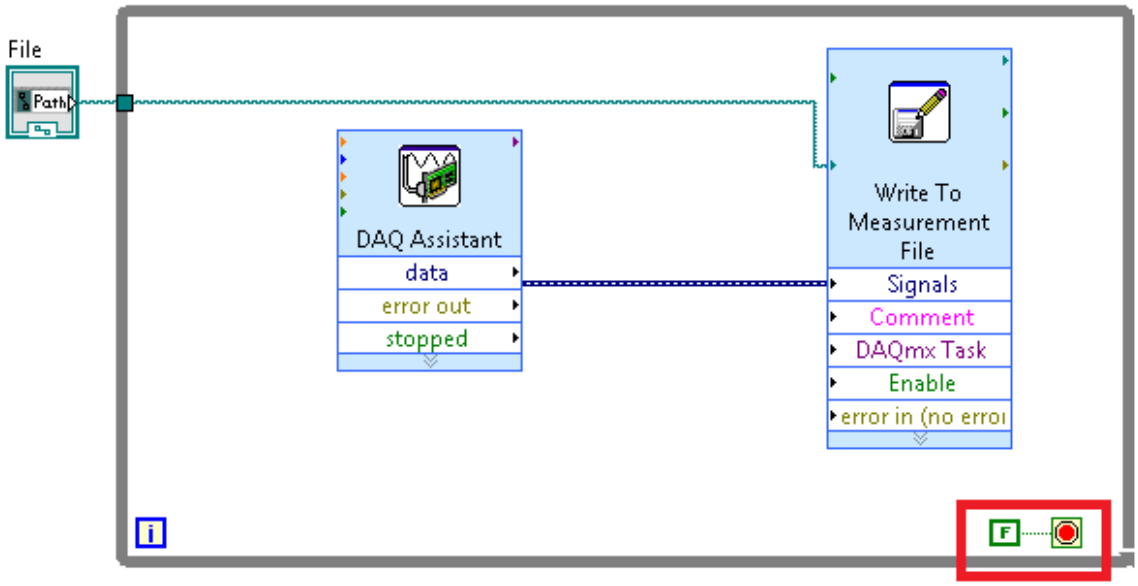

Figure 10: LabVIEW program containing an infinite While Loop. (Smell is highlighted)

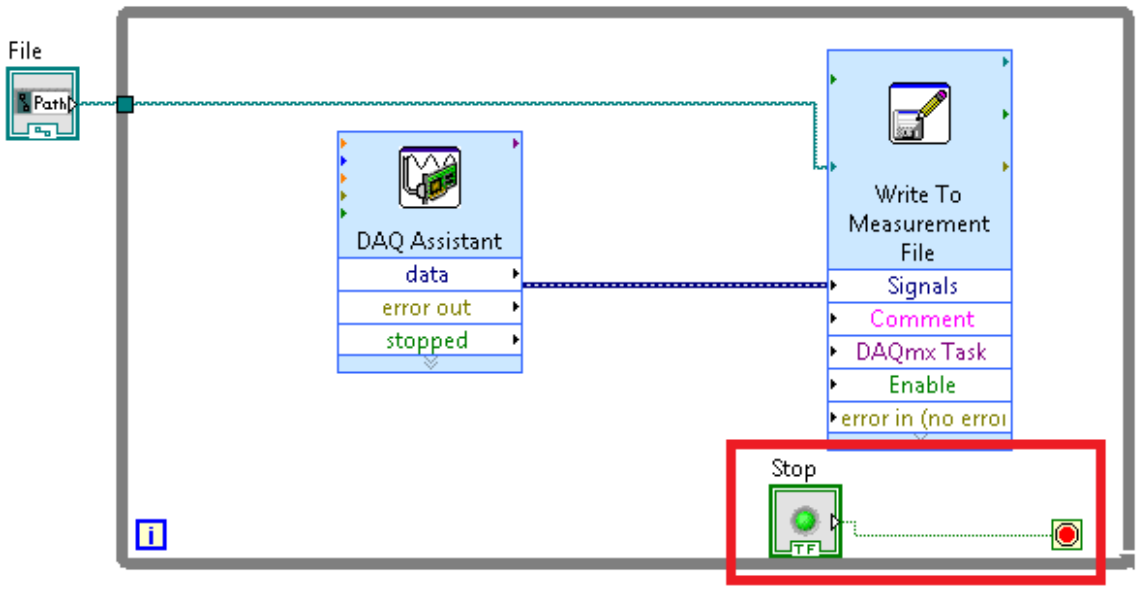

Figure 11: Transformed LabVIEW program no longer containing an Infinite While Loop (Transformation is highlighted)

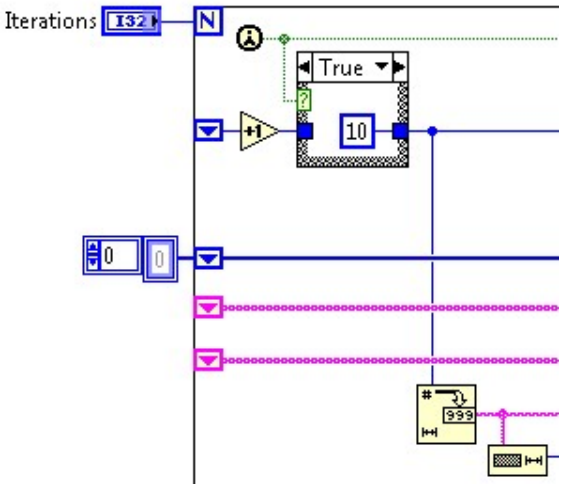

Figure 12: LabVIEW example showing initialized (second blue shift arrow node) and uninitialized (the remaining three) shift registers

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 13: Transformation for Uninitialized Shift Register

For an identified Redundant Operation:

1. Delete the unnecessary (repeated) nodes
2. Rewire the VI so that the output nodes are connected to their inputs

Figure 14: Transformation for Redundant Operations

For a Build Array in a Loop:

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 15: Transformation for Build Array in a Loop

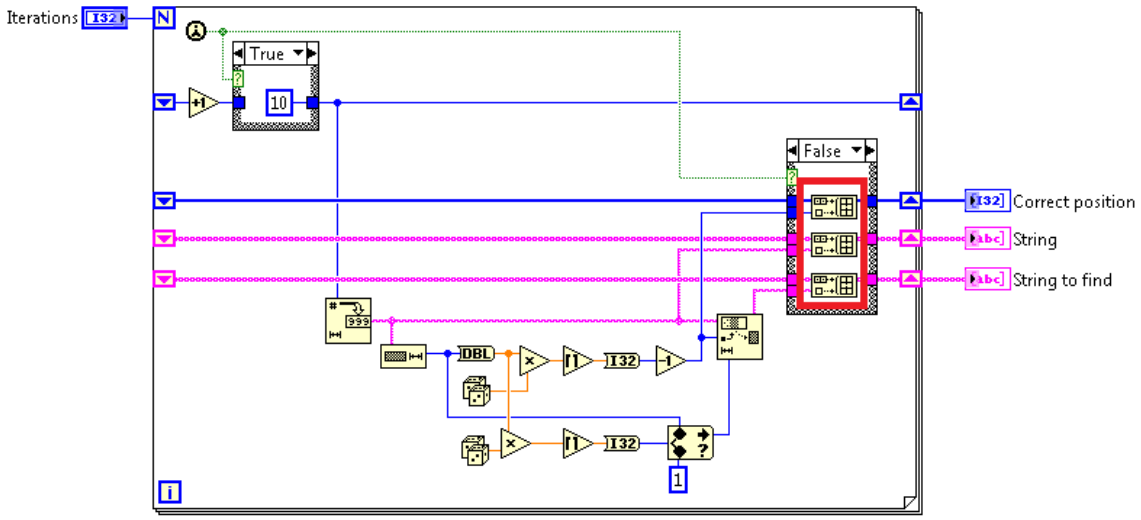

Figure 16: LabVIEW program showing three Build Array Nodes in a For Loop

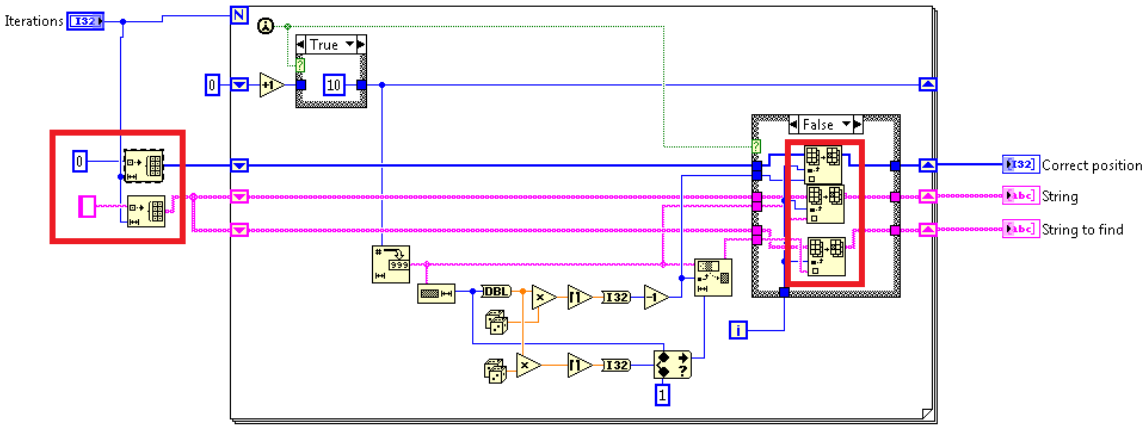

Figure 17: Transformed LabVIEW example showing all Build Array Nodes transformed. Left highlight shows initialization and right highlight shows Replace Array Subset (Requiring user involvement to set the array size) and the strings are added to the array using Replace Array Subset

For a String Concatenation in a Loop:

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 18: Transformation for String Concatenation in a Loop

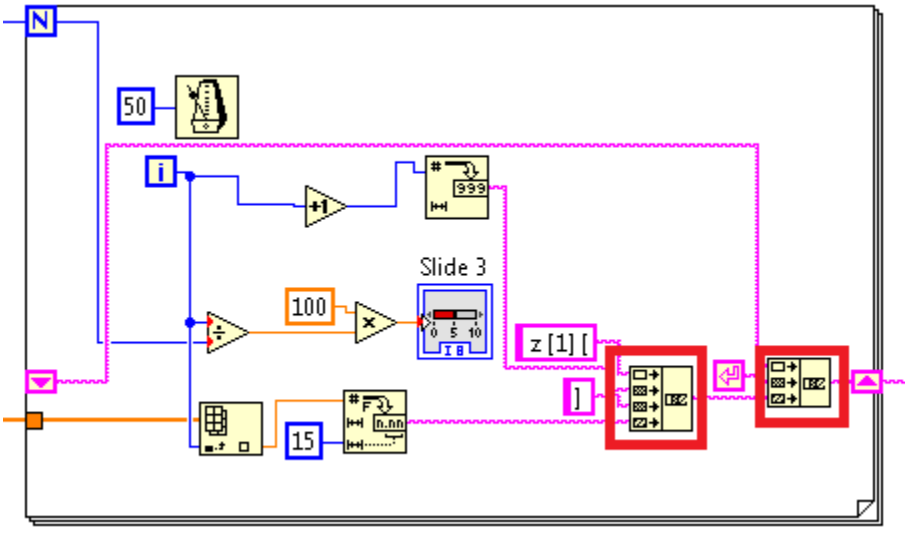

Figure 19: LabVIEW Program showing a String Concatenation Node in a Loop.

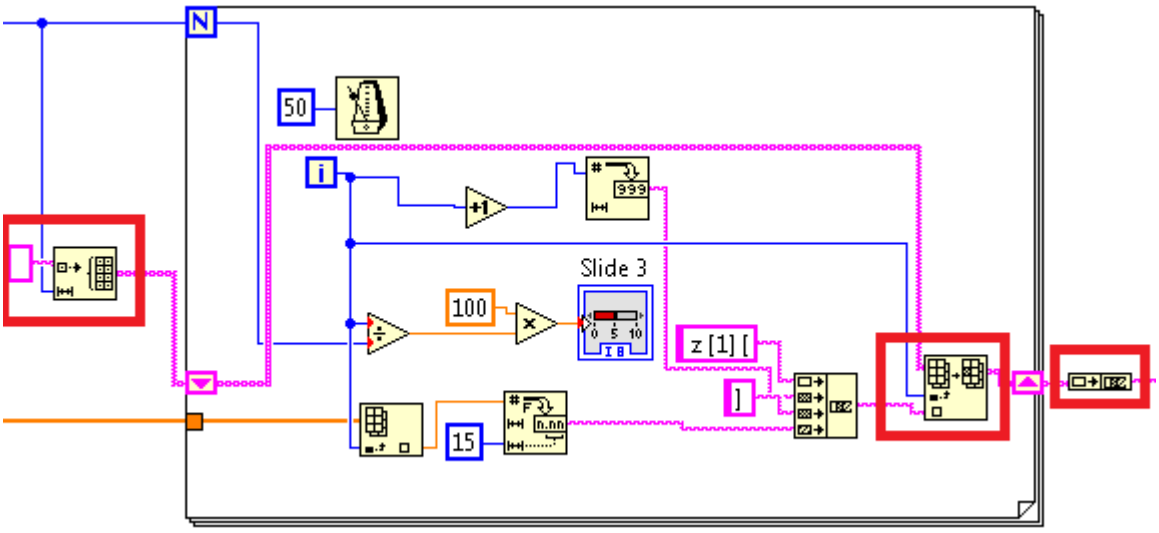

Figure 20: Transformed LabVIEW example removing String Concatenation Node in Loop transformed. Left highlight shows initialization and Middle highlight shows Replace Array Subset, Right highlight shows string concatenation

For a specific instance of Array Copy:

1. Highlight nodes that cause the problem
2. If Array Copy is caused by wire branch into structure
    a. Describe why this causes a problem
    b. Give programmer fix (Only one wire into structure, split occurs inside structure)
3. Else if Array Copy is caused by wire branch not related to a structure
    a. Describe why this is a problem
    b. Provide possible refactorings
4. Else if Array Copy is in a loop
    a. Describe the problem
    b. Discuss ways to move it outside of the loop
5. Else:
    a. Describe general problem with Array Copies
    b. Provide general solutions that may work, such as In-place structures

Figure 21: Transformation for Multiple Array Copy

For a Non-Reentrant SubVI:

1. Describe the potential issue that this can cause
2. Describe why setting the subVI to Reentrant can help performance
3. Provide step-by-step guide to changing the subVI setting to Reentrant as well as changing the priority.

Figure 22: Transformation for a Non-Reentrant SubVI

For a Queue with no constraint:

1. Highlight Queue
2. Provide Information about constraints and why they are useful to apply to Queues
3. If programmer decides to add a constraint
    a. Ask how large they would like the constraint to be
    b. Ask what they would like to do when the constraint is hit. Explain options.
4. Provide step-by-step guide to help the user implement a constraint based on the answers provided by the programmer

Figure 23: Transformation for a Queue with no constraint

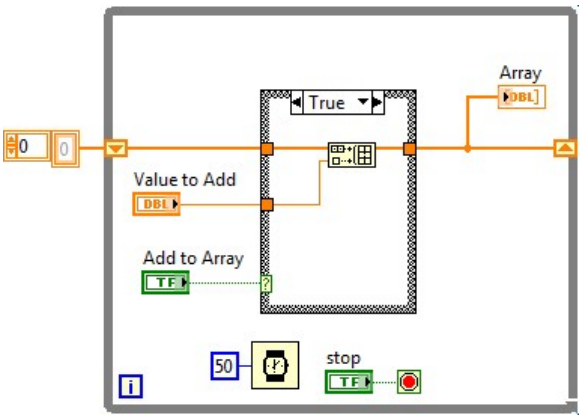

Figure 24: LabVIEW example demonstrating terminals (Add to Array Boolean) that must be included in a structure (while loop) for the program to execute correctly

For a Structure with Terminals:

1. Determine the type of structure and the type of terminal
2. Explain the problems that terminals can cause in a structure, as well as why they should be moved outside. Also explain that all of them need to be moved for the full benefit
3. Explain cases where terminals cannot be moved
4. If the structure is a loop
    a. Describe why certain terminals can cause problems inside of a loop and why it can be helpful to move them outside of the loop
    b. If terminal is a graph or array, explain why these are often more problematic to performance
5. If structure is anything else
    a. Describe that these cases are less problematic, but there may still be some benefit

Figure 25: Transformation for Terminals in Structure

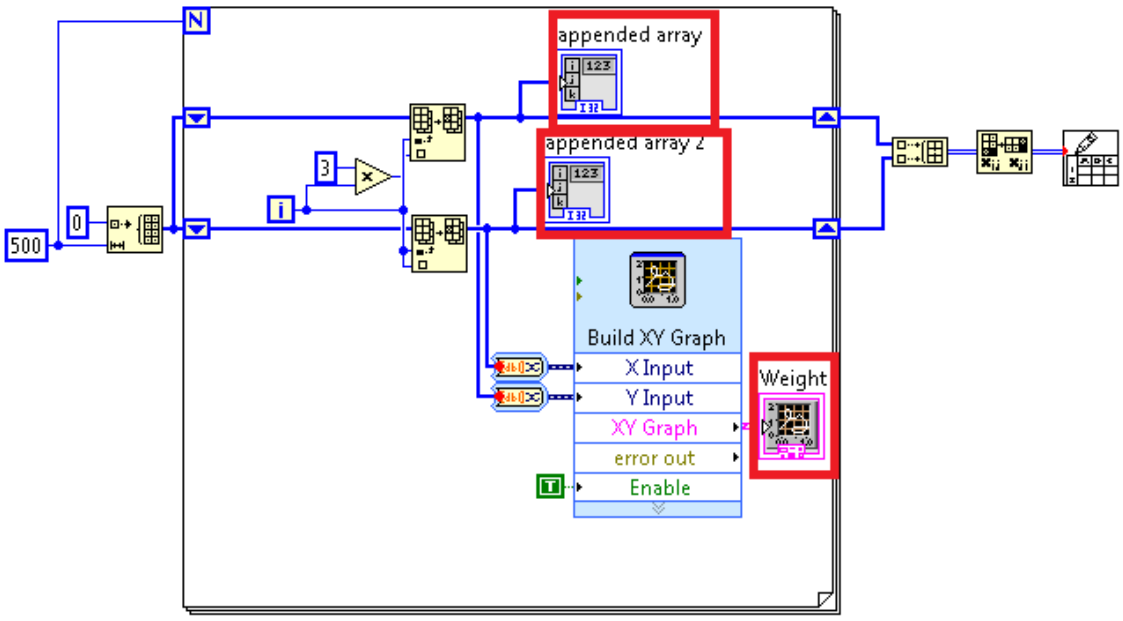

Figure 26: LabVIEW program showing three terminals inside of the loop. All terminals are highlighted in red

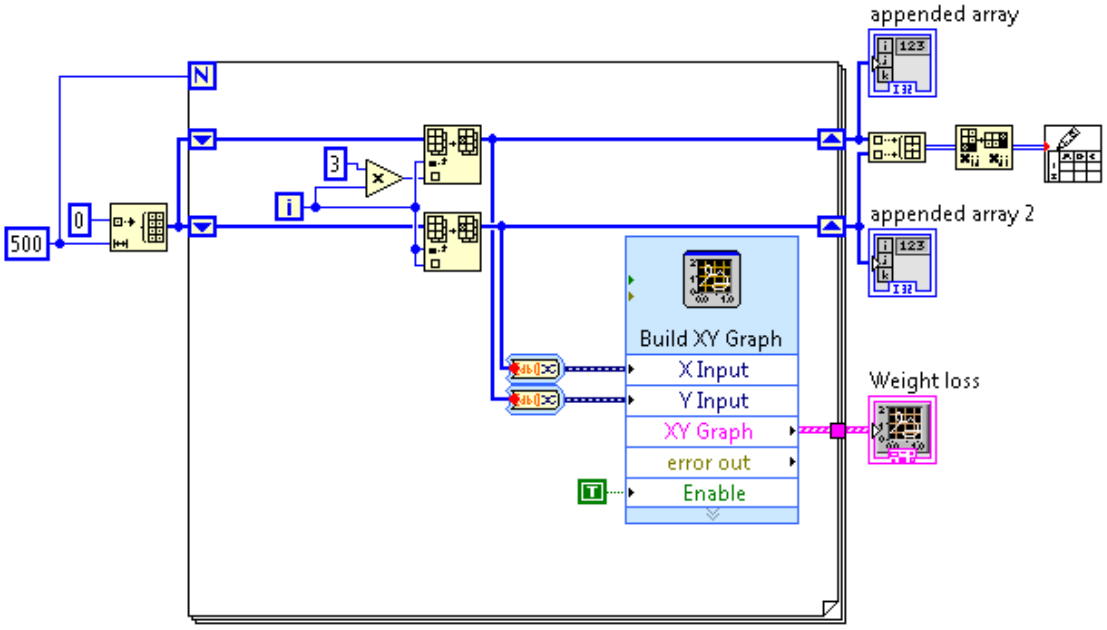

Figure 27: Transformed LabVIEW program showing three terminals moved outside of the loop

For a VI with too many variables:

1. Highlight all of the variables (both read and write) in the VI
2. Describe the problems this case cause in the VI and why it is better to not use variables
3. Describe potential ways to remove them such as:
    a. Using Shift Registers on Loops
    b. Replacing variables with Functional Global Variables
    c. Refactoring the code completely

Figure 28: Transformation for Too Many Variables