

# Improving the Performance of Your NI LabVIEW Applications

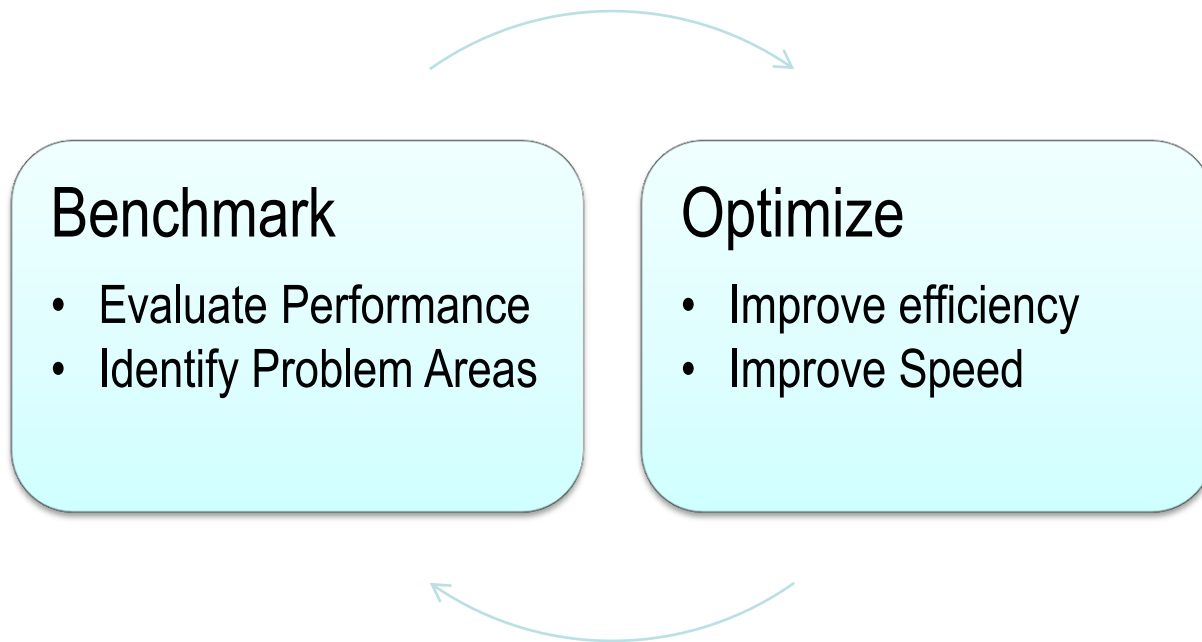
Dan Hedges

Senior Software Engineer  
LabVIEW Performance Group

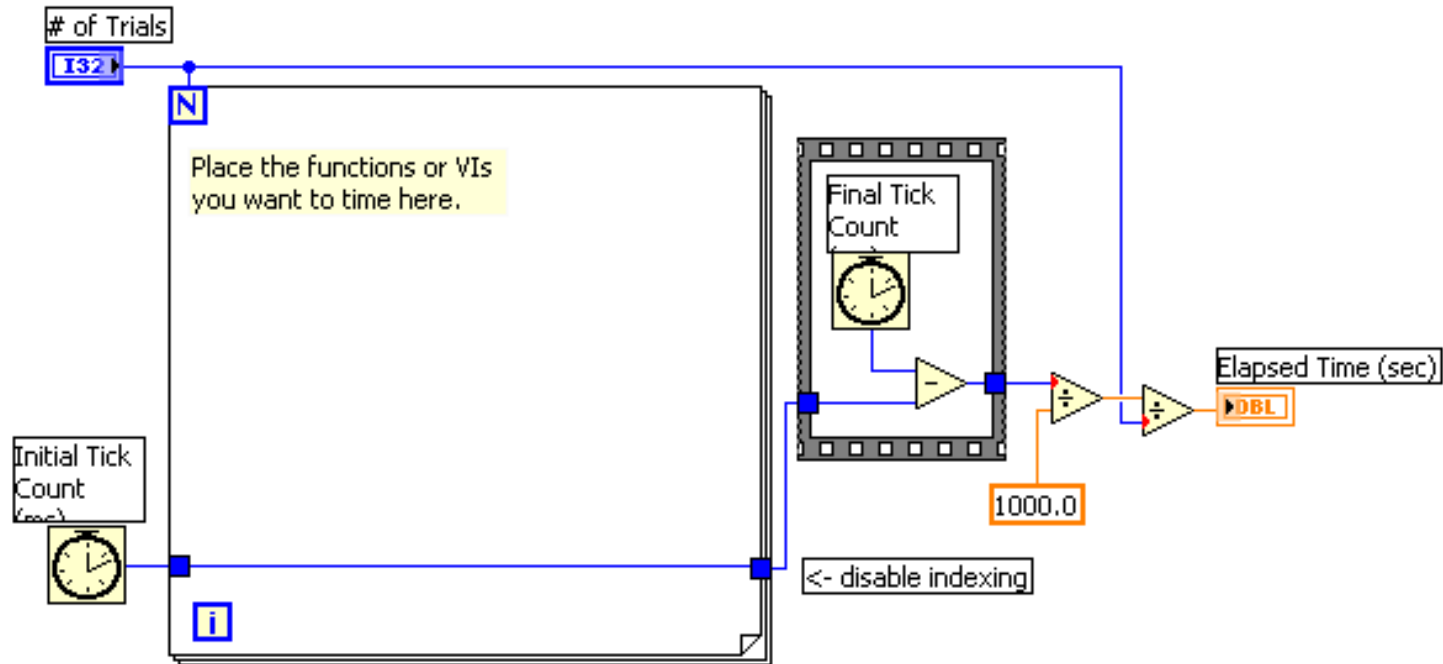
# Agenda

- How to find performance problems
  - Benchmarking
  - Profiling
- Understanding LabVIEW under the hood
  - Memory usage
  - Execution system

# Optimization Cycle

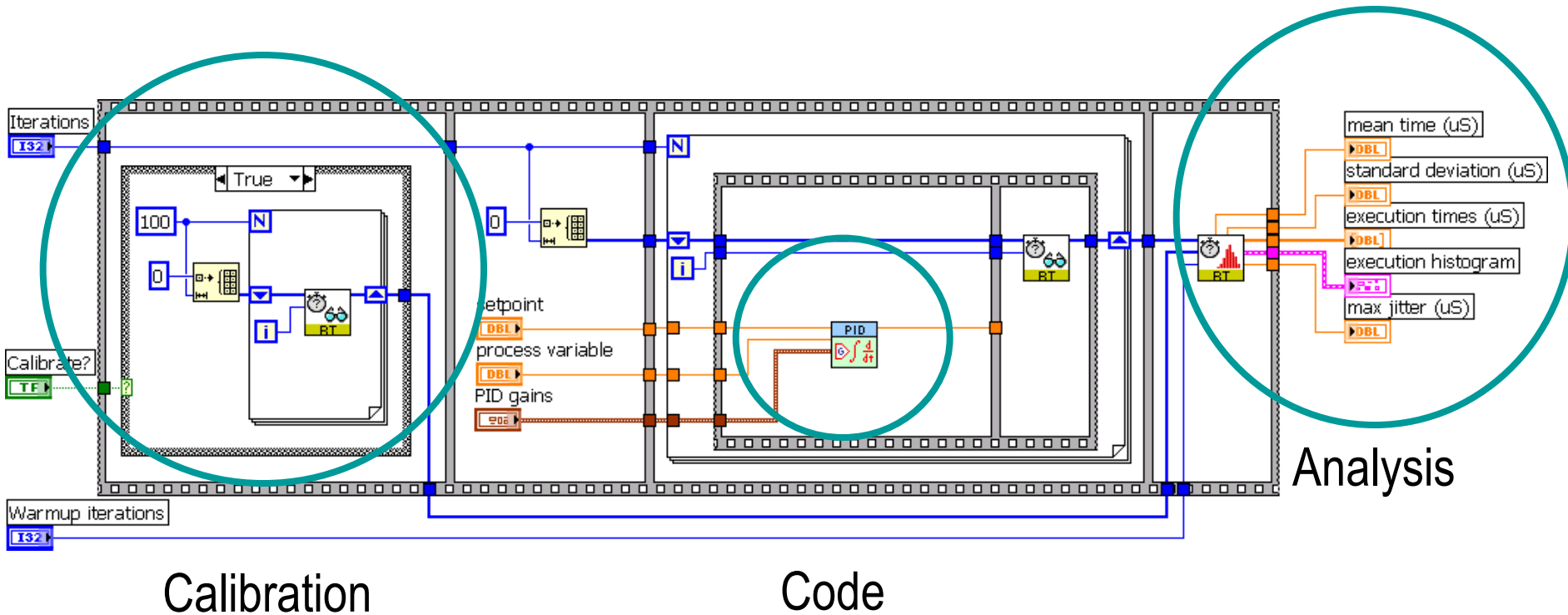


# Benchmarking Code Execution



“Timing Template (data dep)” – LabVIEW Shipping Example

# Benchmarking Code Execution



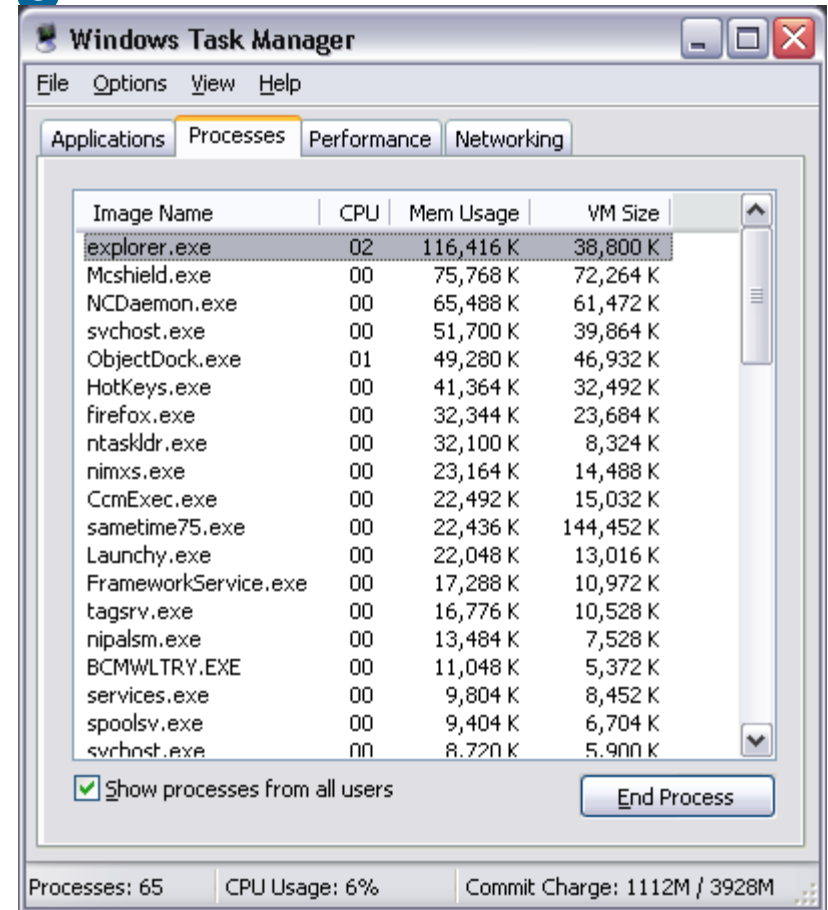
“Benchmark Project” – LabVIEW Real-Time Shipping Example

# Tools for Measuring Resource Usage (Windows)

- Task Manager
- Perfmon

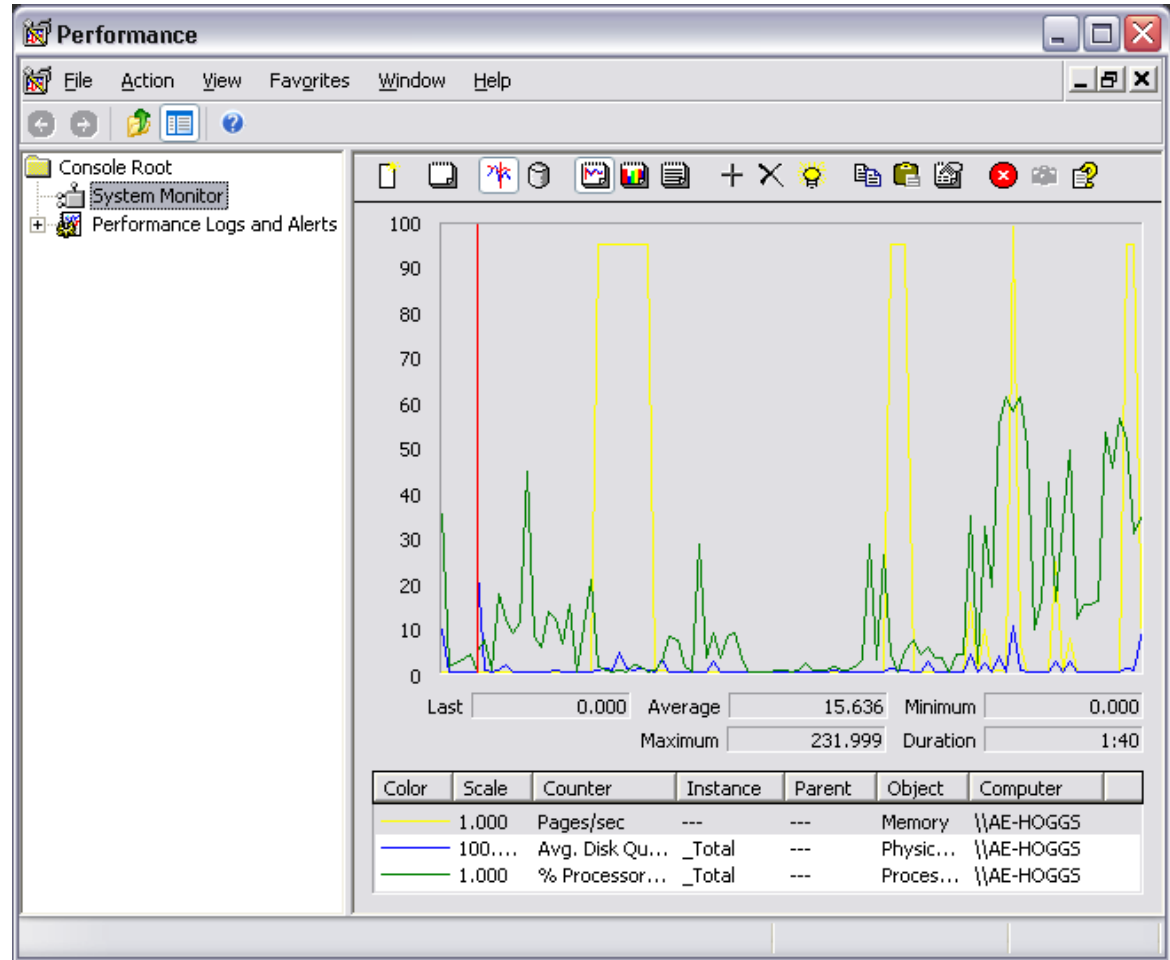
# Windows Task Manager

- Gives user a rough idea of whether memory or CPU is the bottleneck
- Can be helpful in identifying memory leaks
- **View»Select Columns ...** allows you to add additional stats



# Perfmon

- Allows you to monitor
  - Processors
  - Disk I/O
  - Network Tx/Rx
  - Memory/Paging
- Access by typing “**perfmon**” into the Windows Run dialog





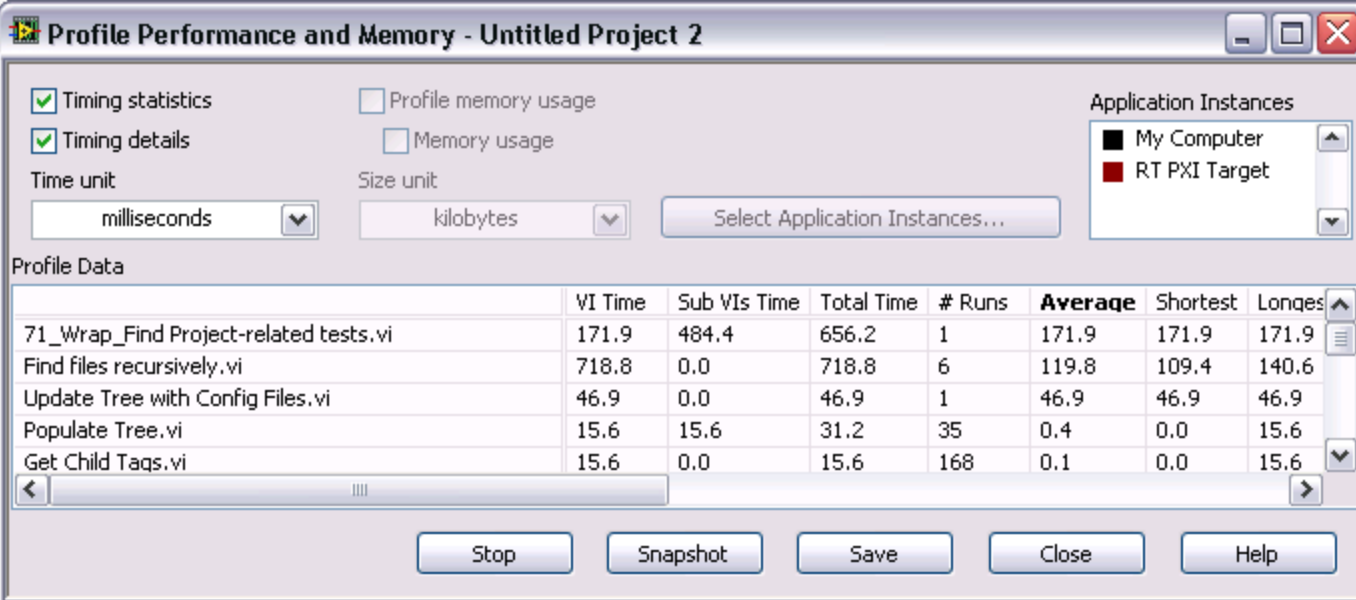
# Why Should You Profile Your VIs?

## The 80/20 rule of software performance

- 80 percent of the execution time is spent in 20 percent of the code
- Performance improvements are most effective in the 20 percent
- Guessing which 20 percent is difficult

# VI Profiler

- Tools >> Profile >> Performance and Memory...



The screenshot shows the 'Profile Performance and Memory - Untitled Project 2' window. It includes checkboxes for 'Timing statistics' and 'Timing details' (both checked), and 'Profile memory usage' and 'Memory usage' (both unchecked). The 'Time unit' is set to 'milliseconds' and the 'Size unit' is 'kilobytes'. The 'Application Instances' list shows 'My Computer' and 'RT PXI Target'. The 'Profile Data' table is as follows:

|                                       | VI Time | Sub VIs Time | Total Time | # Runs | Average | Shortest | Longest |
|---------------------------------------|---------|--------------|------------|--------|---------|----------|---------|
| 71_Wrap_Find Project-related tests.vi | 171.9   | 484.4        | 656.2      | 1      | 171.9   | 171.9    | 171.9   |
| Find files recursively.vi             | 718.8   | 0.0          | 718.8      | 6      | 119.8   | 109.4    | 140.6   |
| Update Tree with Config Files.vi      | 46.9    | 0.0          | 46.9       | 1      | 46.9    | 46.9     | 46.9    |
| Populate Tree.vi                      | 15.6    | 15.6         | 31.2       | 35     | 0.4     | 0.0      | 15.6    |
| Get Child Tags.vi                     | 15.6    | 0.0          | 15.6       | 168    | 0.1     | 0.0      | 15.6    |

# Demo – VI Profiling

# LabVIEW Desktop Execution Trace Toolkit



- Detailed execution traces
- Thread and VI information
- Measurement of execution time

The screenshot displays the Desktop Execution Trace Toolkit interface. On the left, a tree view shows the trace data for a 'Main Application Instance' with two sessions: '2/1/2010 - 10:59:29.401335' and '2/1/2010 - 11:02:29.235938'. A light blue box labeled 'Multiple Sessions' is overlaid on the tree view. The main area shows a table of execution events for the second session. The table has columns for '#', 'Time', 'VI', 'Event', 'Thread', 'CPU Id', and 'Details'. The events include 'VI Start Execution', 'VI Call', 'Memory Resize', 'Memory Free', and 'Memory Allocate'. A light blue box labeled 'Threads, CPU and Memory' is overlaid on the table. A light blue box labeled 'VIs' is overlaid on the 'VI' column. The table data is as follows:

| #  | Time            | VI                       | Event              | Thread | CPU Id | Details              |
|----|-----------------|--------------------------|--------------------|--------|--------|----------------------|
| 0  | 11:02:29.235938 | 0 - Main.vi              | VI Start Execution | 5      | 4      |                      |
| 1  | 11:02:29.235943 | 0 - Main.vi              | VI Call            | 5      | 4      |                      |
| 2  | 11:02:29.235947 | 0 - Main.vi              | Memory Resize      | 5      | 4      | Handle: 0x404B7C4; C |
| 3  | 11:02:29.235950 | 0 - Find Optimal Move.vi | VI Call            | 5      | 4      |                      |
| 4  | 11:02:29.235958 | 0 - Find Optimal Move.vi | Memory Resize      | 5      | 4      | Handle: 0x404B7C8; C |
| 5  | 11:02:29.235961 | 0 - Find Optimal Move.vi | Memory Resize      | 5      | 4      | Handle: 0x404B7CC; C |
| 6  | 11:02:29.235962 | 0 - Find Optimal Move.vi | Memory Resize      | 5      | 4      | Handle: 0x404B7D0; C |
| 7  | 11:02:29.235964 | 0 - Find Optimal Move.vi | Memory Resize      | 5      | 4      | Handle: 0x404B7D4; C |
| 8  | 11:02:29.235965 | 0 - Find Optimal Move.vi | Memory Free        | 5      | 4      | Handle: 0x404B7D4; S |
| 9  | 11:02:29.235966 | 0 - Find Optimal Move.vi | Memory Free        | 5      | 4      | Handle: 0x404B7D0; S |
| 10 | 11:02:29.235968 | 0 - Find Optimal Move.vi | Memory Allocate    | 5      | 4      | Handle: 0x404B7D0; S |
| 11 | 11:02:29.235969 | 0 - Find Optimal Move.vi | Memory Allocate    | 5      | 4      | Handle: 0x404B7D4; S |
| 12 | 11:02:29.235971 | 0 - Find Optimal Move.vi | Memory Resize      | 5      | 4      | Handle: 0x404B7DC; C |
| 13 | 11:02:29.235973 | 0 - Test Rotate.vi       | VI Call            | 5      | 4      |                      |
| 14 | 11:02:29.235974 | 0 - Test Rotate.vi       | Memory Resize      | 5      | 4      | Handle: 0x404B7E0; C |
| 15 | 11:02:29.235975 | 0 - Test Rotate.vi       | Memory Resize      | 5      | 4      | Handle: 0x404B7E0; C |
| 16 | 11:02:29.235976 | 0 - Test Rotate.vi       | Memory Resize      | 5      | 4      | Handle: 0x404B7E4; C |
| 17 | 11:02:29.235976 | 0 - Test Rotate.vi       | VI Return          | 5      | 4      |                      |
| 18 | 11:02:29.235977 | 0 - Find Optimal Move.vi | Memory Resize      | 5      | 4      | Handle: 0x404B7E8; C |
| 19 | 11:02:29.235980 | 0 - Find Optimal Move.vi | Memory Resize      | 5      | 4      | Handle: 0x404B7EC; C |
| 20 | 11:02:29.235980 | 0 - Find Optimal Move.vi | Memory Resize      | 5      | 4      | Handle: 0x404B7E8; C |

# Profiling and Benchmarking Summary

| To answer this question:                 | Use these tools:                        |
|--|---|
| What is my current performance?          | Benchmark VIs                           |
| What are my limiting resources?          | Task Manager, Perfmon                   |
| How much time are each of my VIs taking? | VI Profiler                             |
| In what order are events occurring?      | LabVIEW Desktop Execution Trace Toolkit |

# Under LabVIEW's Hood

Memory  
Management

Execution  
System

# What Is In Memory?

Panel

Diagram

Compiled  
Code

Data

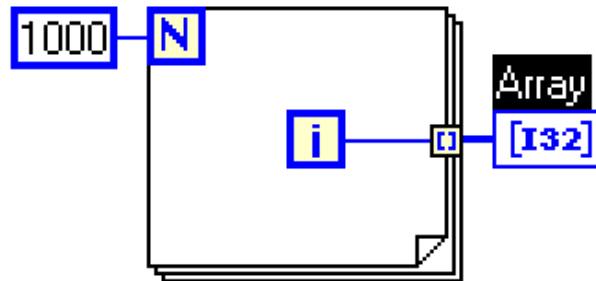
# VIs in Memory

- When a VI is loaded into memory
  - We always load the data
  - We load the code if it matches our platform (x86 Windows, x86 Linux, x86 Mac, PowerPC Mac)
  - We load the panel and diagram only if we need to (for instance, we need to recompile the VI)



# Panel and Diagram Data

- How many bytes of memory does this VI use?
- The answer depends on:
  - Is the panel in memory?
  - Is the environment multi-threaded?



# Execute, Operate and Transfer Data

4K Execute  
Data

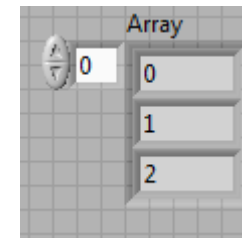
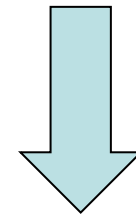
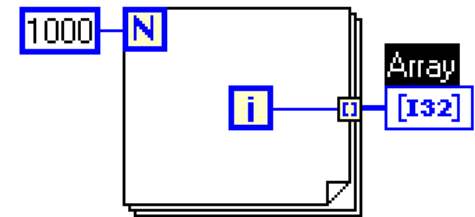
- Populated by Code

4K Transfer  
Data

- Temporary Buffer

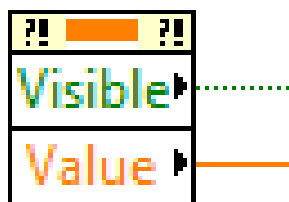
4K Operate  
Data

- Copy for Indicator

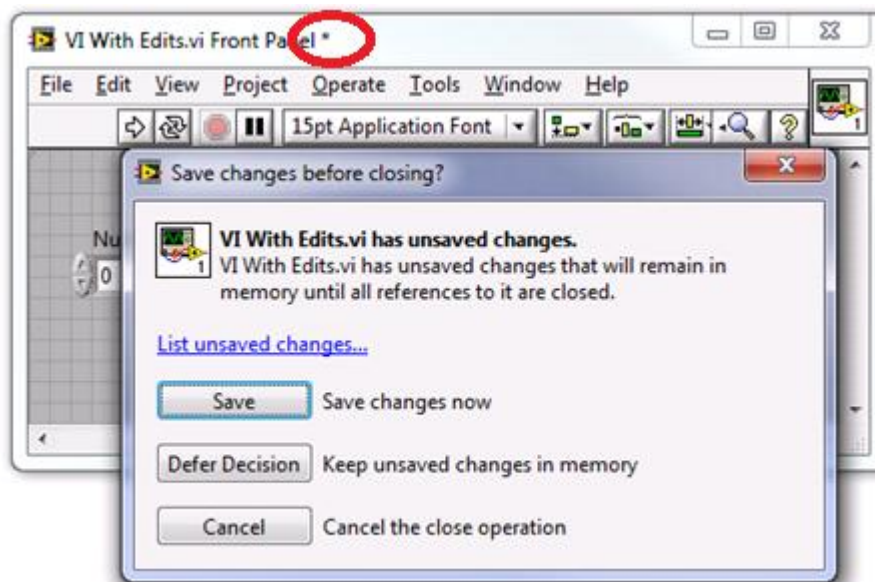


# Avoid Loading Panels, Save Memory

Numeric

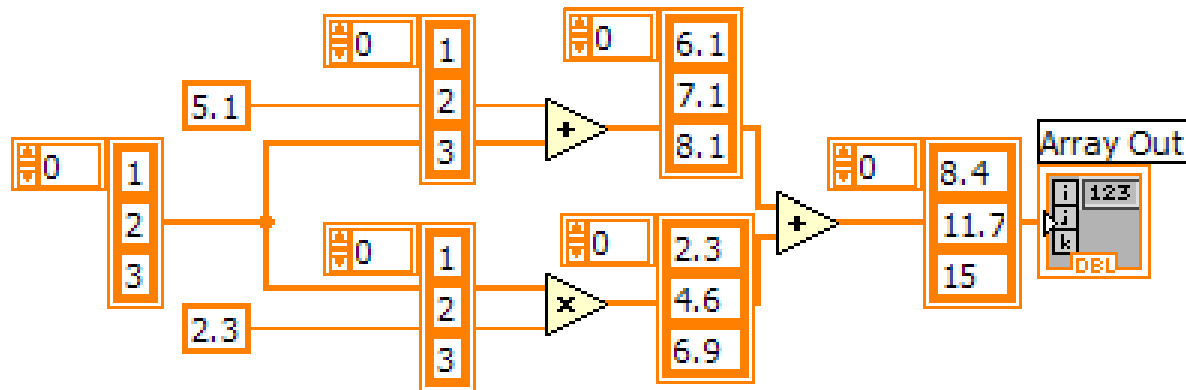


Visible



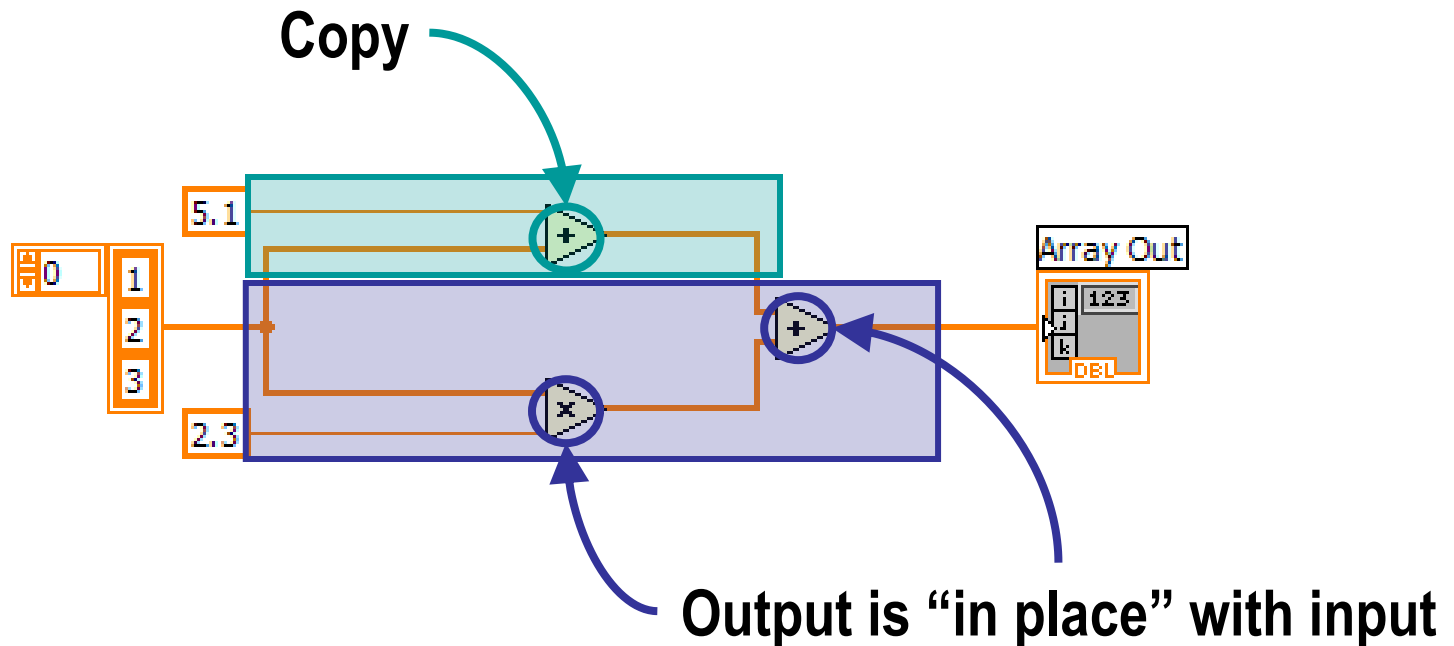
# Wire Semantics

- Every wire is a buffer
- Branches typically create copies



# Optimizations by LabVIEW

The theoretical 5 copies become 1 copy operation

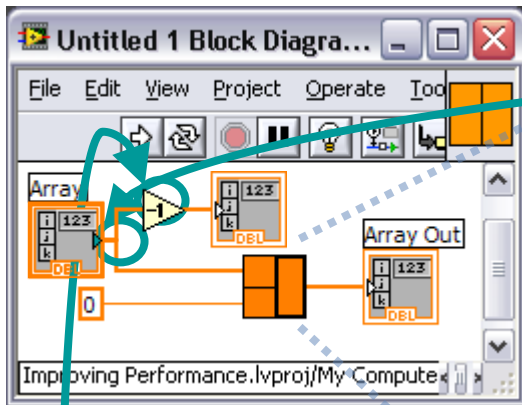


# The “In Place” Algorithm

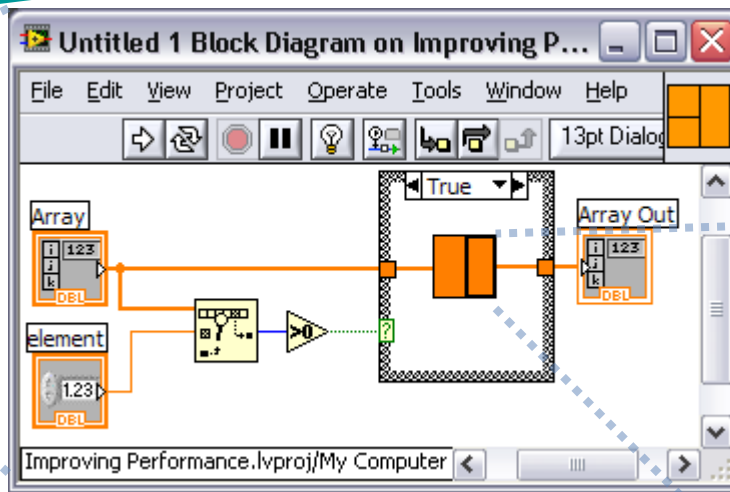
- Determines when a copy needs to be made
  - Weighs arrays and clusters higher than other types
- Algorithm runs during compilation, not execution
  - Does not know the size of an array or cluster
- Relies on the sequential aspects of the program
  - Branches may require copies

# Bottom Up

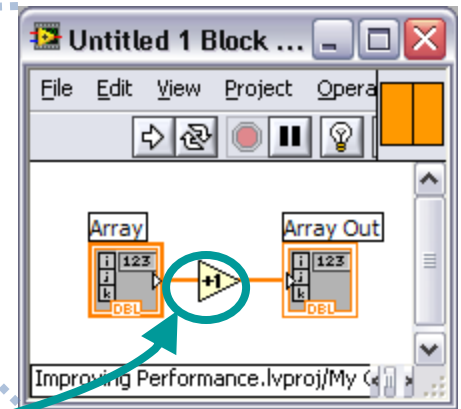
In-place information is propagated bottom up



Copy because of increment



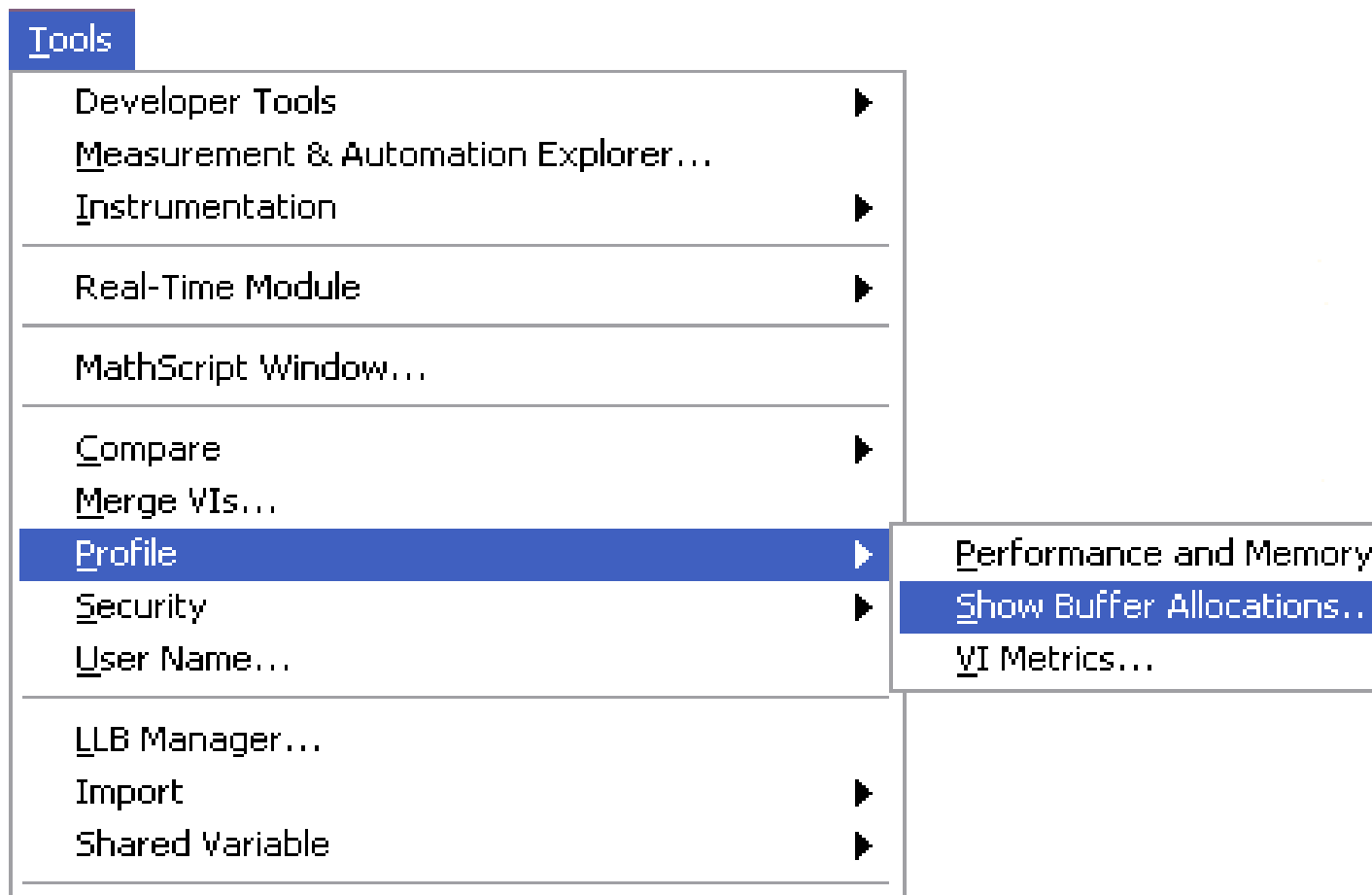
No copies required



Increments array in place

Branched wire

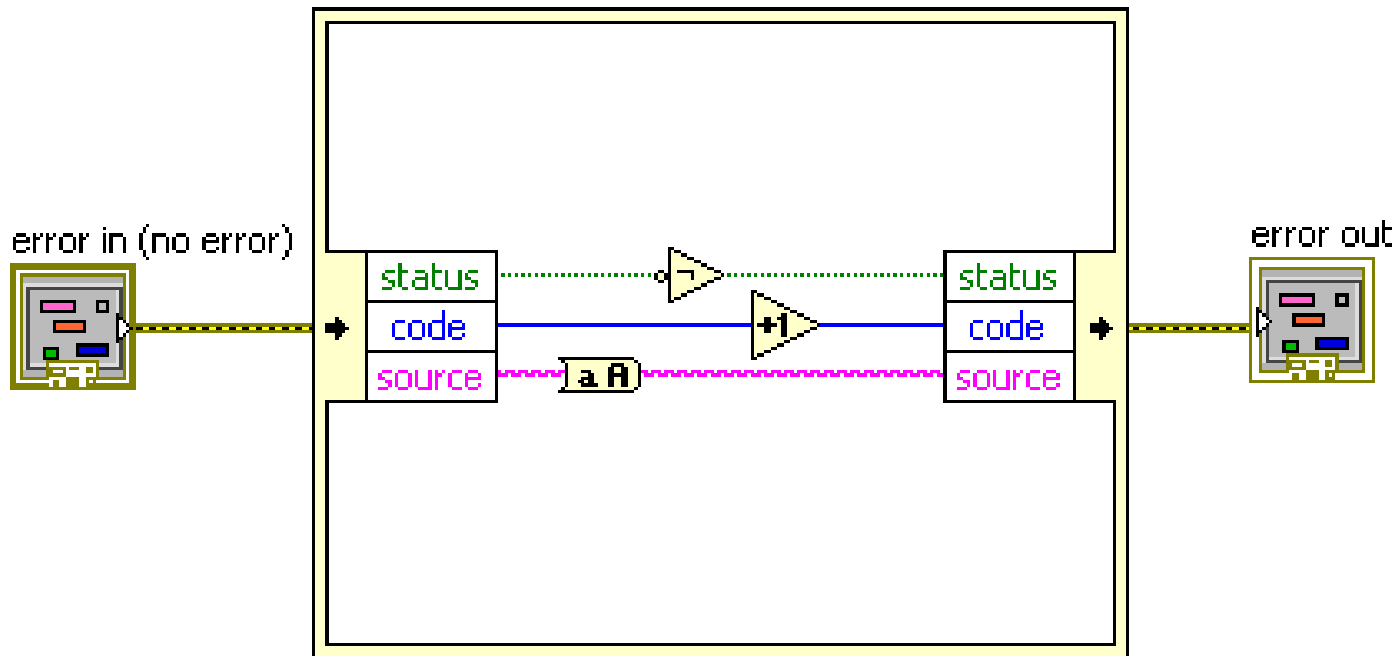
# Showing Buffer Allocations





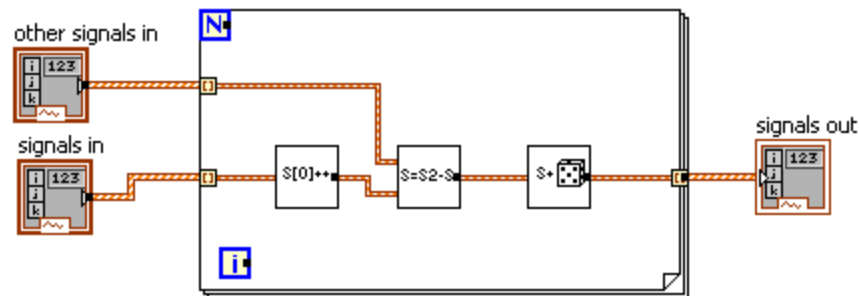
# The In-Place Element Structure

Allows you to explicitly modify data “in place”

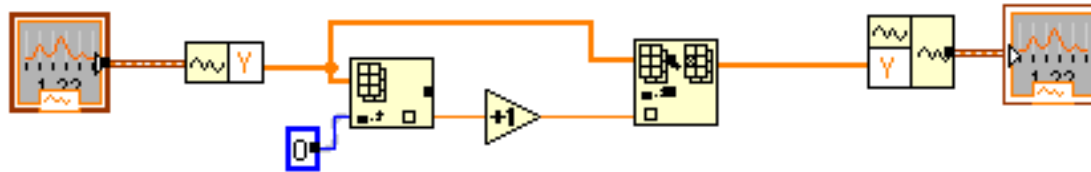


# Example of In Place Optimization

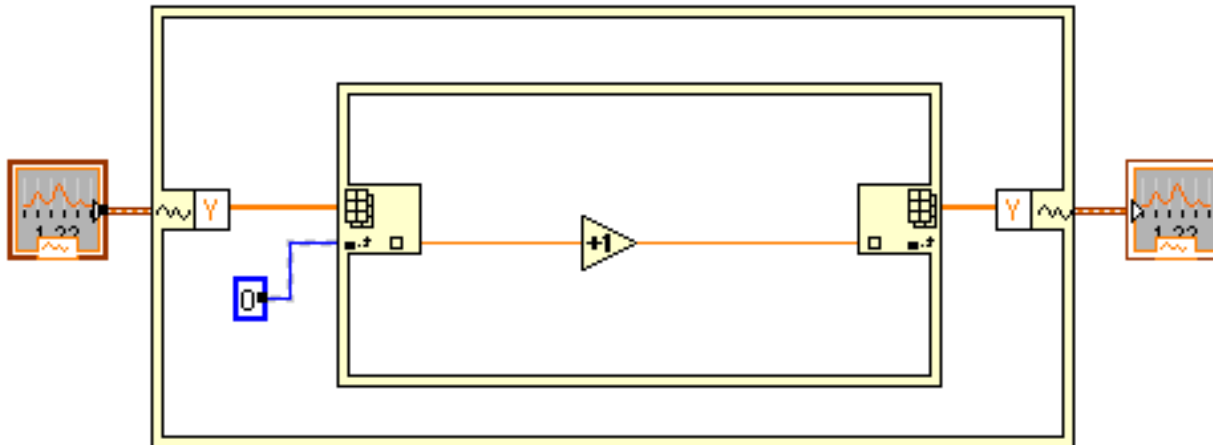
Operate on each element of an array of waveforms



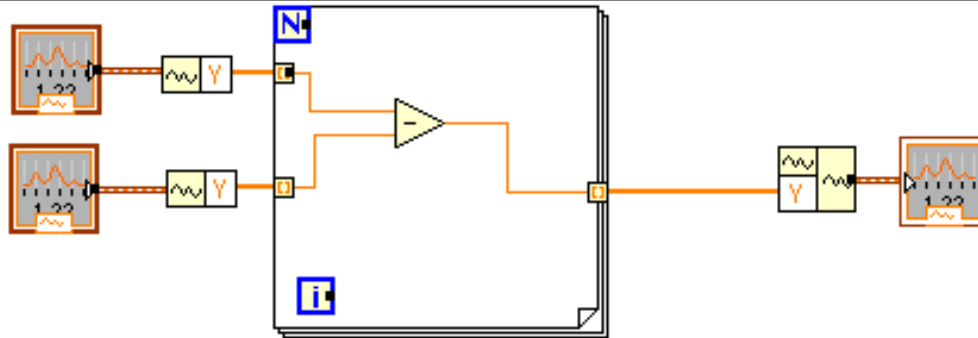
# Make the First SubVI “In Place”



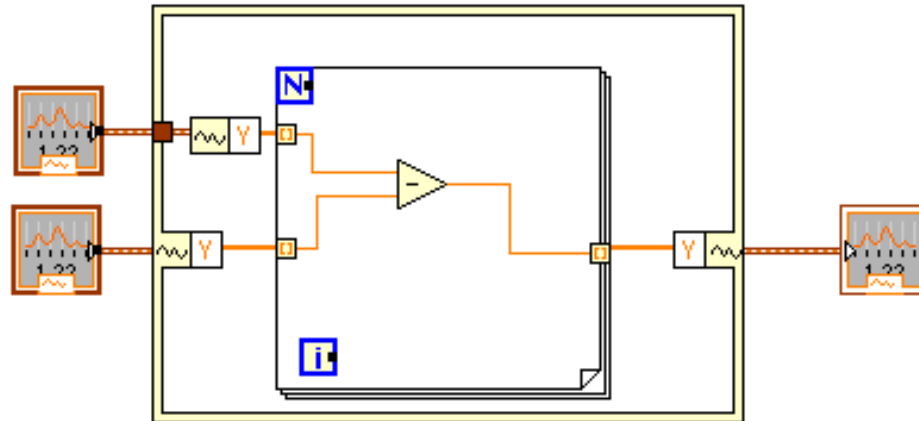
changes into...



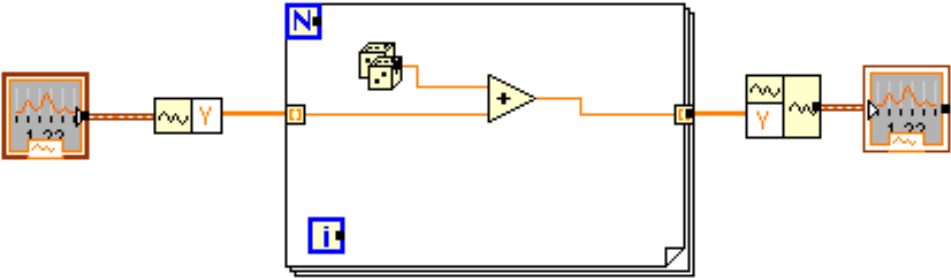
# SubVI 2 Is Made “In Place”



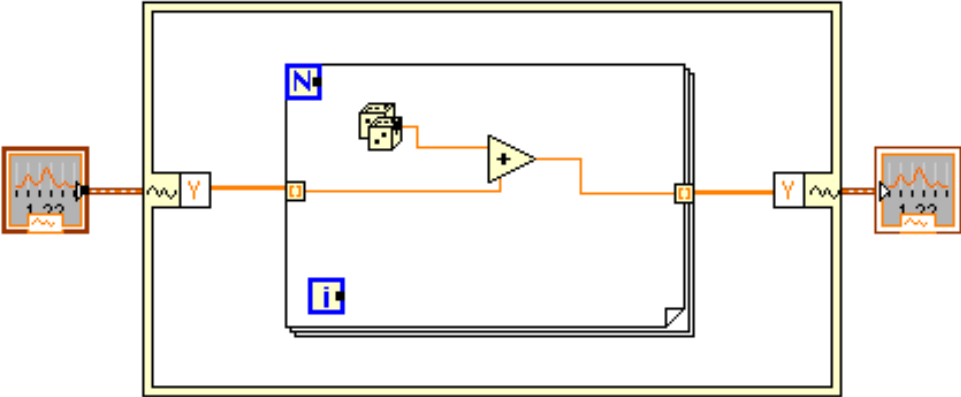
Changes into ...



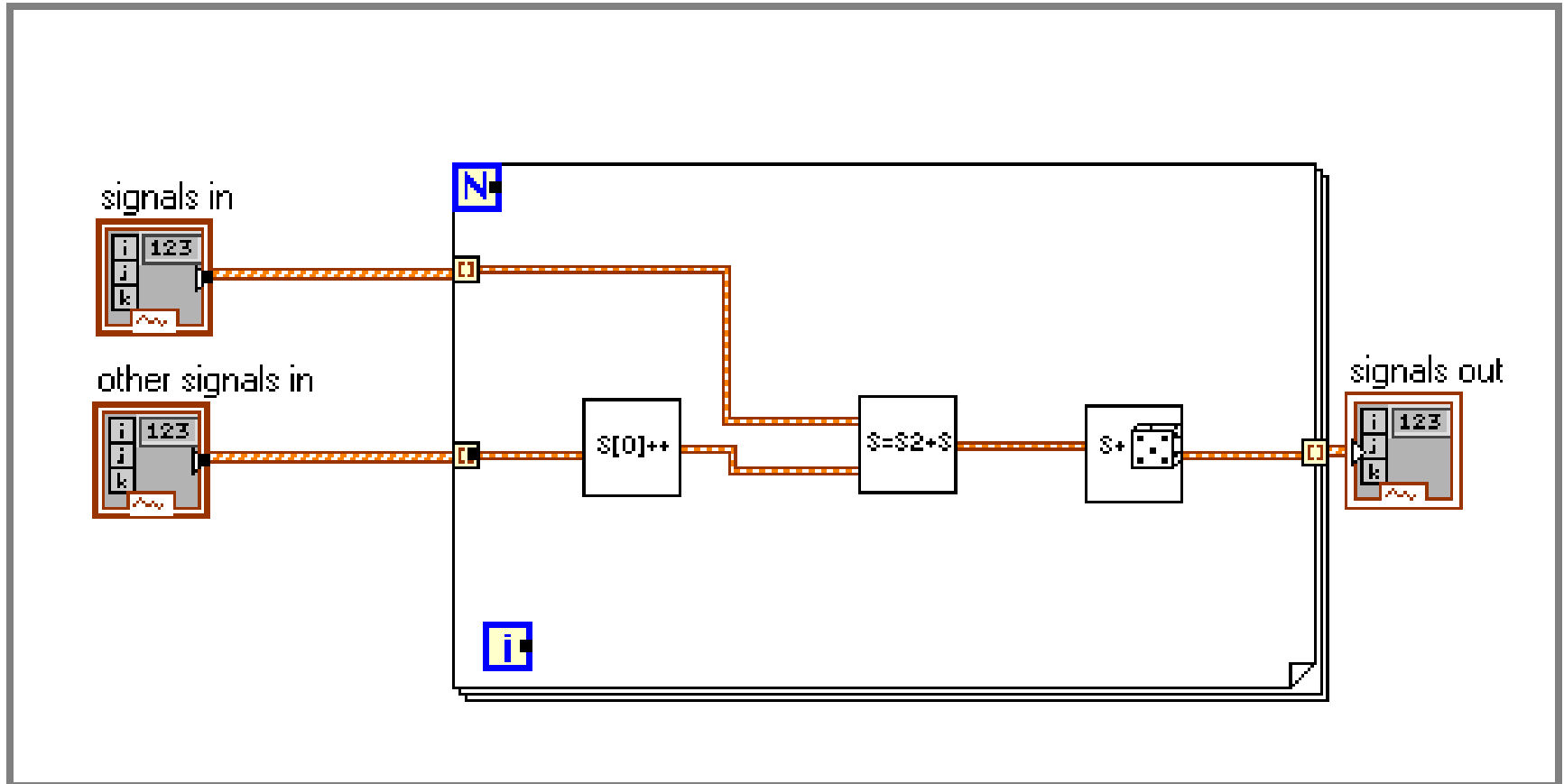
# SubVI 3 Is Made “In Place”



Changes into ...

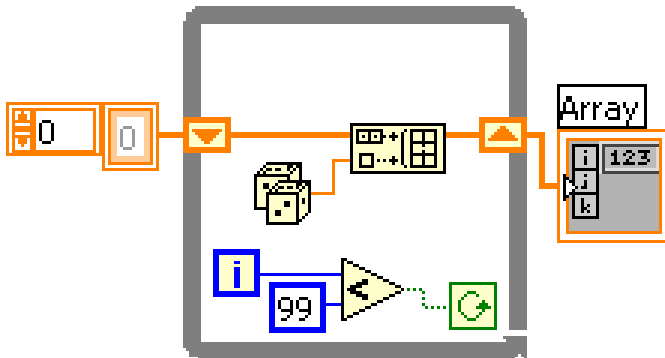


# Final Result: Dots Are Hidden



# Building Arrays

There are a number of ways to build arrays and some are better than others

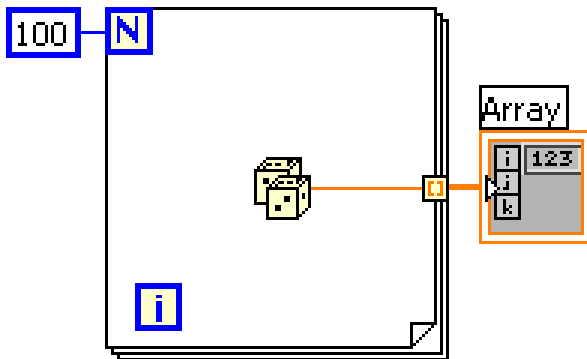


## Bad

- Reallocates array memory on every loop iteration
- No compile time optimization

# Building Arrays

There are a number of ways to build arrays. Try to minimize reallocations.



## Best

- Memory preallocated
- Indexing tunnel eliminates need for copies



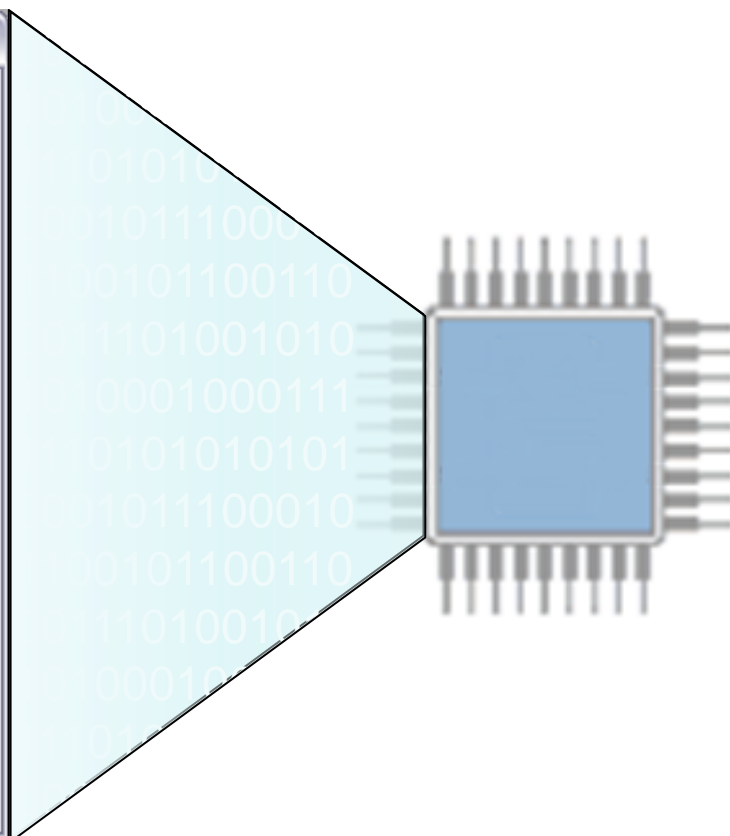
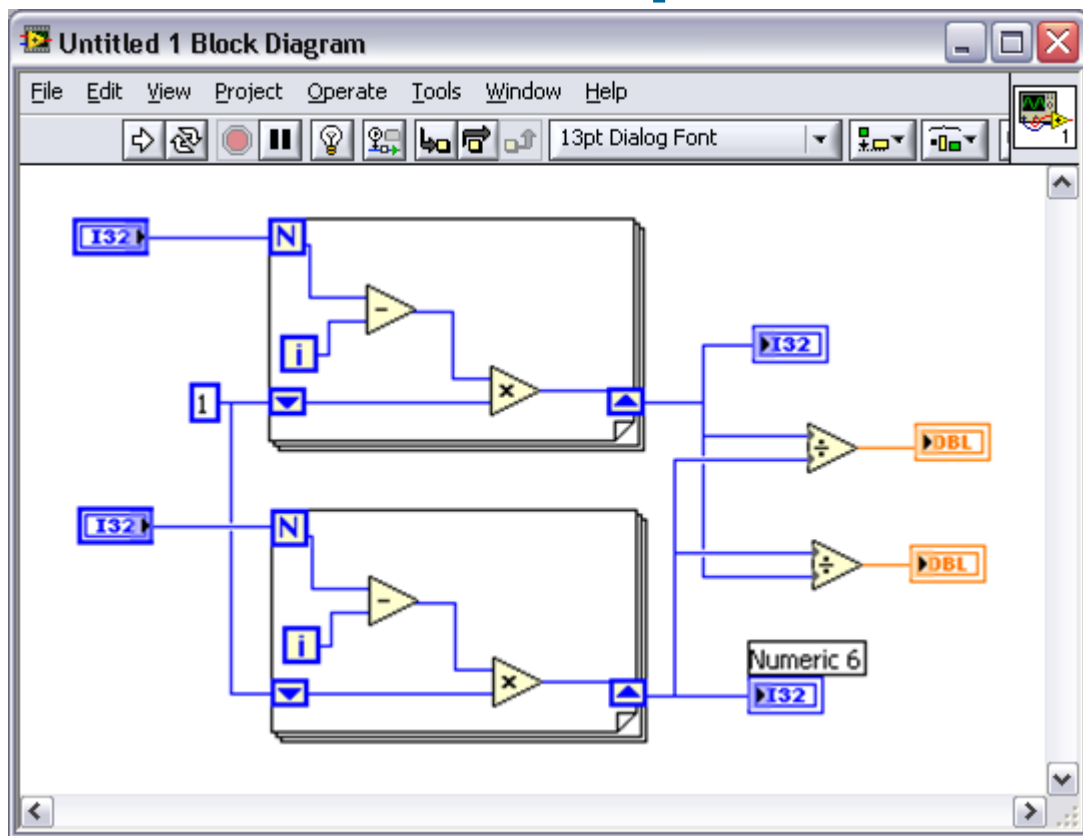
# Demo – Effects of Memory Optimization

# Under LabVIEW's Hood

Memory  
Management

Execution  
System

# VIs Are Compiled

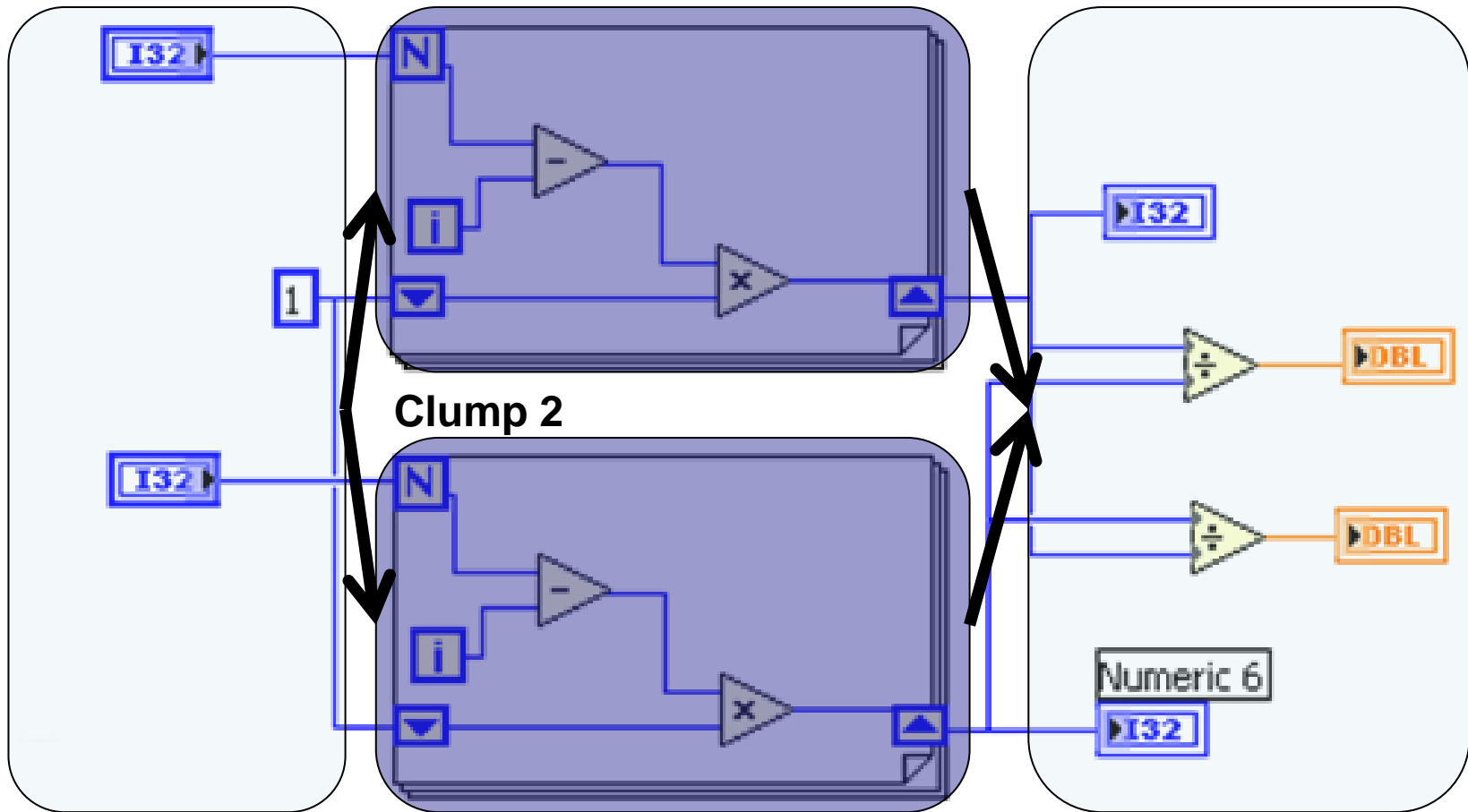


# VIs Are Compiled: "Clumps"

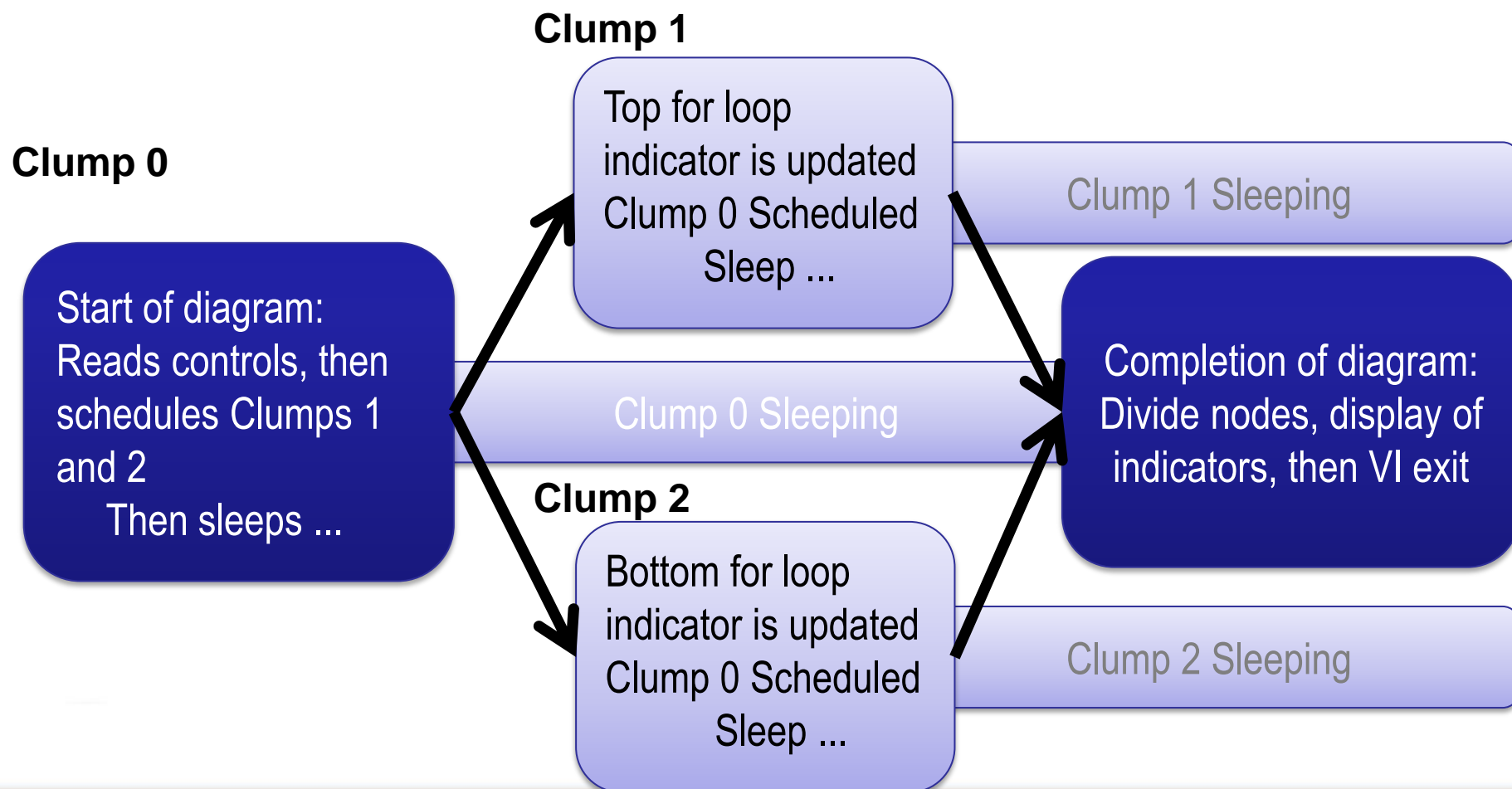
Clump 0

Clump 1

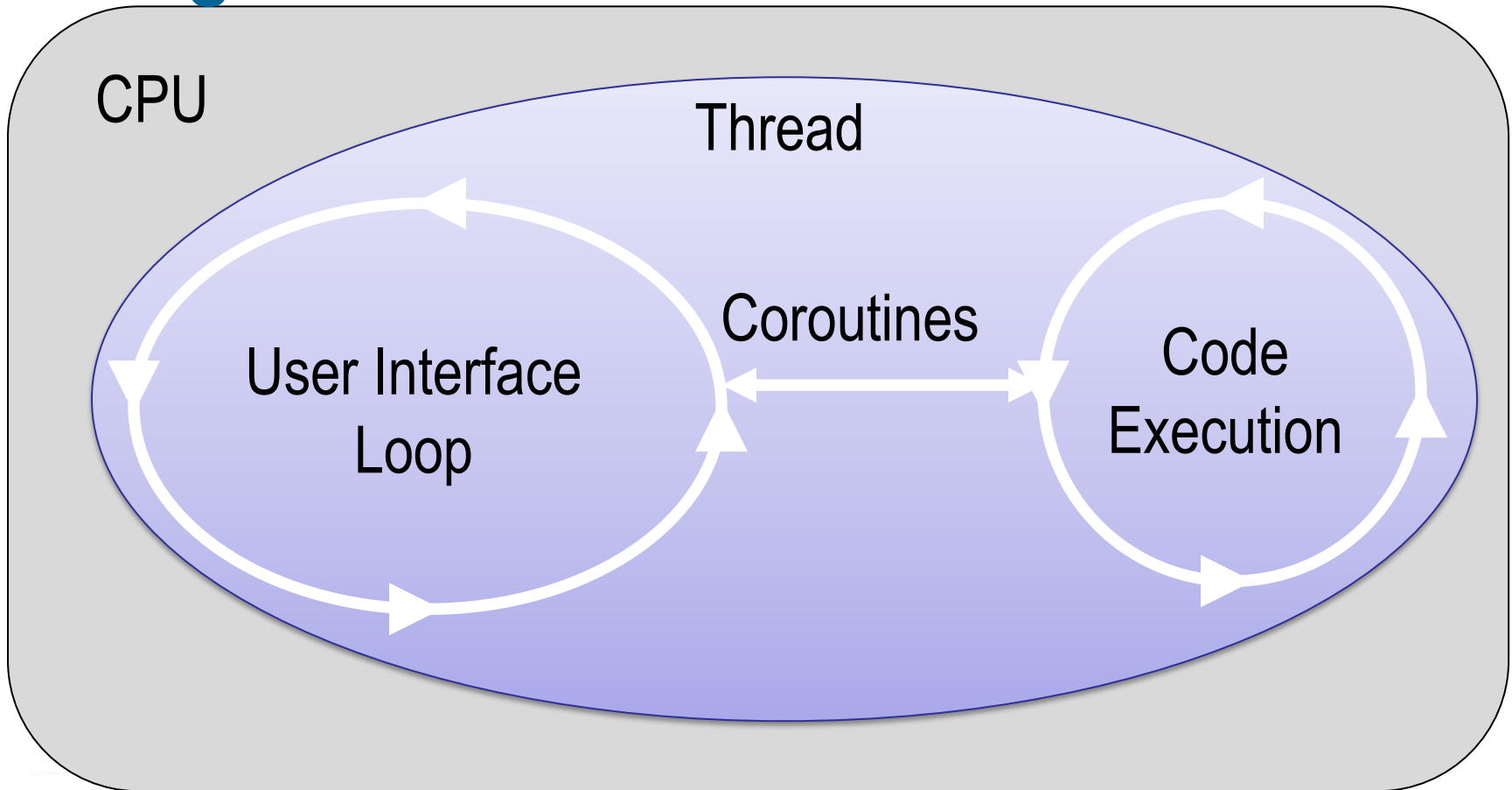
Clump 0



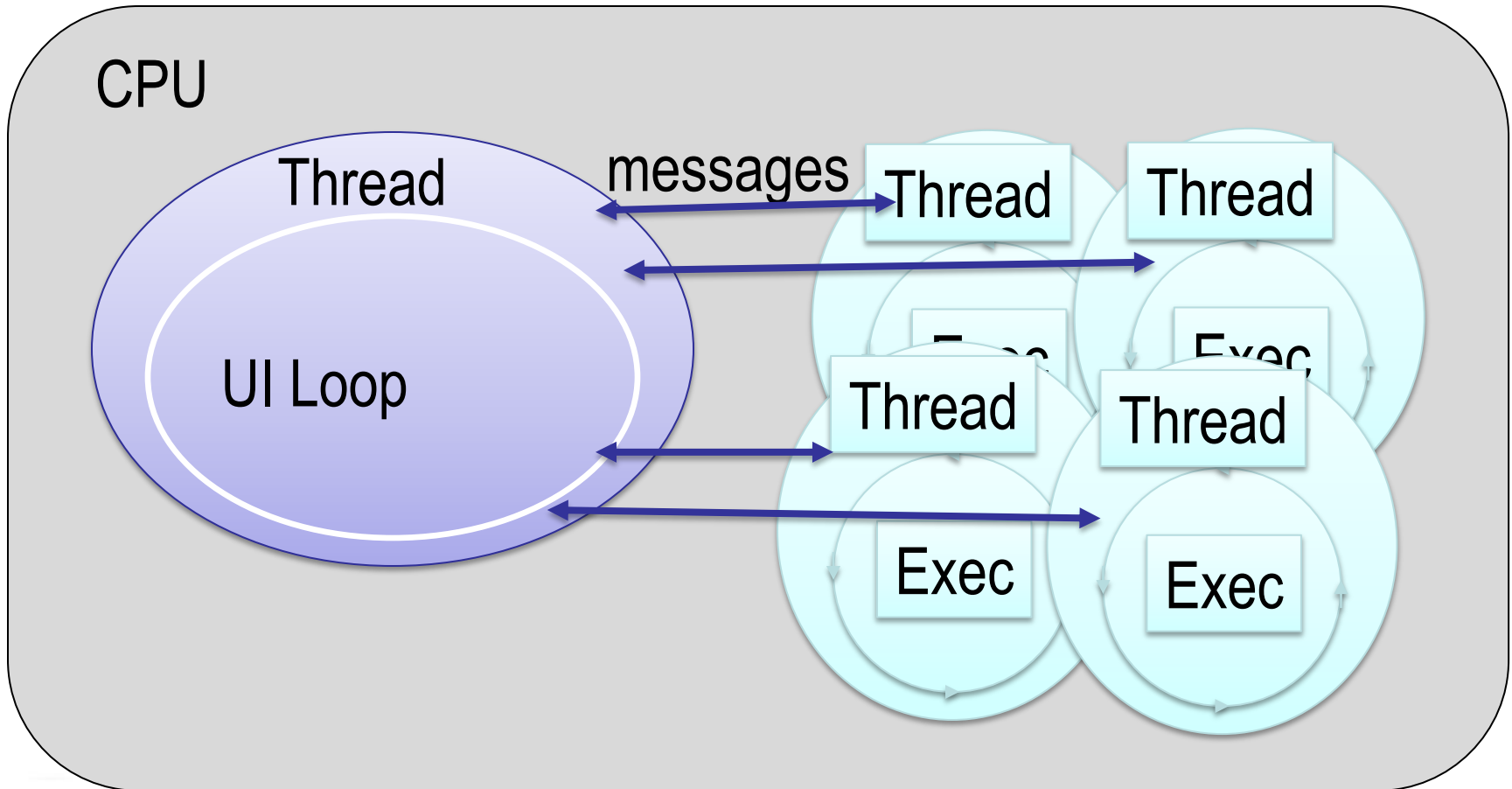
# VIs Are Compiled: “Clumps”



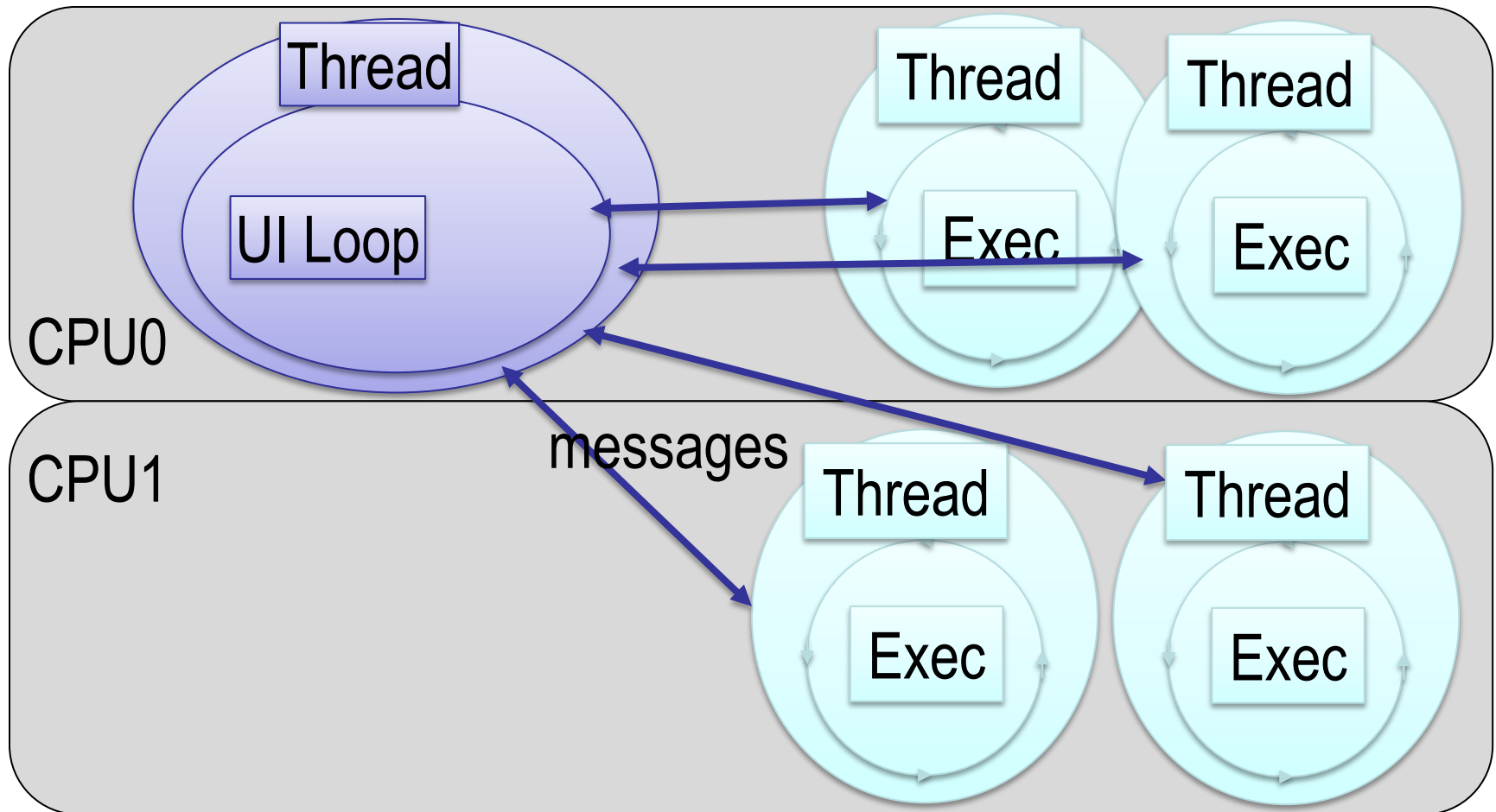
# Single-Threaded LabVIEW



# Multithreaded LabVIEW



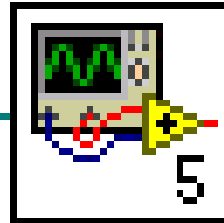
# LabVIEW on a Multicore Machine



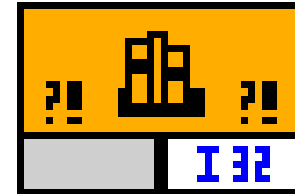


# Some Operations Require the UI Thread

Waveform Graph



Front Panel Control References

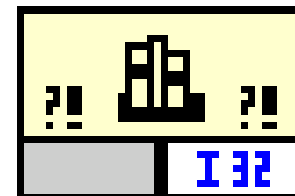


Call Library Nodes

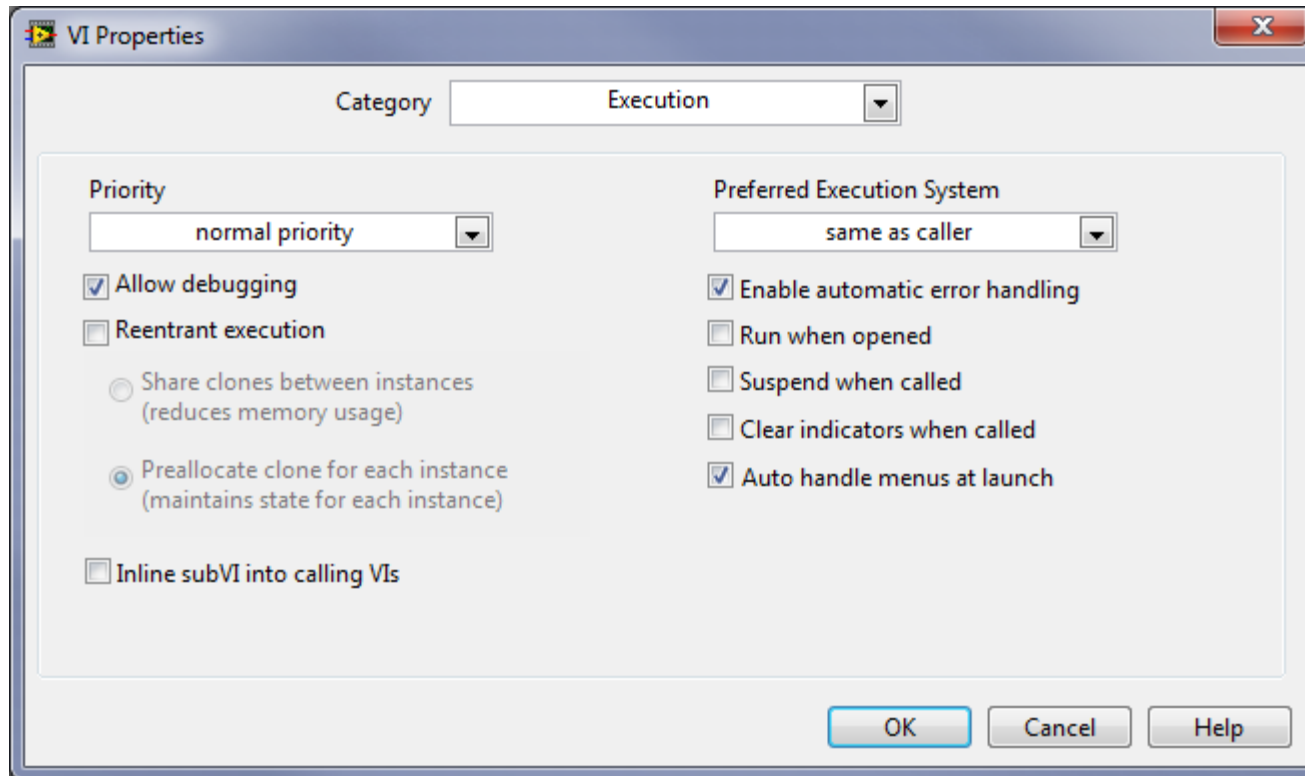
Array



Control/Indicator Property Nodes



# Execution Properties



# Reentrant VIs

- Reentrancy allows one subVI to be called simultaneously from different places
  - Requires extra memory for each instance
- Use reentrant VIs in two different cases
  - To allow a subVI to be called in parallel
  - To allow a subVI instance to maintain its own state

# LabVIEW 2010 Compiler

- Generates code that runs faster, ~30%
- Takes longer to run (~5x-7x)



# Demo – Effects of Execution Optimization





























# Next Steps

## In LabVIEW

- LabVIEW Help

## On the Web

- [ni.com/multicore](http://ni.com/multicore)
- [ni.com/devzone](http://ni.com/devzone)

- [-]  Managing Performance and Memory
  - [-]  Concepts
    -  Using the Profile Performance and Memory Window
    -  VI Execution Speed
    -  VI Memory Usage
    -  Memory Management for Large Data Sets
  - [-]  Multitasking, Multithreading, and Multiprocessing
    -  LabVIEW Threading Model
  - [-]  Multitasking
    -  Multitasking in LabVIEW
  - [-]  Multithreading
    -  Benefits of Multithreaded Applications
    -  Creating Multithreaded Applications
    -  Manually Assigning CPUs
    -  Multithreading Programming Examples
    -  Pipelining for Systems with Multiple CPUs
    -  Prioritizing Parallel Tasks
    -  Suggestions for Using Execution Systems and Priorities
  - [-]  Multiprocessing
    -  Multiprocessing and Hyperthreading in LabVIEW
- [-]  How-To
  -  Profiling VI Execution Time and Memory Usage
  -  Extending Virtual Memory Usage for 32-bit Windows
- [-]  How LabVIEW Stores Data in Memory
  - [-]  Concepts
    -  Flattened Data
    -  Type Descriptors
    -  Type Descriptors in LabVIEW 7.x and Earlier