LabVIEW™ Performance 2009 Exercises

Course Software Version 2009
April 2010 Edition
Part Number 325539A-01

Worldwide Technical Support and Product Information
ni.com

National Instruments Corporate Headquarters
11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 328 90 10, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Additional Information and Resources* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the Info Code feedback.

Contents

Lesson 5
Optimizing for Execution Speed

Appendix A
Additional Information and Resources

2

Designing Your Application

Exercise 2-1 Designing for Adaptability

Goal

To create a VI using the CPU Information Palette that executes differently depending on CPU architecture.

Scenario

You are writing an application that must execute on machines with different architectures. Use the CPU Information Palette to adjust a parallel For Loop that will take advantage of varying numbers of machine cores.

Implementation

1. Open a blank VI and save it as `CPU Info with Parallel For Loop.vi` in the `<Exercises>\LabVIEW Performance\Design` directory.

2. Create a block diagram containing a 2D array Specified by an Array Size control, as shown in Figure 2-1, using the following items:

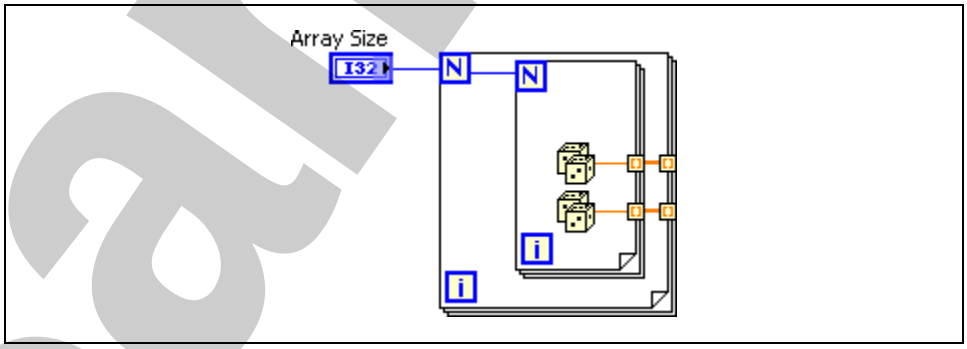


Figure 2-1. CPU Info with Parallel For Loop VI – First Set of For Loops

- ❑ Two For Loops, one inside the other
  - – Right-click the count terminal of the outer loop and select **Create»Control**.
  - – Rename the control as `Array Size`.

- ❑ Two Random Number numerics

3. To the right of the nested For Loops shown in Figure 2-1, add the following items to create the structure shown in Figure 2-2:

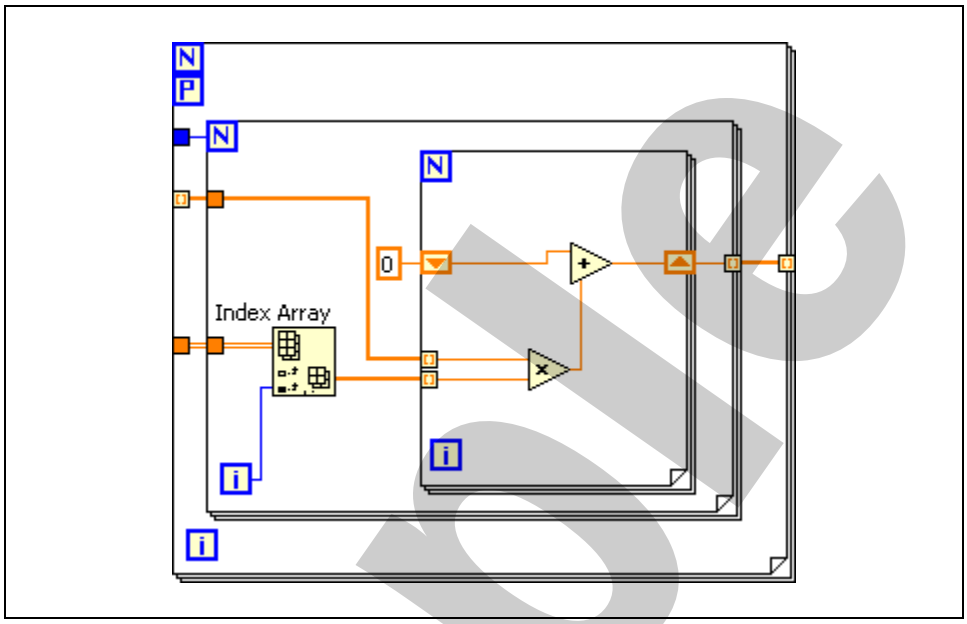
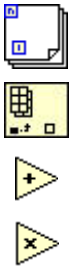


Figure 2-2. CPU Info with Parallel For Loop VI – Second Set of For Loops



- ❑ Three For Loops, nested inside one another
- ❑ Index Array
- ❑ Add function
- ❑ Multiply function
- ❑ Right-click the left shift register of the innermost For Loop and select **Create»Constant**. Set the constant to `0`.
- ❑ Right-click the outermost For Loop and select **Configure Iteration Parallelism**.
  - – Enable the **Enable loop iteration parallelism** checkbox.
  - – Click **OK**.
- ❑ Disable indexing on the innermost loop tunnels.
- ❑ Disable indexing on the outermost tunnel to Index Array.

4. Complete the block diagram, as shown in Figure 2-3, using the following items:

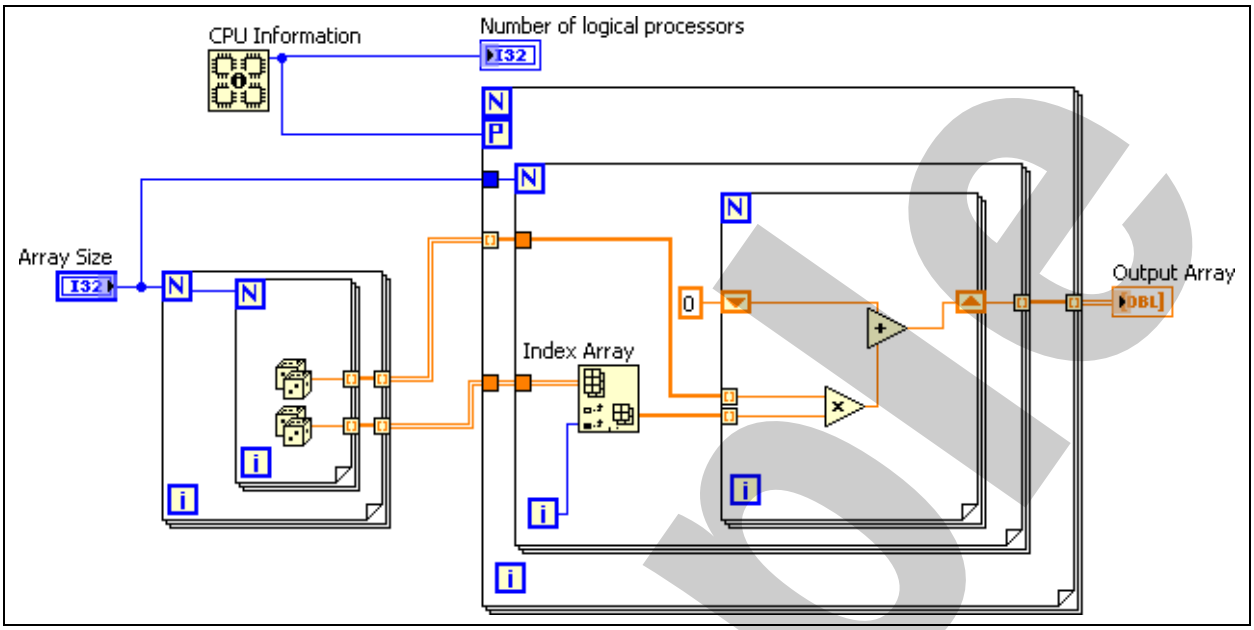


Figure 2-3. CPU Info with Parallel For Loop VI – Completed Block Diagram

- ❑ CPU Information function.
- ❑ Right-click the rightmost tunnel on the block diagram and select **Create»Indicator**. Rename the indicator as `Output Array`.
- ❑ Right-click the # of logical processors terminal and select **Create» Indicator**. Rename the indicator as `Number of Logical Processors`.

5. Save the VI.

Testing

1. Open Task Manager.
2. Click the **Performance Tab**.
3. On the front panel of the CPU Info with Parallel For Loop VI, set **Array Size** to `1000`.
4. Run the VI. Note the CPU Usage History. On a Multi-core machine, this code should utilize all cores.
5. Close Task Manager and the VI.

End of Exercise 2-1

Exercise 2-2 Static and Dynamic VI Calls

Goal

To understand how statically calling a VI or dynamically calling a VI affects the loading and unloading of VI Components.

Description

Identify the characteristics of calling a VI statically or dynamically.

Static VI Call

VI `aa.vi` is placed directly on the block diagram of `toplevel.vi`. The default settings are used, which means `aa.vi` is statically called from `toplevel.vi`.

1. When is the execution code for `aa.vi` loaded in memory?
   a. When `toplevel.vi` is loaded
   b. When `aa.vi` is first called
   c. Every time `aa.vi` is called
2. When is the execution code for `aa.vi` unloaded from memory?
   a. When `aa.vi` is done executing
   b. Sometime after `aa.vi` is done executing
   c. When `toplevel.vi` is done executing
3. Which of the following are true about statically calling `aa.vi`?
   a. Calling `aa.vi` will have a low amount of jitter because memory was allocated when the `toplevel.vi` was loaded, not when `aa.vi` is called.
   b. LabVIEW does not automatically reclaim the memory of `aa.vi` when `aa.vi` is done executing, which means that memory can be reused if `aa.vi` is called a second time.
   c. Loading `toplevel.vi` may be slower because `aa.vi` must also be loaded.

Dynamic VI Call

VI `bb.vi` is placed on the block diagram of `toplevel.vi`. You right-click the subVI and change the Call setup of `bb.vi` to Load and retain on first call.

1. When is the execution code for `bb.vi` loaded in memory?
   a. When `toplevel.vi` is loaded
   b. When `bb.vi` is first called
   c. Every time `bb.vi` is called
2. When is the execution code for `bb.vi` unloaded from memory?
   a. When `bb.vi` is done executing
   b. Sometime after `bb.vi` is done executing
   c. When `toplevel.vi` is done executing
3. Which of the following are true about dynamically calling `bb.vi`?
   a. Calling `bb.vi` will have a low amount of jitter because memory was allocated when the `toplevel.vi` was loaded, not when `bb.vi` is called.
   b. LabVIEW does not automatically reclaim the memory of `bb.vi` when `bb.vi` is done executing, which means that memory can be reused if `bb.vi` is called a second time.
   c. Loading `toplevel.vi` may be slower because `bb.vi` must also be loaded.

Solutions

Static VI Call

VI `aa.vi` is placed directly on the block diagram of `toplevel.vi`. The default settings are used, which means `aa.vi` is statically called from `toplevel.vi`.

1. When is the execution code for `bb.vi` loaded in memory?
   a. **When `toplevel.vi` is loaded**
   b. When `aa.vi` is first called
   c. Every time `aa.vi` is called
2. When is the execution code for `aa.vi` unloaded from memory?
   a. When `aa.vi` is done executing
   b. Sometime after `aa.vi` is done executing
   c. **When `toplevel.vi` is done executing**
3. Which of the following are true about statically calling `aa.vi`?
   a. **Calling `aa.vi` will have a low amount of jitter because memory was allocated when the `toplevel.vi` was loaded, not when `aa.vi` is called.**
   b. **LabVIEW does not automatically reclaim the memory of `aa.vi` when `aa.vi` is done executing, which means that memory can be reused if `aa.vi` is called a second time.**
   c. **Loading `toplevel.vi` may be slower because `aa.vi` must also be loaded.**

Dynamic VI Call

VI `bb.vi` is placed on the block diagram of `toplevel.vi`. You right-click the subVI and change the Call setup of `bb.vi` to Load and retain on first call.

1. When is the execution code for `aa.vi` loaded in memory?
   a. When `toplevel.vi` is loaded
   b. When `bb.vi` is first called
   c. Every time `bb.vi` is called
2. When is the execution code for `bb.vi` unloaded from memory?
   a. When `bb.vi` is done executing
   b. Sometime after `bb.vi` is done executing
   c. When `toplevel.vi` is done executing
3. Which of the following are true about dynamically calling `bb.vi`?
   a. Calling `bb.vi` will have a low amount of jitter because memory was allocated when the `toplevel.vi` was loaded, not when `bb.vi` is called.
   b. LabVIEW does not automatically reclaim the memory of `bb.vi` when `bb.vi` is done executing, which means that memory can be reused if `bb.vi` is called a second time.
   c. Loading `toplevel.vi` may be slower because `bb.vi` must also be loaded.

End of Exercise 2-2

Exercise 2-3 Reducing Redundant Code

Goal

To reduce duplicate code using subVIs.

Scenario

You have a VI with duplicate code. Reducing redundant code reduces the overall size of your VIs on disk and improves code maintainability and readability. You must replace the redundant code with a subVI and improve the subVI algorithm.

Implementation

1. Update the Duplicate Code VI to replace duplicate code with subVIs.

   ❑ Open `Duplicate Code.vi` from the `<Exercises>\LabVIEW Performance\Design` directory.

   ❑ On the block diagram, select the code outlined in Figure 2-4.

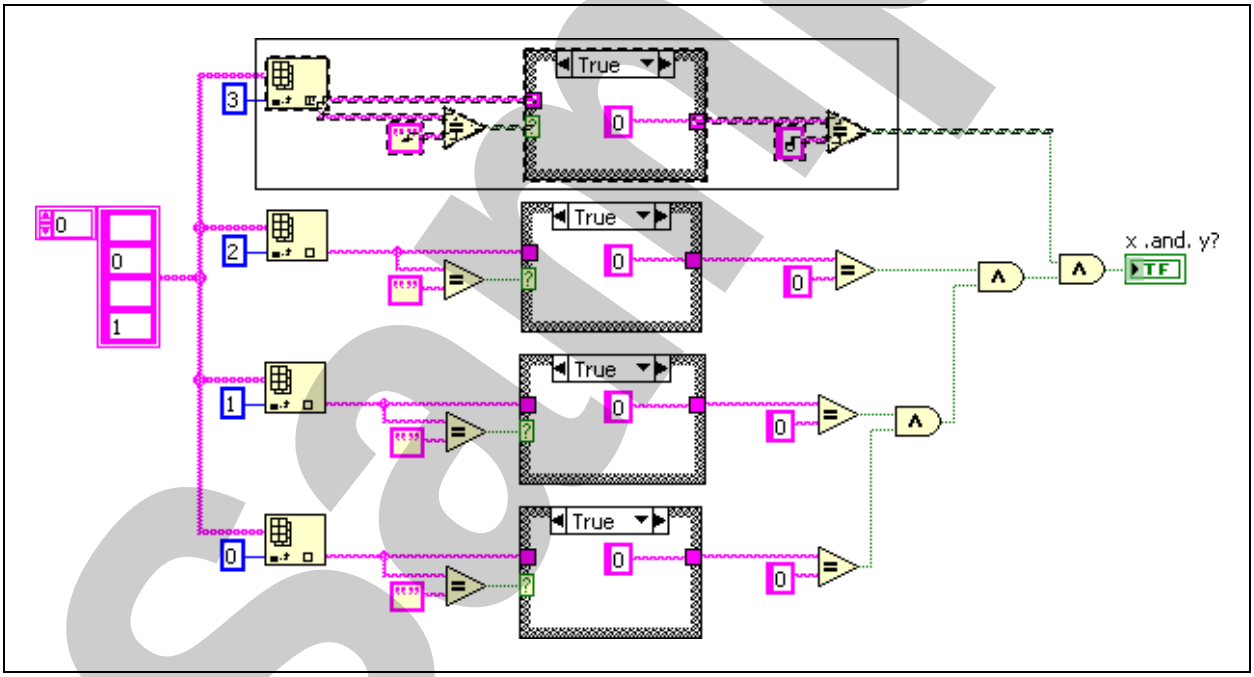


Figure 2-4. Duplicate Code VI with Code Section Selected

- ❑ Select **Edit»Create SubVI**.
- ❑ Double-click the subVI on the block diagram to open it.
- ❑ Select **File»Save As**. Save the subVI as `Duplicate Code SubVI.vi` in the `<Exercises>\LabVIEW Performance\Design` directory.

2. Replace the other three instances of duplicate code with `Duplicate Code SubVI.vi`.
   - ❑ Delete each duplicate code section.
   - ❑ Add three instances of `Duplicate Code SubVI.vi` to the block diagram.
   - ❑ Reconnect wires as shown in Figure 2-5.

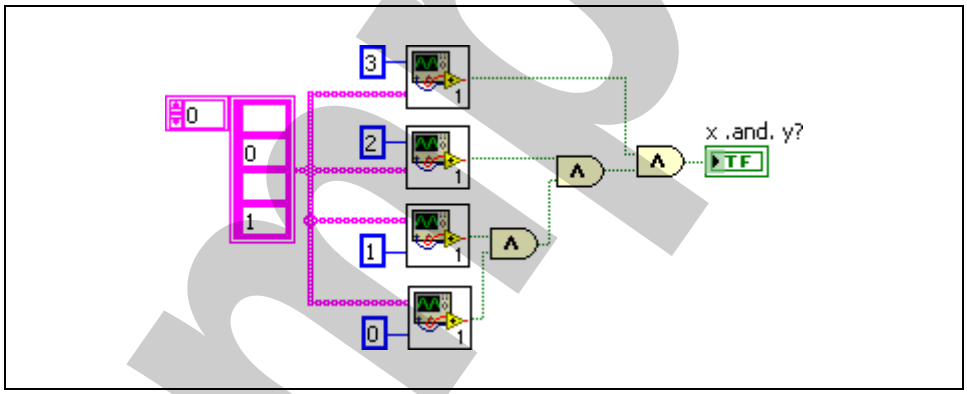


Figure 2-5. Duplicate Code VI with SubVIs

3. Replace the three And functions with a single Compound Arithmetic function.
   - ❑ Delete the existing And functions.
   - ❑ Place Compound Arithmetic function on the block diagram.
     - – Right-click the Compound Arithmetic function and select **Change Mode»AND**. Expand the node to accept four inputs.

❑ Wire the block diagram as shown in Figure 2-6.

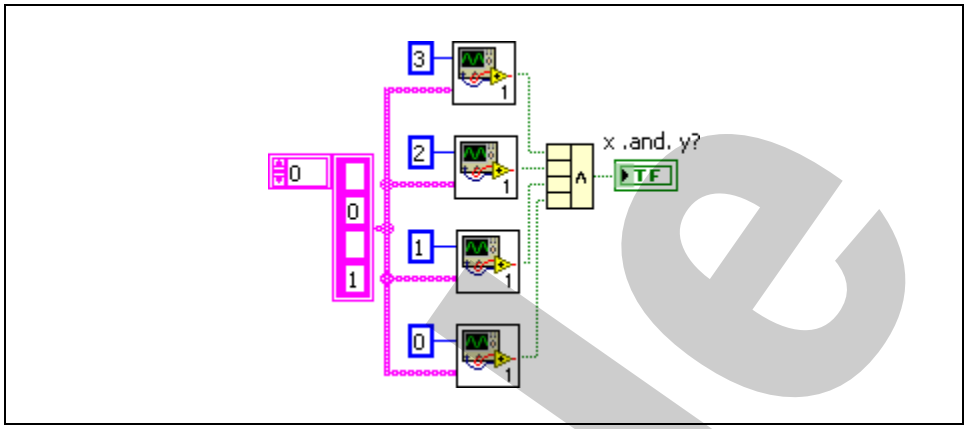


Figure 2-6. Duplicate Code VI with Compound Arithmetic Function

4. Improve the algorithm in the Duplicate Code SubVI VI.

❑ Open the block diagram of `Duplicate Code SubVI.vi`.

❑ Delete the Case structure.

❑ Replace the left Equal comparison with an **Empty String/Path?** function. Delete the empty constant.

❑ Place an Or function on the block diagram.

❑ Delete broken wires and arrange as shown in Figure 2-7.

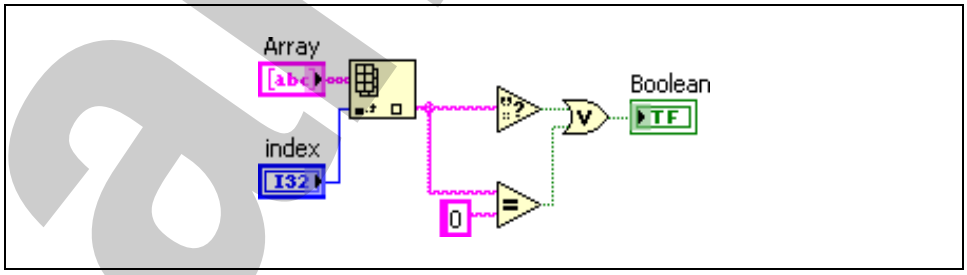


Figure 2-7. Improved Duplicate Code SubVI VI

Note Use the **Clean Up Block Diagram** tool to improve block diagram appearance.

5. Save and close the VI and subVI.

End of Exercise 2-3

Exercise 2-4 Improving I/O Performance

Goal

To profile a VI and determine which VI has the most execution time and which uses the most memory.

Scenario

Given a VI which logs data to file, compare the efficiency of File IO operations. Modify the VI to improve the overall performance.

Implementation

1. Open `FileIO.vi` in the `<Exercises>\LabVIEW Performance\Design` directory.
2. Examine the time to write waveform data to an ASCII file.
   - ❑ Run the FileIO VI.
   - ❑ Note the Time to Write to file (ms): _____
   - ❑ Note also that this file write operation loop iterates 10,000 times.
   - ❑ Examine the block diagram. Notice that this elapsed time applies only to the time to write the waveform data to a file. This file can be viewed as a text file or as a tab delimited text file in Excel to verify data.
3. Update the VI to write to file more efficiently.
   - ❑ Right-click each of the **Write to Text File VIs** and select **Replace» File I/O Palette**. Select **Write to Binary File** to replace the existing VIs.
   - ❑ Rename the `data.txt` string constant to `databinary`. This is the name of the file to which you are writing.
4. Examine the time to write waveform data to an binary file.
   - ❑ Run the FileIO VI.
   - ❑ Note the Time to Write to file (ms): _____

5. Compare the two values. You can see that writing to a binary file is much more efficient than writing to an ASCII file.

   Also note that FileIO_subvi must convert and format data to make the text file more readable. If you are writing to a binary file, overall efficiency can be improved by not converting data to strings. The limitation is that if you are writing to a binary file, you must also build the VI to read the data since it is necessary to know what data types were written to the file in order to decode them.

6. Close and save the FileIO VI.

End of Exercise 2-4

Notes

Notes