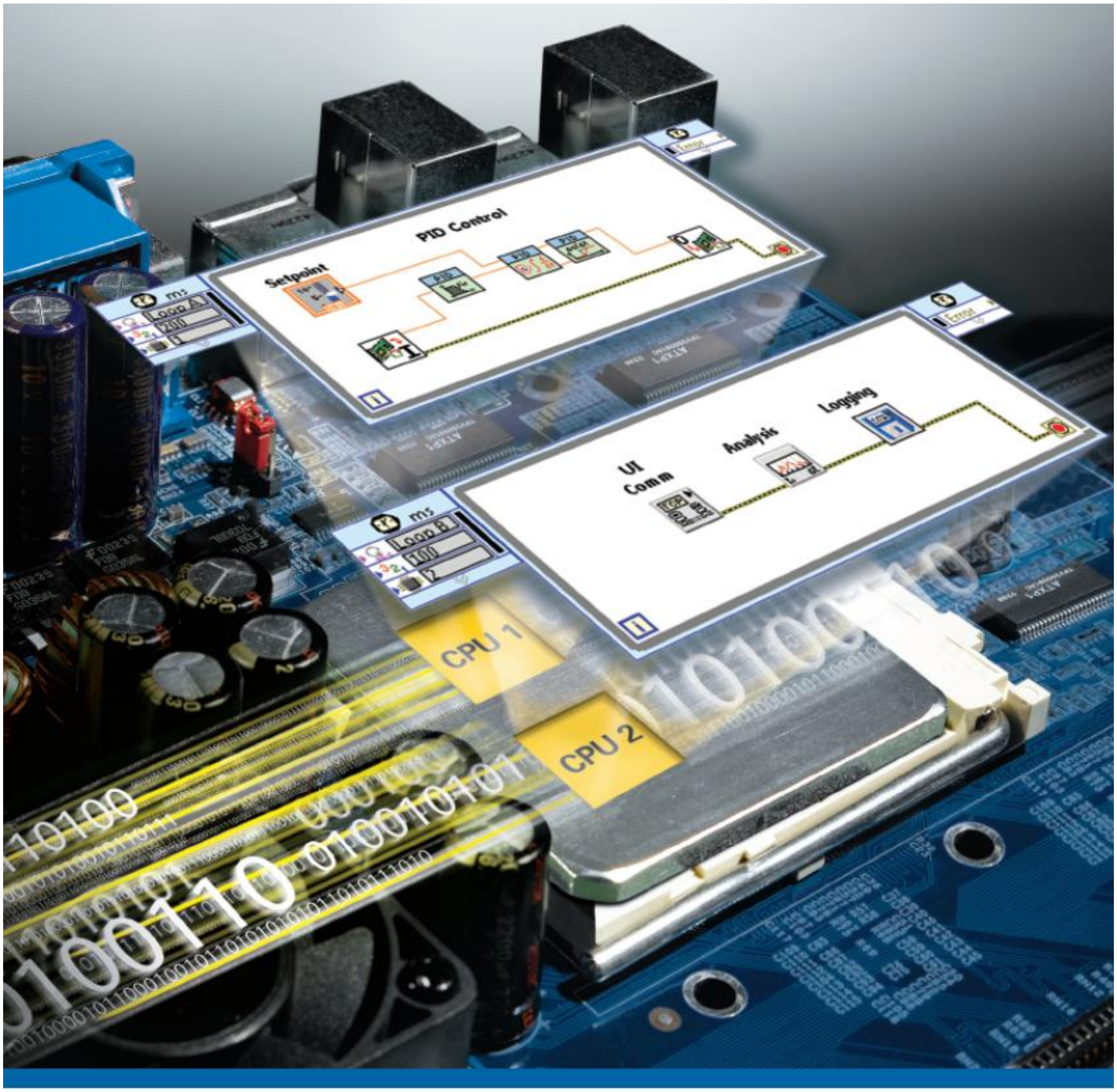
Multicore Programming with LabVIEW
Technical Resource Guide



www.ni.com/multicore

NATIONAL INSTRUMENTS™

INTRODUCTORY TOPICS

PROGRAMMING STRATEGIES

REAL-TIME CONSIDERATIONS

Understanding Parallel Hardware: Multiprocessors, Hyperthreading, Dual-Core, Multicore and FPGAs

Overview

Parallel hardware is becoming a ubiquitous component in computer processing technology. Learn the difference between common parallel hardware architectures found in the marketplace today, including Multiprocessor, Hyperthreading, Dual-Core, Multicore and FPGAs.

Multiprocessors

Multiprocessor systems contain multiple CPUs that are not on the same chip. Multiprocessor systems were made common in the 1990s for the purpose of IT servers. At that time they were typically processor boards that would slide into a rack-mount server. Today, multiprocessors are commonly found on the same physical board and connected through a high-speed communication interface.

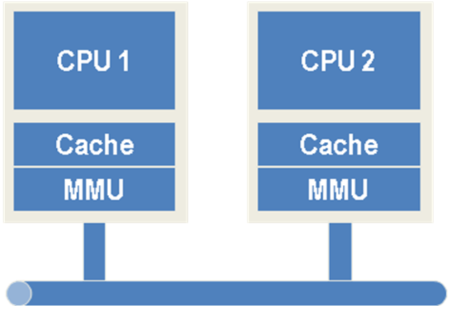


Figure 1. The multiprocessor system has a divided cache and MMU with long-interconnects

Multiprocessor systems are less complex than multicore systems, because they are essential single chip CPUs connected together. The disadvantage with multiprocessor systems is that they are expensive because they require multiple chips which is more expensive than a single chip solution.

Hyperthreading

Hyperthreading is a technology that was introduced by Intel, with the primary purpose of improving support for multi-threaded code. Pentium 4 processors are an example of CPUs that implement hyperthreading.

Dual-Core and Multicore Processors

Dual-core processors are two CPUs on a single chip. Multicore processors are a family of processors that contain any number of multiple CPUs on a single chip, such as 2, 4, and 8. The challenge with multicore processors is in the area of software development. Performance speed-up is directly related to the how parallel the source code of an application was written through multi-threading.

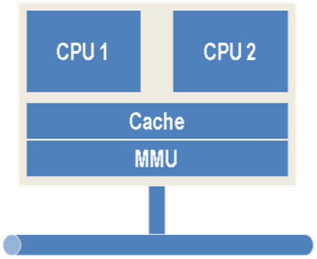


Figure 2. The multicore processors share the cache and MMU with short interconnects

FPGAs

FPGAs (Field Programmable Gate Arrays) are a type of silicon composed of logic gates. They are considered a massively parallel hardware device and are well-suited for high-performance computing and number crunching, such as digital signal processing (DSP) applications. FPGAs run at a lower clock rate than microprocessors and consumer more power.

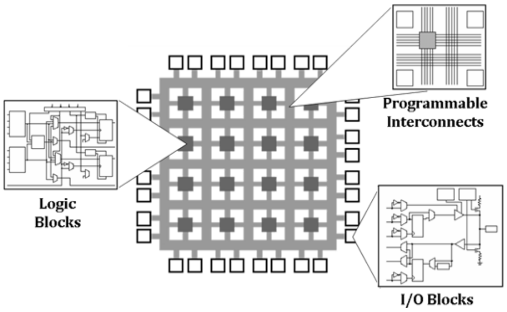


Figure 3. FPGA

A FPGA is programmable chip composed of three basic components. First, the logic blocks are where bits are crunched and processed to produce programmatic results. Second, the logic blocks are connected together through programmable interconnects routing signals from one logic block to the next block. The interconnects on the FPGA serve as a micro switch matrix. Third, the I/O blocks connect to the pins on the chip providing two-way communication to surrounding circuitry.

Since FPGAs execute in a parallel fashion, it allows the user to create any number of task-specific cores that all run like simultaneous parallel circuits inside one FPGA chip. The parallel nature of the logical gates on the FPGA allow for very high throughput of data, more so than their microprocessor counterparts.

How LabVIEW Programs Parallel Hardware

The dataflow nature of LabVIEW allows parallel code to easily map to parallel hardware. Therefore, it is an ideal development language for targeting multiprocessor, hypertheaded, and multicore processor systems.In the case of programming FPGAs, LabVIEW generates VHDL code which is automatically compiled to a bitstream that can target Xilinx FPGAs.

Differences between Multithreading and Multitasking

Multitasking

In computing, multitasking is a method by which multiple tasks, also known as processes, share common processing resources such as a CPU. With a multitasking OS, such as Windows XP, you can simultaneously run multiple applications. Multitasking refers to the ability of the OS to quickly switch between each computing task to give the impression the different applications are executing multiple actions simultaneously.

As CPU clock speeds have increased steadily over time, not only do applications run faster, but OSs can switch between applications more quickly. This provides better overall performance. Many actions can happen at once on a computer, and individual applications can run faster.

Single Core

In the case of a computer with a single CPU core, only one task runs at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves this problem by scheduling which task may run at any given time and when another waiting task gets a turn.

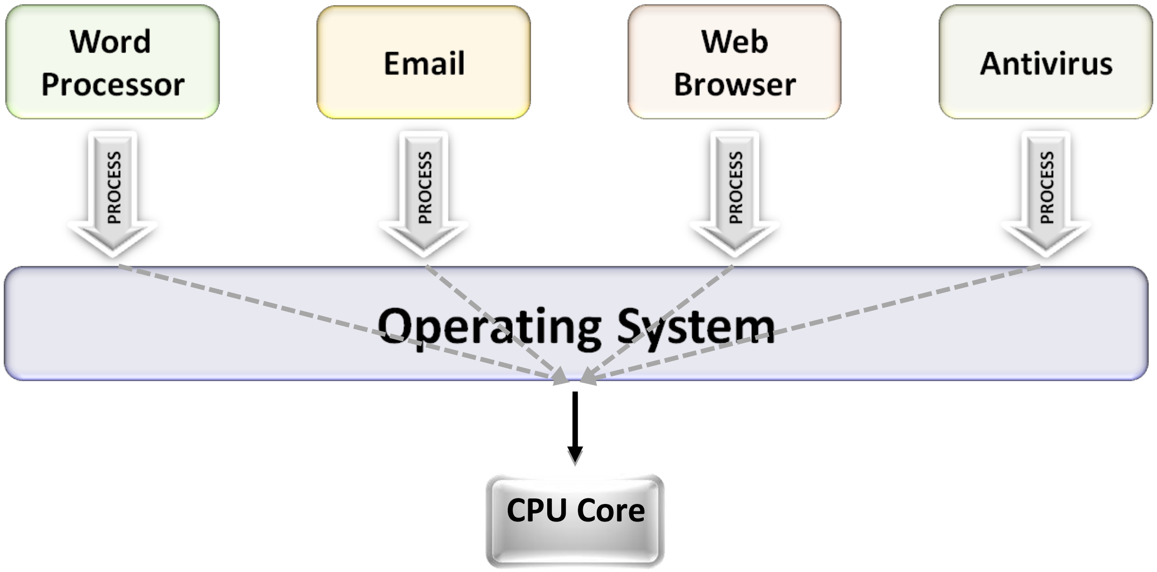


Figure 1. Single-core systems enable multitasking OSs.

Multicore

When running on a multicore system, multitasking OSs can truly execute multiple tasks concurrently. The multiple computing engines work independently on different tasks.

For example, on a dual-core system, four applications - such as word processing, e-mail, Web browsing, and antivirus software - can each access a separate processor core at the same time. You can multitask by checking e-mail and typing a letter simultaneously, thus improving overall performance for applications.

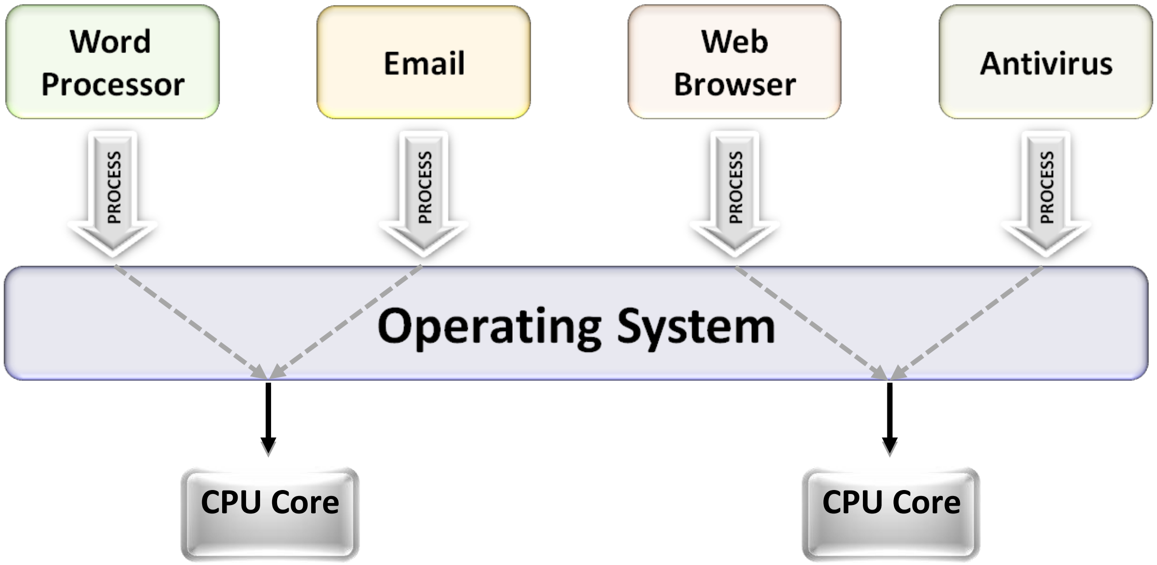


Figure 2. Dual-core systems enable multitasking OSs, such as Windows XP, to truly execute two tasks simultaneously.

The OS executes multiple applications more efficiently by splitting the different applications, or processes, between the separate CPU cores. The computer can spread the work - each core is managing and switching through half as many applications as before - and deliver better overall throughput and performance. In effect, the applications are running in parallel.

Multithreading

Multithreading extends the idea of multitasking into applications, so you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Engineering and scientific applications are typically on dedicated systems (i.e. little multitasking). In a multithreaded National Instruments LabVIEW program, an example application might be divided into four threads - a user interface thread, a data acquisition thread, network communication, and a logging thread. You can prioritize each of these so that they operate independently. Thus, in multithreaded applications, multiple tasks can progress in parallel with other applications that are running on the system.

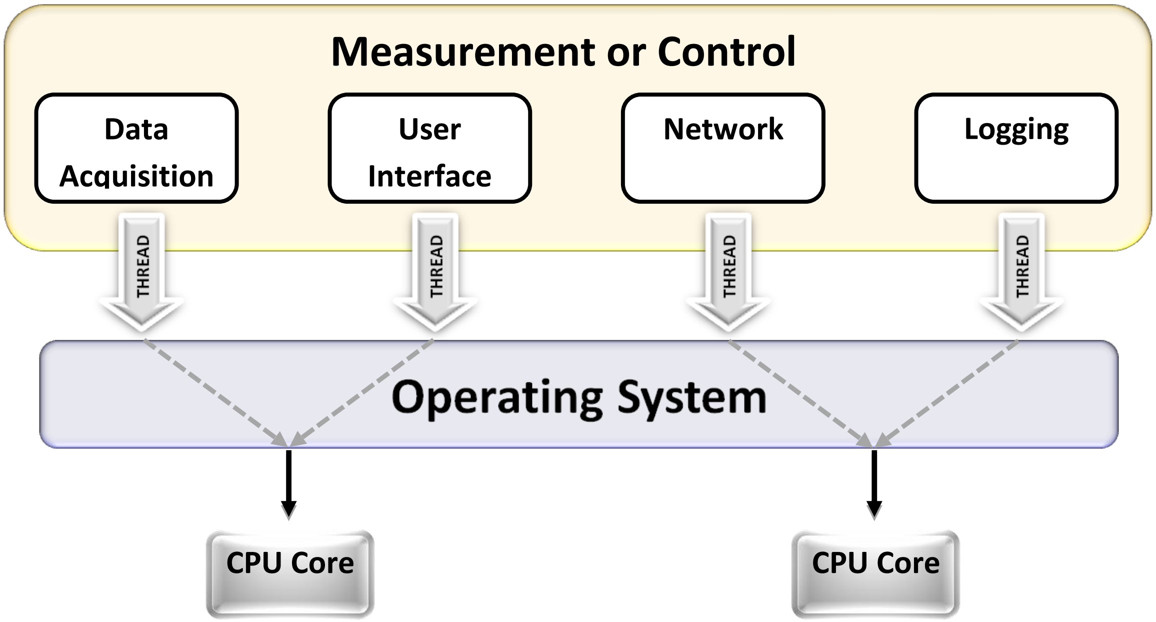


Figure 3. Dual-core system enables multithreading

Applications that take advantage of multithreading have numerous benefits, including the following:

- More efficient CPU use
- Better system reliability
- Improved performance on multiprocessor computers

In many applications, you make synchronous calls to resources, such as instruments. These instrument calls often take a long time to complete. In a single-threaded application, a synchronous call effectively blocks, or prevents, any other task within the application from executing until the operation completes. Multithreading prevents this blocking.

While the synchronous call runs on one thread, other parts of the program that do not depend on this call run on different threads. Execution of the application progresses instead of stalling until the synchronous call completes. In this way, a multithreaded application maximizes the efficiency of the CPU because it does not idle if any thread of the application is ready to run.

Multithreading with LabVIEW

NI LabVIEW automatically divides each application into multiple execution threads. The complex tasks of thread management are transparently built into the LabVIEW execution system.

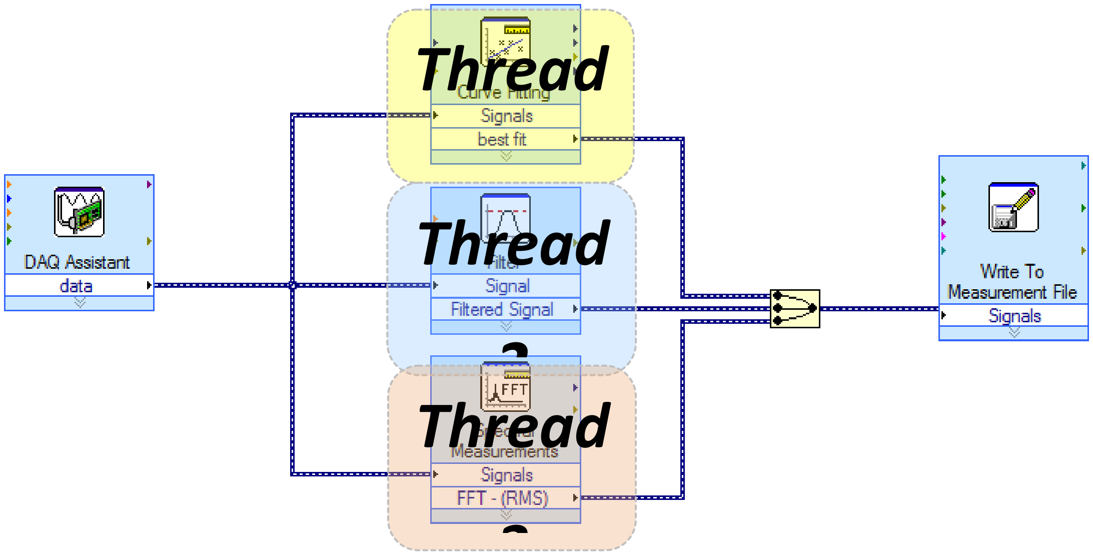


Figure 4. LabVIEW enables the user to execute multiple execution thread

Multitasking in LabVIEW

LabVIEW uses preemptive multithreading on OSs that offer this feature. LabVIEW also uses cooperative multithreading. OSs and processors with preemptive multithreading employ a limited number of threads, so in certain cases, these systems return to using cooperative multithreading.

The execution system preemptively multitasks VIs using threads; however, a limited number of threads are available. For highly parallel applications, the execution system uses cooperative multitasking when available threads are busy. Also, the OS handles preemptive multitasking between the application and other tasks.

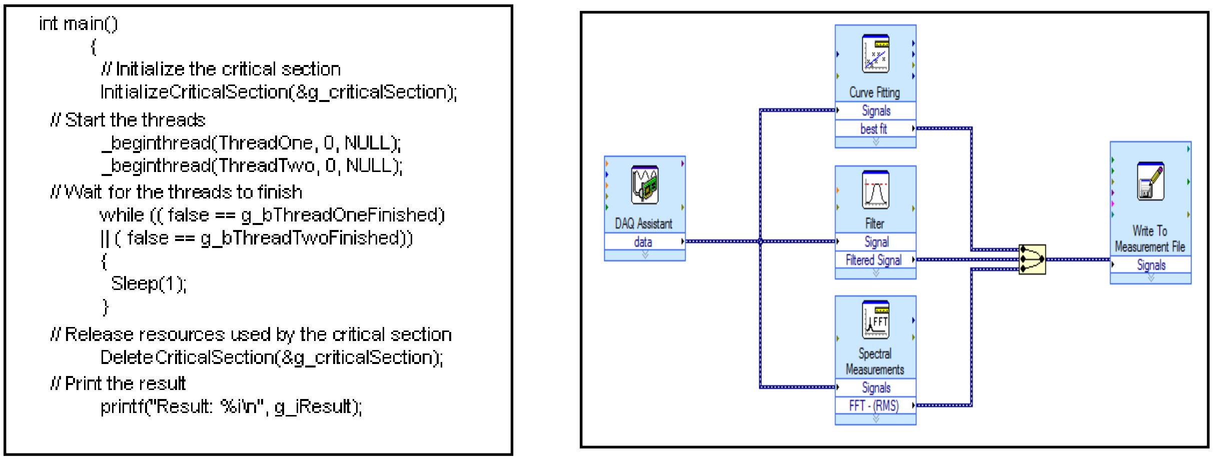
Overcoming Multicore Programming Challenges: Thread Synchronization and Visual Code Debugging

This paper outlines several challenges facing multicore programmers and highlights features of the National Instruments LabVIEW 8.5 graphical programming environment. Specifically, this paper discusses designing parallel application architectures, dealing with thread synchronization, and debugging multicore programs.

Designing Parallel Code

The first major challenge in programming a parallel application is identifying which sections of a given program can actually run in parallel with each other, and then implementing those sections in code. A piece of code that is able to run in parallel with another piece is a thread; therefore, an entire parallel application is multithreaded.

Traditionally, text-based programmers have had to explicitly define these threads in their applications using APIs such as OpenMP or POSIX. Because text-based programming is inherently serial in nature, attempting to visualize parallelism in a multithreaded piece of code is difficult. On the other hand, by harnessing the graphical nature of NI LabVIEW, coders can easily visualize and program parallel applications. In addition, LabVIEW automatically generates threads for parallel sections of code, so engineers and scientists with little or no programming background can spend more time problem solving and less time worrying about low-level implementation of their applications.



```
  int main()
          {
           // Initialize the critical section
           InitializeCriticalSection(&g_criticalSection);
   // Start the threads
           _beginthread(ThreadOne, 0, NULL);
           _beginthread(ThreadTwo, 0, NULL);
   // Wait for the threads to finish
           while (( false == g_bThreadOneFinished)
           || ( false == g_bThreadTwoFinished))
           {
             Sleep(1);
           }
   // Release resources used by the critical section
           DeleteCriticalSection(&g_criticalSection);
   // Print the result
           printf("Result: %i\n", g_iResult);
```

Figure 1 - Comparison of Multithreading in LabVIEW and Text-Based Languages

Thread Synchronization

A second challenge of multicore programming is thread synchronization. With multiple threads running in a given application, you must ensure that all these threads work well together. For example, if two or more threads attempt to access a memory location at the same time, data corruption can occur. Clearly, identifying all possible conflicting pieces of code in an application is a daunting task.

By graphically creating a block diagram in LabVIEW, however, you can quickly take a specific task from idea to implementation without considering thread synchronization. Figure 2 shows an application in which both parallel sections of graphical code access a hard disk when writing a file. LabVIEW automatically handles the thread synchronization.

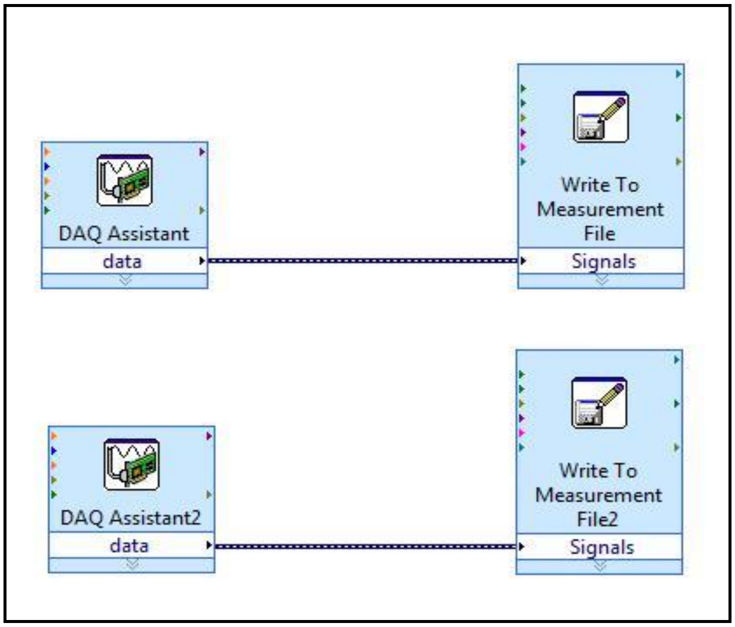


Figure 2 - Simple Application Demonstrating Automatic Thread Synchronization in LabVIEW

Debugging

Most programs do not function perfectly the first time they are executed. This is true for single-core as well as multicore applications. To logically determine where any functional errors occur in a given piece of code, you must rely on debugging tools in the development environment to produce the correct behavior.

Debugging poses a unique challenge in multicore applications. Not only must you trace execution of two pieces of code at once, you must also determine which piece of code is running on which processor. Additionally, if you program multithreaded applications frequently, you must deal with thread swapping and starvation issues, which need to be identified during the debugging process.

LabVIEW contains several features that greatly simplify debugging multicore applications. Specifically, you can use the execution highlighting feature to quickly and easily visualize the parallel execution of a program (LabVIEW is inherently based on data flow). For instance, observe the simple application in Figure 3. When execution highlighting is turned on, you can easily visualize parallel sections of code executing.

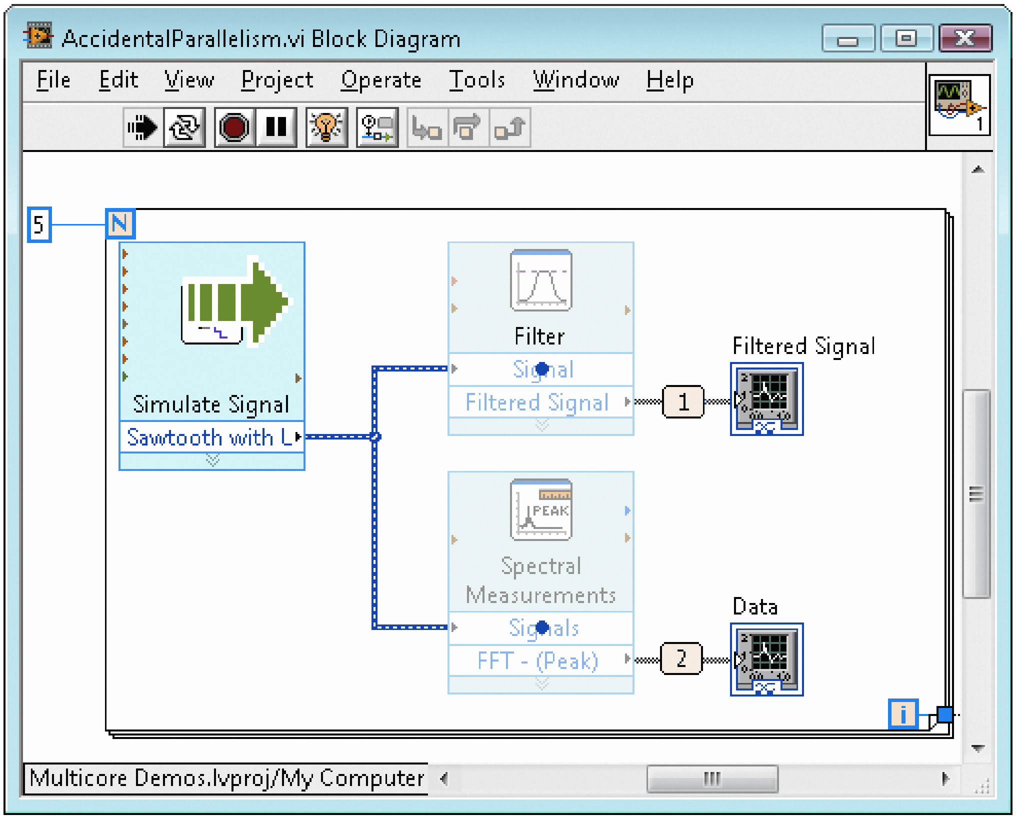
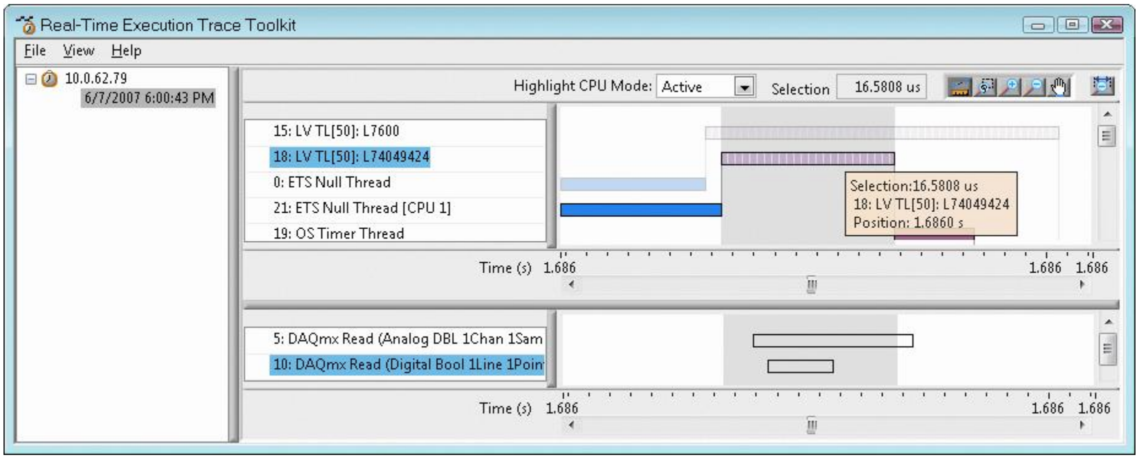


Figure 3 - Graphical Execution Highlighting in the LabVIEW Development Environment

In addition, the LabVIEW Real-Time Module provides both deterministic execution on multicore machines and extensive debugging information. New to LabVIEW 8.5, the Real-Time Execution Trace Toolkit allows you to visualize which processor a given thread is running on and to quickly identify issues such as thread starvation and swapping.


Highlight CPU Mode: Active
Selection 16.5808 us

Selection:16.5808 us
18: LV TL[50]: L74049424
Position: 1.6860 s

Real-Time Execution Trace Toolkit
File View Help
10.0.62.79
6/7/2007 6:00:43 PM
15: LV TL[50]: L7600
18: LV TL[50]: L74049424
0: ETS Null Thread
21: ETS Null Thread [CPU 1]
19: OS Timer Thread
Time (s) 1.686 1.686 1.686
5: DAQmx Read (Analog DBL 1Chan 1Sam
10: DAQmx Read (Digital Bool 1Line 1Poin
Time (s) 1.686 1.686 1.686

Figure 4 - LabVIEW 8.5 Execution Trace Toolkit

To conclude, multicore-savvy programmers must take into account the unique challenges of multithreading. A few of these challenges include, but are not limited to, parallel application architecture, thread synchronization, and debugging. As the number of cores per processor increases, it is increasingly important to employ correct parallel programming techniques in multithreaded applications.

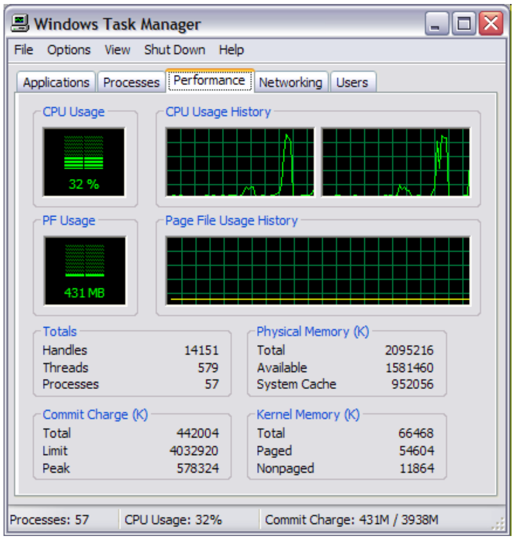
Will My LabVIEW Programs Run Faster When I Upgrade to a Multicore Computer?

Several factors determine whether the performance of a National Instruments LabVIEW program improves on a multicore system (for example, dual—core and quad—core machines). Such factors include specifics of the new hardware, structure of the LabVIEW application, system software, and so on. Tests of common LabVIEW program show, on average, a 25 to 35 percent improvement in execution time because of the natural parallelism in most LabVIEW code. However, the nature of an individual program can significantly affect this estimate. Optimizing a LabVIEW program for a multicore computing is the best way to ensure the maximum speed-up when running code on a multicore computing system. This whitepaper addresses the major factors affecting LabVIEW program performance on multicore systems.

Performance Metrics

When evaluating LabVIEW program performance, memory usage and execution time are the main metrics considered. Execution time is the amount of time required to process a group of instructions, usually measured in seconds. Memory usage is the amount of memory space required to process a group of instructions, usually measured in bytes. These measurements are good indicators of individual program performance, with execution time being key. Another performance improvement seen on multicore systems is responsiveness. Responsiveness-how quickly a program or system responds to inputs-does not take into account the amount of time required to execute a desired action. Multicore systems exhibit improved responsiveness due to multitasking with several cores available. While this is a performance improvement, it is not necessarily an indication of program execution time improvement.

For a view of performance and CPU load on Windows, the Task Manager can be used which splits the view by core:



32 %

431 MB

Totals	
Handles	14151
Threads	579
Processes	57

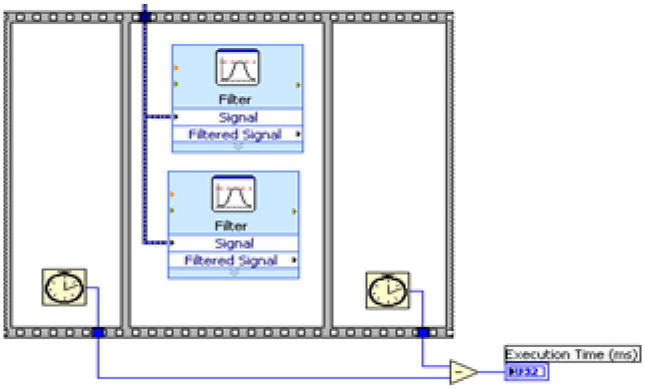
Physical Memory (K)	
Total	2095216
Available	1581460
System Cache	952056

Commit Charge (K)	
Total	442004
Limit	4032920
Peak	578324

Kernel Memory (K)	
Total	66468
Paged	54604
Nonpaged	11864

Processes: 57 CPU Usage: 32% Commit Charge: 431M / 3938M

The Windows Task Manager does not provide any kind of precise benchmark, so the best method to measure performance in your LabVIEW application would be to do basic benchmarking around sections of code.



In addition, you can compare the speed-up from 1-core to 2-core by going to Windows Task Manager and manually setting affinity to the LabVIEW process, and re-running your LabVIEW benchmark.

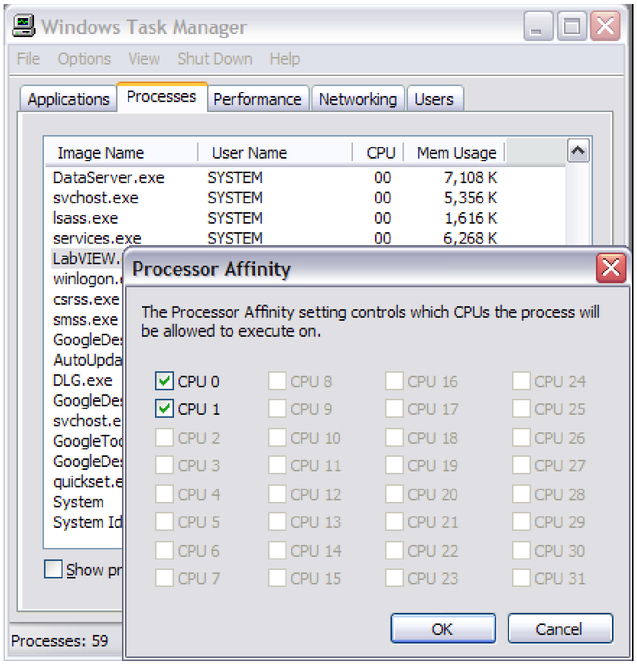


Image Name	User Name	CPU	Mem Usage
DataServer.exe	SYSTEM	00	7,108 K
svchost.exe	SYSTEM	00	5,356 K
lsass.exe	SYSTEM	00	1,616 K
services.exe	SYSTEM	00	6,268 K
LabVIEW.			
winlogon.			
csrss.exe			
smss.exe			
GoogleDe			
AutoUpda			
DLG.exe			
GoogleDe			
svchost.e			
GoogleToo			
GoogleDe			
quickset.e			
System			
System Id			

□ Show pr

Processes: 59

Processor Affinity

The Processor Affinity setting controls which CPUs the process will be allowed to execute on.

☑ CPU 0 □ CPU 8 □ CPU 16 □ CPU 24
☑ CPU 1 □ CPU 9 □ CPU 17 □ CPU 25
□ CPU 2 □ CPU 10 □ CPU 18 □ CPU 26
□ CPU 3 □ CPU 11 □ CPU 19 □ CPU 27
□ CPU 4 □ CPU 12 □ CPU 20 □ CPU 28
□ CPU 5 □ CPU 13 □ CPU 21 □ CPU 29
□ CPU 6 □ CPU 14 □ CPU 22 □ CPU 30
□ CPU 7 □ CPU 15 □ CPU 23 □ CPU 31

It’s important to note though, even if you set LabVIEW to one CPU, Windows can still schedule other tasks on that same CPU. So there would always be margins for error due to variations in how Windows handles background system tasks.

Considering Clock Speed and Number of Processors

LabVIEW automatically scales programs to take advantage of multiple processors on high end computing systems by determining the number of cores available and allowing a greater number of threads to be created. For example, LabVIEW creates eight threads for program execution when running on an octal—core computer.

More processors, same clock speed

Ideally, program execution speed increases by a factor equal to the number of cores on the multicore computing system (for example, four times speed up on a quad—core system); however, communication overhead between threads and cores prevents ideal execution time improvement. If the LabVIEW program in question is completely sequential and running on a single processor, it needs less shared processor time with other software, leading to program execution time improvement. If the program in question is completely parallel and consists of tasks of equal size having no data dependency, near ideal execution time improvement can be achieved.

More Processors, Slower Clock Speed

Replacing a single—core computing system with a multicore system that uses processors with slower clock speeds creates an ambiguous scenario for determining changes in program execution time. If the LabVIEW program in question is completely sequential and has exclusive access to a single core on the multicore system, relative clock speeds of the systems and task scheduling likely determine the effect on execution time. Execution time of a LabVIEW program that is completely parallel, consists of tasks of equal size having no data dependency, and that is given access to all available cores on the multicore machine is also dependent on relative clock speeds and task scheduling.

Communications Overhead

Memory Schemes

Memory organization in multicore computing systems affects communication overhead and LabVIEW program execution speed. Common memory architectures are shared memory, distributed memory and hybrid shared-distributed memory. Shared memory systems use one large global memory space, accessible by all processors, providing fast communication. However, as more processors are connected to the same memory, a communication bottleneck between the processors and memory occurs. Distributed memory systems use local memory space for each processor and communicate between processors via a communication network, causing slower interprocessor communication than shared memory systems.   In addition, shared-distributed memory architecture is used on some systems to exploit the benefits of both architectures. Memory schemes have a significant effect on communication overhead and as a result an effect on program execution speed in any language (LabVIEW, C, Visual Basic, and so on).

Interprocessor Communication

Physical distances between processors and the quality of interprocessor connections affect LabVIEW program execution speed through communication overhead. Multiple processors on separate ICs exhibit higher interprocessor communication latency than processors on a single IC. This results in larger communication overhead penalties, which slow LabVIEW program execution time. For example, in Figure 1, the dual—processor system (two sockets) on the left has higher latency than the single—chip, dual—core processor on the right.

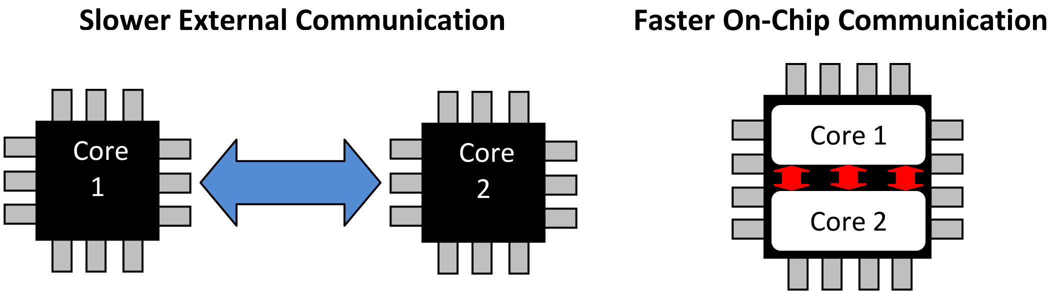


Figure 1. Dual Processor vs. Dual-Core Processor

Program and Software Issues

Code Organization

LabVIEW program execution time on a multicore computer depends just as much on the program as on the computer running it. The program must be written in such a way that it can benefit from the unique environment presented on multicore systems. The degree of program parallelism has a large effect on program execution time, as does granularity (the ratio of computation to communication) and load balancing. A large amount of existing G code is written for sequential execution; however, this type of code likely has some inherent parallelism due to the nature of dataflow programming. As stated previously, tests of common LabVIEW program structures show, on average, a 25 to 35 percent improvement in execution time when moved from a single to a multicore system. The nature of an individual program, however, significantly affects this estimate. Optimizing a LabVIEW program for a multicore computing environment can result in large execution time reductions when upgrading to a multicore computing system.

Hardware Specific Tuning

Organizing code to increase execution speed is complicated when you do not know the hardware on which you are executing the program. Understanding the system a multicore program is running on is vital to achieving maximum execution speed. Multicore programming techniques require a more generic approach for systems with an unknown number of cores. This approach helps ensure some execution time reduction on most multicore machines, but may hinder maximum execution speed on any specific system. Hardware—specific tuning of programs can be time consuming and is not always necessary; however, it may be necessary if you require maximum execution speed on specific hardware. For example, to fully utilize an octal—core computing system, you can employ advanced parallel programming techniques such as data parallelism or pipelining. Additionally, you can take advantage of the number of cores on a system, the core layout (two dual—cores or one quad—core), the connection scheme, the memory scheme, and information about known bugs to achieve minimal program execution times on multicore systems.

Software Stack Readiness

Bottlenecks in parallelism may arise at multiple levels of the software stack; avoiding this problem is a challenge in traditional languages such as C. An advantage of LabVIEW programming is the "multicore-ready" software stack, which removes these bottlenecks up front by providing re-entrant capabilities as well as thread-safe and re-entrant drivers. To realize the performance gains that are possible with multicore hardware, the software stack has four layers that must be evaluated to determine multicore readiness: the development tool, libraries, device drivers, and the operating system. If these layers are not multicore ready, performance gains are unlikely, and performance degradation may occur. Table 1 shows the different layers of the software stack that must be evaluated regardless of programming language.

Software Stack	Meaning of "Multicore-Ready"	LabVIEW Support
Development tool	Support provided on the operating system of choice; tool facilitates correct threading and optimization	✓ Example: Multithreaded nature of LabVIEW and structures that provide optimization
Libraries	Thread-safe, re-entrant libraries	✓ Example: Analysis libraries
Device drivers	Drivers designed for optimal multithreaded performance	✓ Example: NI-DAQmx driver software
Operating system	Operating system supports multithreading and multitasking and can load balance tasks	✓ Example: Support for Windows, Mac OS, Linux® OS, and real-time operating systems

Table 1. Software Stack Readiness

Conclusion

A number of factors must be considered when determining the expected execution time of LabVIEW programs on multicore systems. Hardware and software issues outside of LabVIEW can hinder execution time improvements when upgrading to a multicore system if not properly configured. In addition, a LabVIEW program's structure is often the main issue when considering multicore performance. LabVIEW helps minimize the factors that must be considered when upgrading to multicore computing systems by being "multicore-ready" and providing simple and intuitive multicore programming capabilities and examples.

Multithreaded Features of NI LabVIEW Functions and Drivers

Multithreaded programming is a key approach for taking advantage of multicore processors. By splitting your application into multiple threads, the operating system can balance - or schedule - these threads across multiple processing cores available in the PC. This document discusses the benefits, with respect to multicore processors, that come from using multithread-safe and reentrant functions and drivers in National Instruments LabVIEW.

Parallel vs. Multithread-Safe Behavior

In traditional languages, you must break up your program into different threads for parallel execution. Each thread can run at the same time. However, there is a difference between writing code that is safe to run in a multithreaded application, and writing code that executes in parallel to maximize the performance you get from a multicore system. This difference is often illustrated in the drivers or functions that you might use when writing programs. Multithread-safe functions can be called from multiple threads and do not overwrite their data, which prevents conflicts by blocking execution. If one thread is calling the function, any other threads trying to call that function must wait until the first thread is finished. Reentrant functions go a step further by allowing multiple threads to call and execute the same function at the same time, in parallel. Both of these examples execute correctly in a multithreaded program, but you can execute faster with reentrant functions because they run concurrently.

Memory vs. Performance Trade-Off in LabVIEW

In LabVIEW, the data carried on wires is generally independent of the functions that operate on the data. By definition, the data on a wire is not easily accessed by VIs or functions that are not connected to that wire. If necessary, LabVIEW makes a copy of the data when you split a wire so there are independent versions of the data for subsequent VIs to operate on. In addition, most LabVIEW VIs and drivers are both multithread-safe and reentrant. However, you can set the default configuration of some of the built-in library VIs in LabVIEW to be nonreentrant. Because reentrant VIs use more memory, you may need to make a trade-off between memory usage and parallelism. In many cases, LabVIEW opts toward the memory savings or nonreentrant configuration by default, and lets you decide if the goal is maximum parallelism. If so, the library VIs can easily switch to a reentrant setting.

Reentrant Functions

There are cases and environments in which you may need to use a nonreentrant function or program to prevent problems with accessing functions. Many multithread-safe libraries guarantee their “safety” by locking resources - meaning when one thread calls the function, that function or even the entire library is locked so that no other thread can call it. In a parallel situation, if two different paths of your code, or threads, are attempting to call into the same library or function, the lock forces one of the threads to wait, or block the thread, until the other one completes. Also, permitting only one thread at a time to access a function saves memory space because no extra instances are needed.

However, as mentioned previously, combining parallel programming techniques with reentrancy in your functions can help drive performance in your code.

Because device drivers with LabVIEW (such as NI-DAQmx) are both multithread-safe and reentrant, your function can be called by multiple threads at the same time and still operate correctly without blocking. This is an important feature for writing parallel code and optimizing performance with multicore systems. If you are using code that does not have reentrant execution, this could be why your performance has not increased: your code has to wait until the other threads are done using each function before it can access them. To illustrate this point further, take a look at the **VI Hierarchy** feature in LabVIEW. To view an individual VI's hierarchy, select **View » VI Hierarchy**. In the VI Hierarchy shown in Figure 1, both F1 and F2 are dependent on the same VI (in this case a very processing-intensive fast Fourier transform algorithm). It is important that this VI is reentrant if F1 and F2 are to execute in parallel.

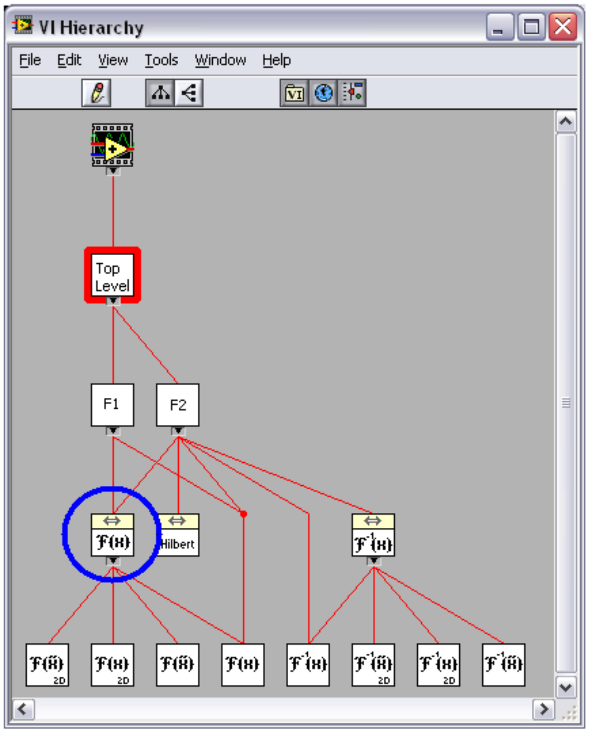
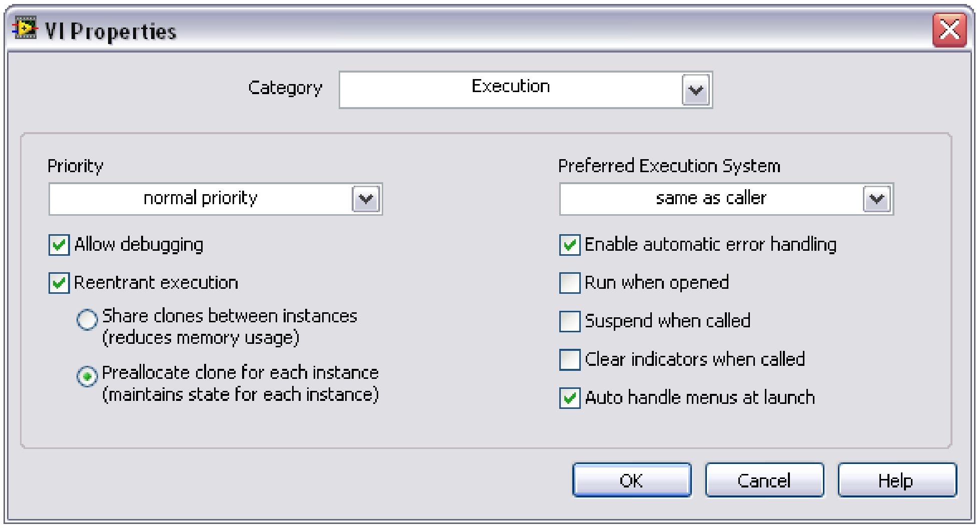


Figure 1. The VI Heirarcy allows you to view dependencies in LabVIEW programs

Reentrancy is an important consideration to eliminate any unnecessary dependencies in your code. Some analysis VIs in LabVIEW are nonreentrant or reentrant by default, so it is important to view the properties of those VIs to ensure they execute in parallel.

LabVIEW Configuration

To set your LabVIEW VI to be reentrant, select **File» VI Properties**, then select **Execution** from the drop-down menu. You now can check the box next to **Reentrant Execution** and decide which cloning option you want to choose.



Category: Execution

Priority: normal priority

☑ Allow debugging

☑ Reentrant execution

○ Share clones between instances (reduces memory usage)

● Preallocate clone for each instance (maintains state for each instance)

Preferred Execution System: same as caller

☑ Enable automatic error handling

□ Run when opened

□ Suspend when called

□ Clear indicators when called

☑ Auto handle menus at launch

Figure 2. VI Properties allows you to set the VI execution to reentrant

LabVIEW supports two types of reentrant VIs. Select the **Preallocate clone for each instance** option if you want to create a clone VI for each call to the reentrant VI before LabVIEW calls the reentrant VI, or if a clone VI must preserve state information across calls. For example, if a reentrant VI contains an uninitialized shift register or a local variable, property, or method that contains values that must remain for future calls to the clone VI, select the **Preallocate clone for each instance** option. Also select this option if the reentrant VI contains the **First Call?** function. This is also the recommended setting for VIs running on LabVIEW Real-Time systems, to ensure minimal jitter.

Select the **Share clones between instances** option to reduce the memory usage associated with preallocating a large number of clone VIs. When you select the **Share clones between instances** option, LabVIEW does not create the clone VI until a VI makes a call to the reentrant VI. With this option, LabVIEW creates the clone VIs on demand, potentially introducing jitter into the execution of the VI. LabVIEW does not preserve state information across calls to the reentrant VI.

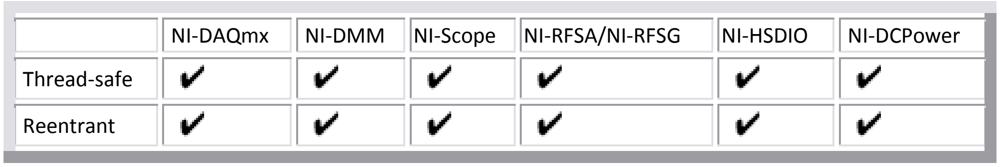
LabVIEW Driver Abilities

Using drivers that are thread-safe and reentrant is important if you interface with any type of hardware. With these attributes, you can take advantage of multicore technology for performance increases.

With previous versions of LabVIEW, reentrancy was not always a given with device drivers. Traditional NI-DAQ, for example, was multithread-safe in that it would not fail if two different threads called it at the same time. It handled this with a global lock - meaning once a thread called any NI-DAQ function, all other threads would have to wait until that function was complete before they could execute any NI-DAQ functions.

Alternately, NI-DAQmx works much more elegantly in a parallel, multithreaded environment. NI-DAQmx is reentrant –multiple threads can call into the driver simultaneously. You can execute two different analog inputs from two different boards from separate threads within the same program, and they both run simultaneously without blocking. In addition, you can execute an analog input and a digital input from two completely different threads at the same time on the same board. This effectively treats a single hardware resource as two separate resources because of the sophistication of the NI-DAQmx driver.

National Instruments modular instrument drivers also operate like NI-DAQmx. All of the drivers listed in Table 1 are both thread-safe and reentrant. They all provide the ability to call two of the same functions on two different devices at the same time. This is especially advantageous for large systems with code using multiple instruments.



	NI-DAQmx	NI-DMM	NI-Scope	NI-RFSA/NI-RFSG	NI-HSDIO	NI-DCPower
Thread-safe	✔	✔	✔	✔	✔	✔
Reentrant	✔	✔	✔	✔	✔	✔

Table 1. National Instrument drivers are both Thread-safe and Reentrant

Conclusion

When using parallel programming to take advantage of multicore architectures, it's important to not only think about the programming language and everything involved in writing parallel code, but also to make sure your drivers and functions are appropriate for a parallel environment.

Optimizing Automated Test Applications for Multicore Processors with NI LabVIEW

LabVIEW provides a unique and easy-to-use graphical programming environment for automated test applications. However, it is its ability to dynamically assign code to various CPU cores that improves execution speeds on multi-core processors. Learn how LabVIEW applications can be optimized to take advantage of parallel programming techniques.

The Challenge of Multi-Threaded Programming

Until recently, innovations in processor technology have resulted in computers with CPUs that operate at higher clock rates. However, as clock rates approach their theoretical physical limits, companies are developing new processors with multiple processing cores. With these new multicore processors, engineers developing automated test applications can achieve the best performance and highest throughput by using parallel programming techniques. Dr. Edward Lee, an electrical and computer engineering professor at the University of California - Berkeley, describes the benefits of parallel processing.

> *"Many technologists predict that the end of Moore's Law will be answered with increasingly parallel computer architectures. If we hope to continue to get performance gains in computing, programs must be able to exploit this parallelism."*

Moreover, industry experts recognize that it is a significant challenge for programming applications to take advantage of multicore processors. Bill Gates, founder of Microsoft, Inc., explains.

> *"To fully exploit the power of processors working in parallel…software must deal with the problem of concurrency. But as any developer who has written multithreaded code can tell you, this is one of the hardest tasks in programming."*

Fortunately, National Instruments LabVIEW software offers an ideal multicore processor programming environment with an intuitive API for creating parallel algorithms that can dynamically assign multiple threads to a given application. In fact, you can optimize automated test applications using multicore processors to achieve the best performance.

Moreover, PXI Express modular instruments enhance this benefit because they take advantage of the high data transfer rates possible with the PCI Express bus. Two specific applications that benefit from multicore processors and PXI Express instruments are multichannel signal analysis and in-line processing (hardware in the loop). This white paper evaluates various parallel programming techniques and characterizes the performance benefits that each technique produces.

Implementing Parallel Test Algorithms

One common automated test application that benefits from parallel processing is multichannel signal analysis. Because frequency analysis is a processor-intensive operation, you can improve execution speed by running test code in parallel so that each channel's signal processing can be distributed to multiple processor cores. From a programmer's perspective, the only change you need to make to gain this benefit is a minor restructuring of the test algorithm.

To illustrate, compare the execution times of two algorithms for multichannel frequency analysis (fast Fourier transform, or FFT) on two channels of a high-speed digitizer. The National Instruments PXIe-5122 14-bit high-speed digitizer uses two channels to acquire signals at the maximum sample rate (100 MS/s). First, examine the traditional sequential programming model for this operation in LabVIEW.

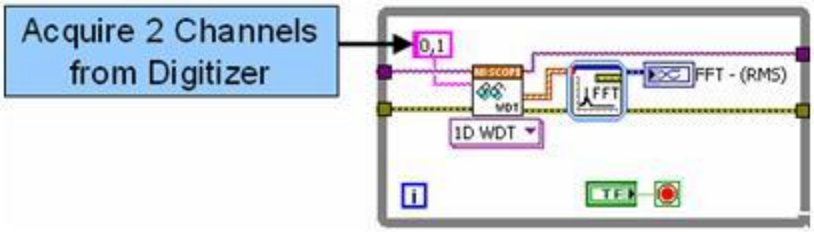


Figure 1. LabVIEW code utilizes sequential execution

In Figure 1, frequency analysis of both channels is performed in an FFT Express VI, which analyzes each channel in series. While the algorithm shown above can still be executed efficiently in multicore processors, you can improve algorithm performance by processing each channel in parallel.

If you profile the algorithm, you notice that the FFT takes considerably longer to complete than the acquisition from the high-speed digitizer. By fetching each channel one at a time and performing two FFTs in parallel, you can significantly reduce the processing time. See Figure 2 for a new LabVIEW block diagram that uses the parallel approach.

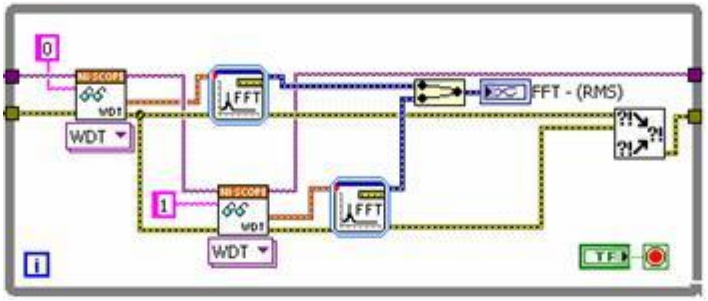


Figure 2. LabVIEW code utilizes parallel execution

Each channel is fetched from the digitizer sequentially. Note that you could perform these operations completely in parallel if both fetches were from unique instruments. However, because an FFT is processor-intensive, you can still improve performance simply by running the signal processing in parallel. As a result, the total execution time is reduced. Figure 3 shows the execution time of both implementations.

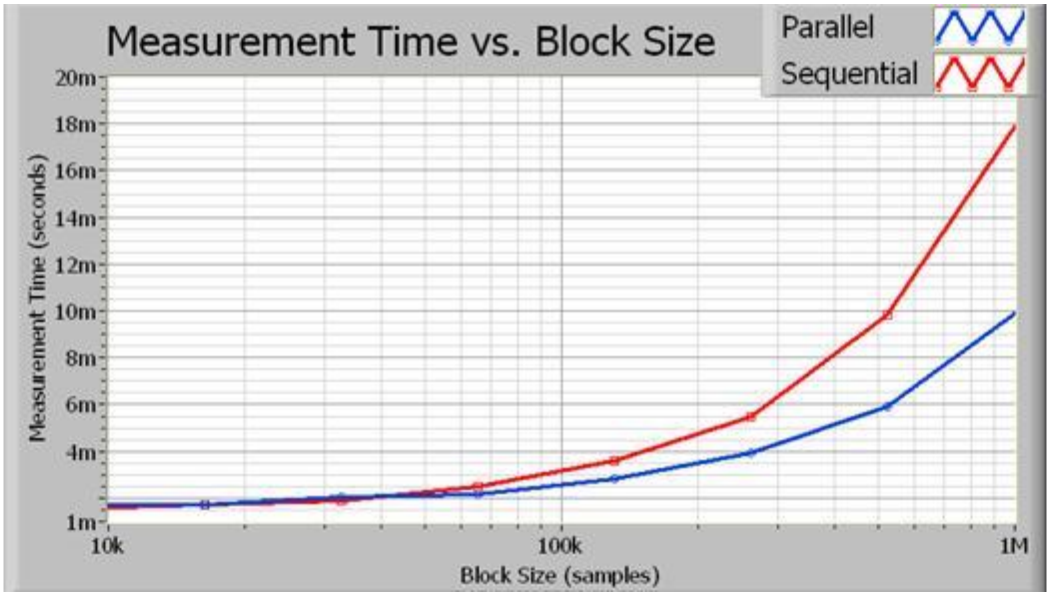


Figure 3. As the block size increases, the processing time saved through parallel execution becomes more obvious

In fact, the parallel algorithm approaches a two times performance improvement for larger block sizes. Figure 4 illustrates the exact percent increase in performance as a function of acquisition size (in samples).

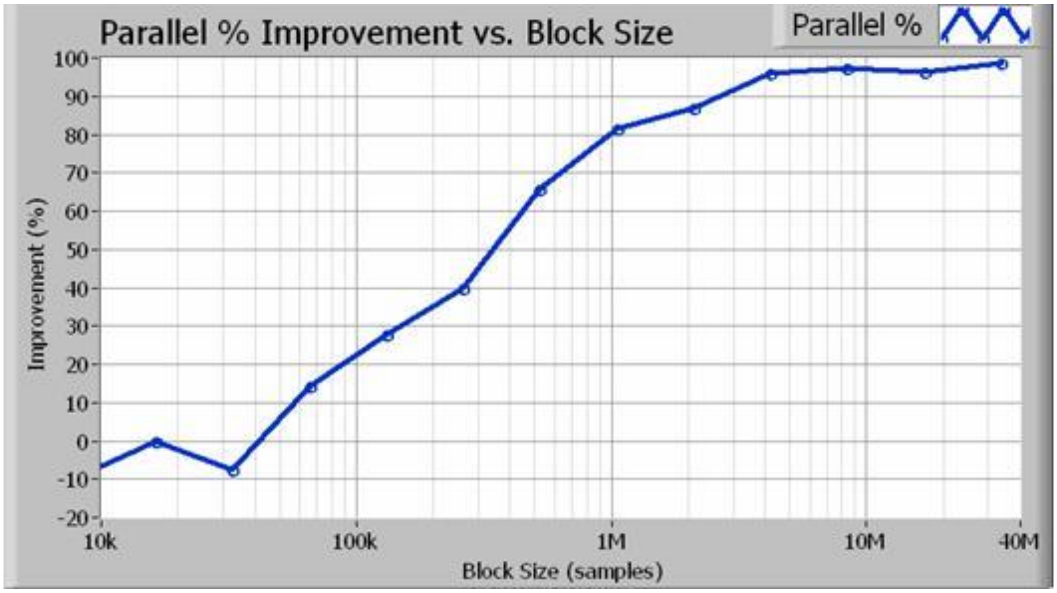


Figure 4. For block sizes greater than 1 million samples (100 Hz resolution bandwidth), the parallel approach results in an 80 percent or better performance increase

Increasing the performance of automated test applications is easy to achieve on multicore processors because you allocate each thread dynamically using LabVIEW. In fact, you are not required to create special code to enable multithreading. Instead, parallel test applications benefit from multicore processors with minimal programming adjustments.

Configuring Custom Parallel Test Algorithms

Parallel signal processing algorithms help LabVIEW divide processor usage among multiple cores. Figure 5 illustrates the order in which the CPU processes each part of the algorithm.

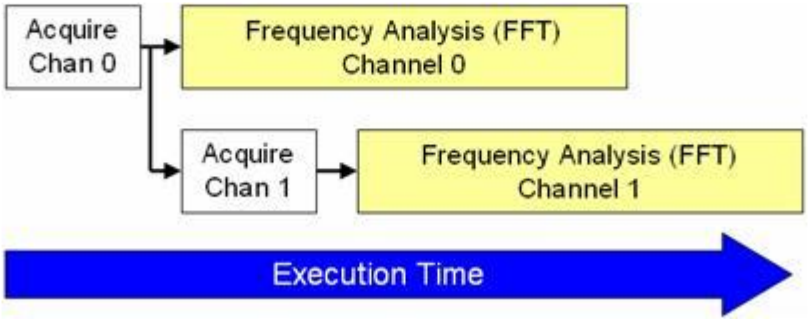
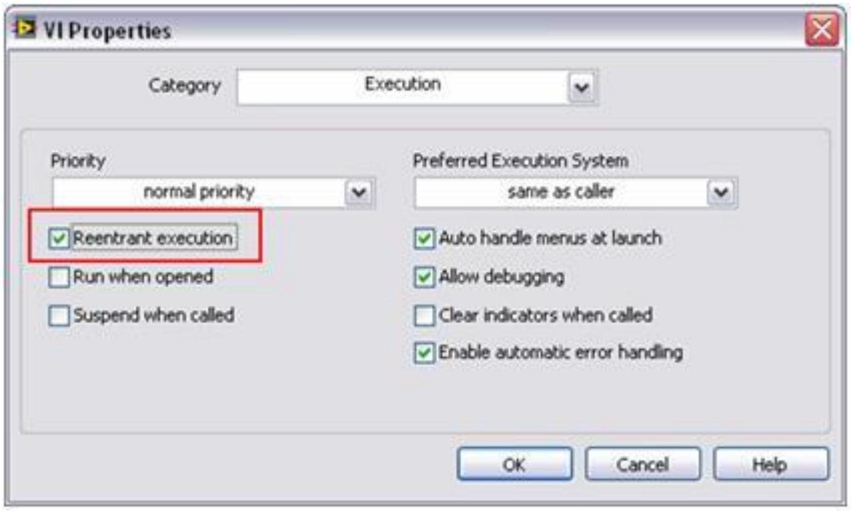

Acquire Chan 0
Frequency Analysis (FFT) Channel 0
Acquire Chan 1
Frequency Analysis (FFT) Channel 1
Execution Time

Figure 5. LabVIEW can process much of the acquired data in parallel, saving execution time

Parallel processing requires LabVIEW to make a copy (or clone) of each signal processing subroutine. By default, many LabVIEW signal processing algorithms are configured to have “reentrant execution.” This means that LabVIEW dynamically allocates a unique instance of each subroutine, including separate threads and memory space. As a result, you must configure custom subroutines to operate in a reentrant fashion. You can do this with a simple configuration step in LabVIEW. To set this property, select File >> VI Properties and choose the Execution category. Then, select the Reentrant execution flag as shown in Figure 6.



Category: Execution

Priority: normal priority

Preferred Execution System: same as caller

☑ Reentrant execution

□ Run when opened

□ Suspend when called

☑ Auto handle menus at launch

☑ Allow debugging

□ Clear indicators when called

☑ Enable automatic error handling

Figure 6. With this simple step, you can execute multiple custom subroutines in parallel, just like standard LabVIEW analysis functions

As a result, you can achieve improved performance in your automated test applications on multicore processors by using simple programming techniques.

Optimizing Hardware-in-the-Loop Applications

A second application that benefits from parallel signal processing techniques is the use of multiple instruments for simultaneous input and output. In general, these are referred to as hardware-in-the-loop (HIL) or in-line processing applications. In this scenario, you may use either a high-speed digitizer or high-speed digital I/O module to acquire a signal. In your software, you perform a digital signal processing algorithm. Finally, the result is generated by another modular instrument. A typical block diagram is illustrated in Figure 7.

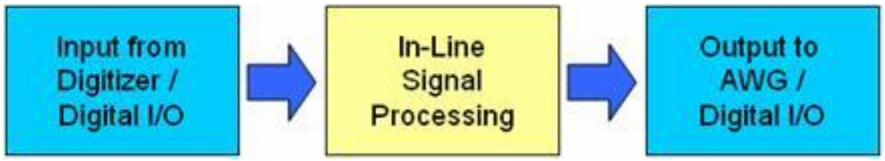

Input from Digitizer / Digital I/O
In-Line Signal Processing
Output to AWG / Digital I/O

Figure 7. This diagram shows the steps in a typical hardware-in-the-loop (HIL) application

Common HIL applications include in-line digital signal processing (such as filtering and interpolation), sensor simulation, and custom component emulation. You can use several techniques to achieve the best throughput for in-line digital signal processing applications.

In general, you can use two basic programming structures – the single-loop structure and the pipelined multiloop structure with queues. The single-loop structure is simple to implement and offers low latency for small block sizes. In contrast, multiloop architectures are capable of much higher throughput because they better utilize multicore processors.
Using the traditional single-loop approach, you place a high-speed digitizer read function, signal processing algorithm, and high-speed digital I/O write function in sequential order. As the block diagram in Figure 8 illustrates, each of these subroutines must execute in a series, as determined by the LabVIEW programming model.

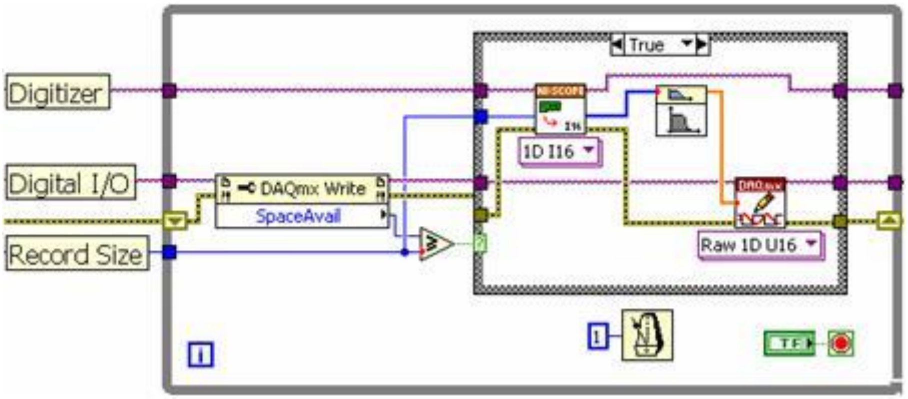


Figure 8. With the LabVIEW single-loop approach, each subroutine must execute in series

The single-loop structure is subject to several limitations. Because each stage is performed in a series, the processor is limited from performing instrument I/O while processing the data. With this approach, you cannot efficiently use a multicore CPU because the processor only executes one function at a time. While the single-loop structure is sufficient for lower acquisition rates, a multiloop approach is required for higher data throughput.

The multiloop architecture uses queue structures to pass data between each while loop. Figure 9 illustrates this programming between while loops with a queue structure.

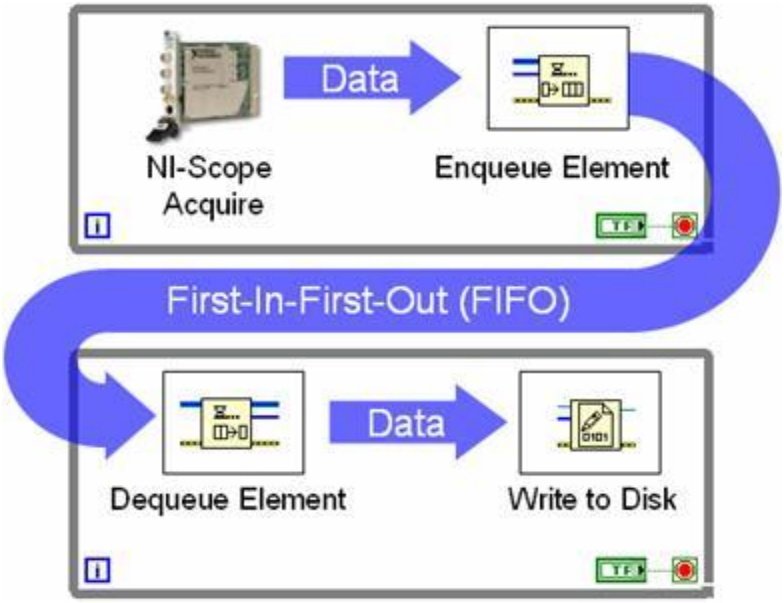


Figure 9. With queue structures, multiple loops can share data

Figure 9 represents what is typically referred to as a producer-consumer loop structure. In this case, a high-speed digitizer acquires data in one loop and passes a new data set to the FIFO during each iteration. The consumer loop simply monitors the queue status and writes each data set to disk when it becomes available. The value of using queues is that both loops execute independently of one another. In the example above, the high-speed digitizer continues to acquire data even if there is a delay in writing it to disk. The extra samples are simply stored in the FIFO in the meantime. Generally, the producer-consumer pipelined approach provides greater data throughput with more efficient processor utilization. This advantage is even more apparent in multicore processors because LabVIEW dynamically assigns processor threads to each core.

For an in-line signal processing application, you can use three independent while loops and two queue structures to pass data among them. In this scenario, one loop acquires data from an instrument, one performs dedicated signal processing, and the third writes data to a second instrument.

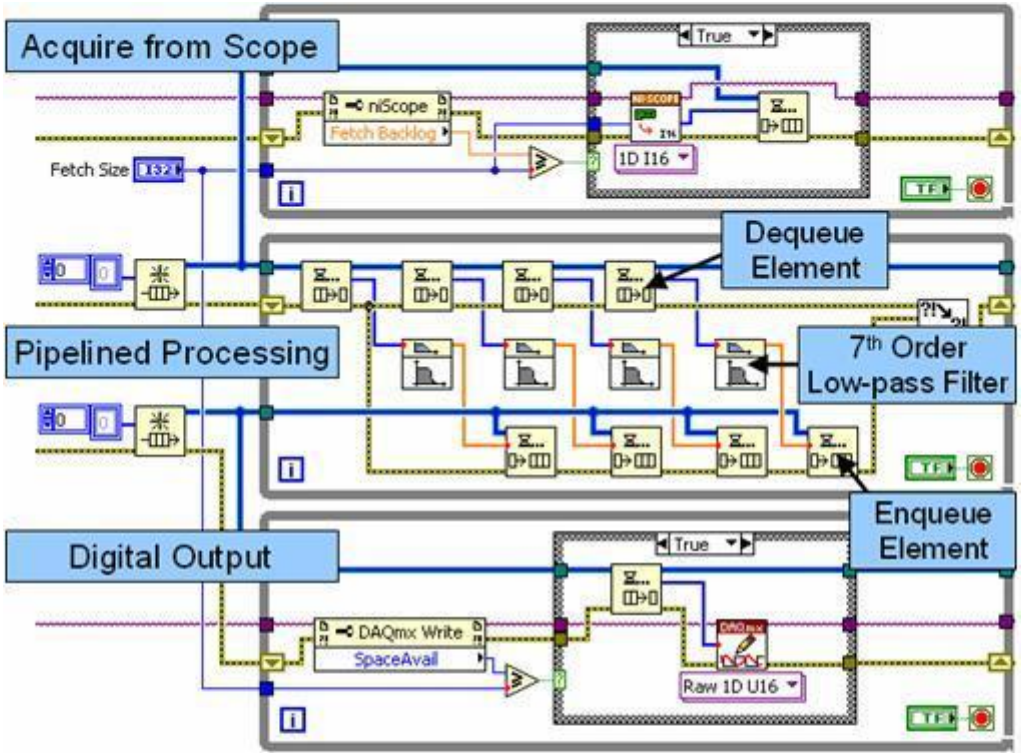


Figure 10. This block diagram illustrates pipelined signal processing with multiple loops and queue structures

In Figure 10, the top loop is a producer loop that acquires data from a high-speed digitizer and passes it to the first queue structure (FIFO). The middle loop operates as both a producer and a consumer. During each iteration, it unloads (consumes) several data sets from the queue structure and processes them independently in a pipelined fashion. This pipelined approach improves performance in multicore processors by processing up to four data sets independently. Note that the middle loop also operates as a producer by passing the processed data into the second queue structure. Finally, the bottom loop writes the processed data to a high-speed digital I/O module.

Parallel processing algorithms improve processor utilization on multicore CPUs. In fact, the total throughput is dependent on two factors – processor utilization and bus transfer speeds. In general, the CPU and data bus operate most efficiently when processing large blocks of data. Also, you can reduce data transfer times even further using PXI Express instruments, which offer faster transfer times.

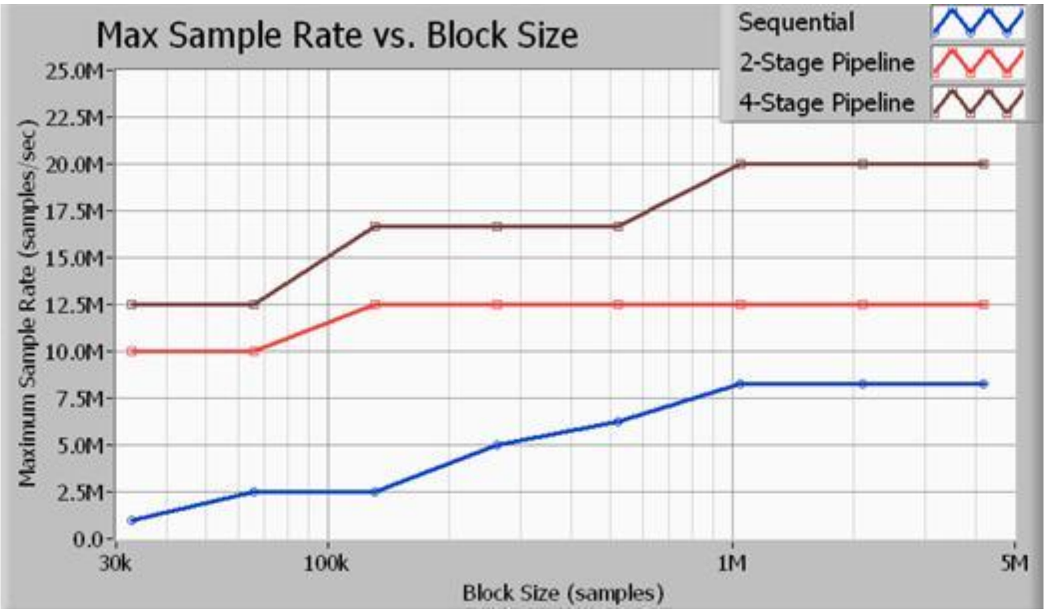
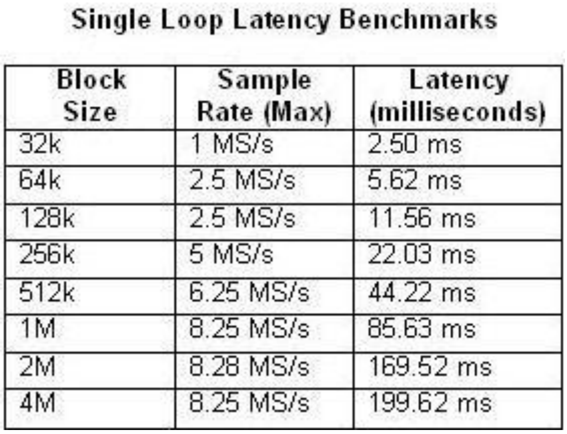


Figure 11. The throughput of multiloop structures is much faster than single-loop structures

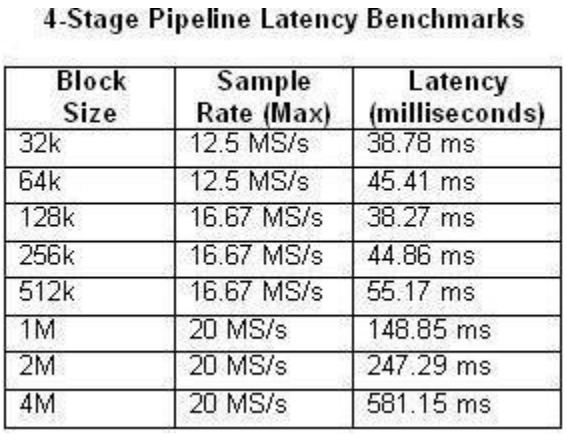
Figure 11 illustrates the maximum throughput in terms of sample rate, according to the acquisition size in samples. All benchmarks illustrated here were performed on 16-bit samples. In addition, the signal processing algorithm used was a seventh-order Butterworth low-pass filter with a cutoff of 0.45x sample rate. As the data illustrates, you achieve the most data throughput with the four-stage pipelined (multiloop) approach. Note that a two-stage signal processing approach yields better performance than the single-loop method (sequential), but it does not use the processor as efficiently as the four-stage method. The sample rates listed above are the maximum sample rates of both input and output for an NI PXIe-5122 high-speed digitizer and an NI PXIe-6537 high-speed digital I/O module. Note that at 20 MS/s, the application bus is transferring data at rates of 40 MB/s for input and 40 MB/s for output for a total bus bandwidth of 80 MB/s.

It is also important to consider that the pipelined processing approach does introduce latency between input and output. The latency is dependent upon several factors, including the block size and sample rate. Tables 1 and 2 below compare the measured latency according to block size and maximum sample rate for the single-loop and four-stage multiloop architectures.



Single Loop Latency Benchmarks

Block Size	Sample Rate (Max)	Latency (milliseconds)
32k	1 MS/s	2.50 ms
64k	2.5 MS/s	5.62 ms
128k	2.5 MS/s	11.56 ms
256k	5 MS/s	22.03 ms
512k	6.25 MS/s	44.22 ms
1M	8.25 MS/s	85.63 ms
2M	8.28 MS/s	169.52 ms
4M	8.25 MS/s	199.62 ms



4-Stage Pipeline Latency Benchmarks

Block Size	Sample Rate (Max)	Latency (milliseconds)
32k	12.5 MS/s	38.78 ms
64k	12.5 MS/s	45.41 ms
128k	16.67 MS/s	38.27 ms
256k	16.67 MS/s	44.86 ms
512k	16.67 MS/s	55.17 ms
1M	20 MS/s	148.85 ms
2M	20 MS/s	247.29 ms
4M	20 MS/s	581.15 ms

Tables 1 and 2. These tables illustrate the latency of single-loop and four-stage pipeline benchmarks

As you expect, the latency increases as the CPU usage approaches 100 percent utilization. This is particularly evident in the four-stage pipeline example with a sample rate of 20 MS/s. By contrast, the CPU usage barely exceeds 50 percent in any of the single-loop examples.

Conclusion

PC-based instrumentation such as PXI and PXI Express modular instruments benefit greatly from advances in multicore processor technology and improved data bus speeds. As new CPUs improve performance by adding multiple processing cores, parallel or pipelined processing structures are necessary to maximize CPU efficiency. Fortunately, LabVIEW solves this programming challenge by dynamically assigning processing tasks to individual processing cores. As illustrated, you can achieve significant performance improvements by structuring LabVIEW algorithms to take advantage of parallel processing.

Programming Strategies for Multicore Programming: Task Parallelism

Task Parallelism

Task parallelism is simply the concurrent execution of independent tasks in software. Consider a single-core processor that is running a Web browser and a word-processing program at the same time. Although these applications run on separate threads, they still ultimately share the same processor. Now consider a second scenario in which the same two programs are running on a dual-core processor. On the dual-core machine, these two applications essentially can run independently of one another. Although they may share some resources that prevent them from running completely independently, the dual-core machine can handle the two parallel tasks more efficiently.

Task Parallelism in LabVIEW

The LabVIEW graphical programming paradigm allows developers to implement task parallelism by drawing parallel sections of code, or branches in code. Two separate tasks that are not dependent on one another for data run in parallel without the need for any extra programming.

Figure 1 shows a simple data acquisition routine. The top section of code consists of an analog voltage input task, and the bottom code is a digital output task.

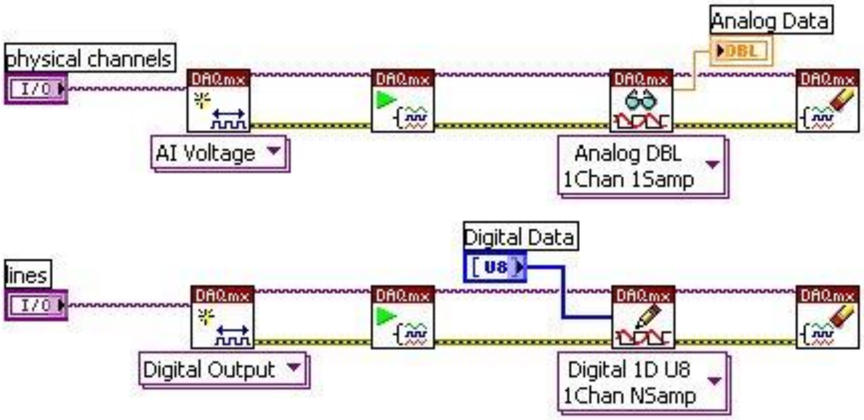


Figure 1. Independent Parallel Analog Input and Digital Output Tasks in LabVIEW

These two independent data acquisition tasks must share the same CPU on a single-core processor. On a multicore processor, each thread can run on its own processor, greatly improving the execution speed.

When using data parallelism in an application, it is important to ensure that the two tasks do not have any shared resources that could create a bottleneck, such as the code in Figure 2.

An Application of Task Parallelism

At Eaton Corporation, a LabVIEW application for Transmission Testing based on a state-machine architecture with three primary tasks, was able to more than quadruple the number of data acquisition channels. This LabVIEW application comprises of three asynchronous loops completing the parallel tasks of acquisition, test control, and user interface. Before implementing this solution, the team could only meet its testing requirements by either dividing the workload sequentially or by using several single-core desktops to run the application. Using standard, off-the-shelf desktop systems, the team was able to minimize power consumption, thermal output, and test time and thus reduce overall costs. They were also able to move the test system to a "mobile tester" and execute code in-vehicle using a laptop with a multicore processor.

Programming Strategies for Multicore Programming: Data Parallelism

As hardware designers turn toward multicore processors to improve computing power, software programmers must find new programming strategies that harness the power of parallel computing. One technique that effectively takes advantage of multicore processors is data parallelism. This is also known as the "Divide and Conquer" technique.

Data Parallelism

Data parallelism is a programming technique for splitting a large data set into smaller chunks that can be operated on in parallel. After the data has been processed, it is combined back into a single data set. With this technique, programmers utilize multicore processing power, so that it can efficiently use all processing power available.

Consider the scenario in Figure 1, which illustrates a large data set being operated on by a single processor. In this scenario, the other three CPU cores available are idle, while the first processor solely bears the load of processing the entire data set.

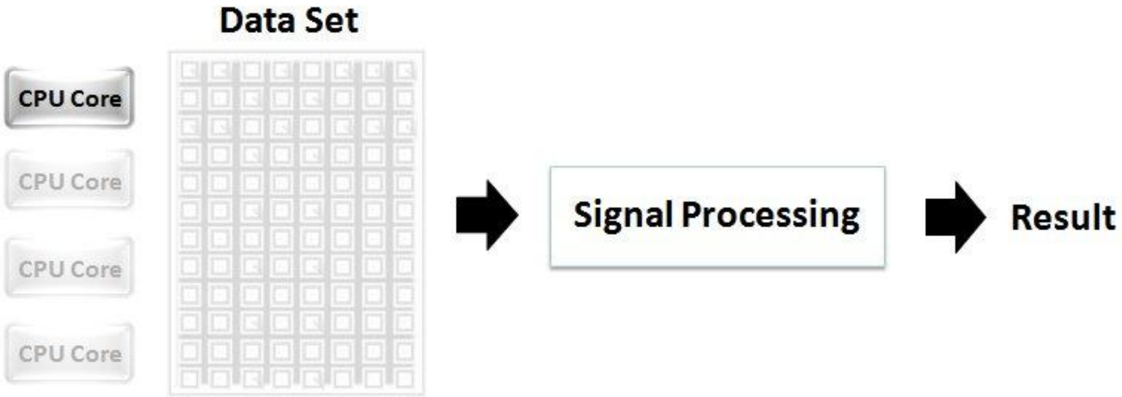


Figure 1: Processing a Large Data Set on a Single CPU

Now consider the implementation shown in Figure 2, which uses data parallelism to fully harness the processing power offered by a quad-core processor. In this case, the large data set is broken into four subsets. Each subset is assigned to an individual core for processing. After processing is complete, these subsets are rejoined into a single, full data set.

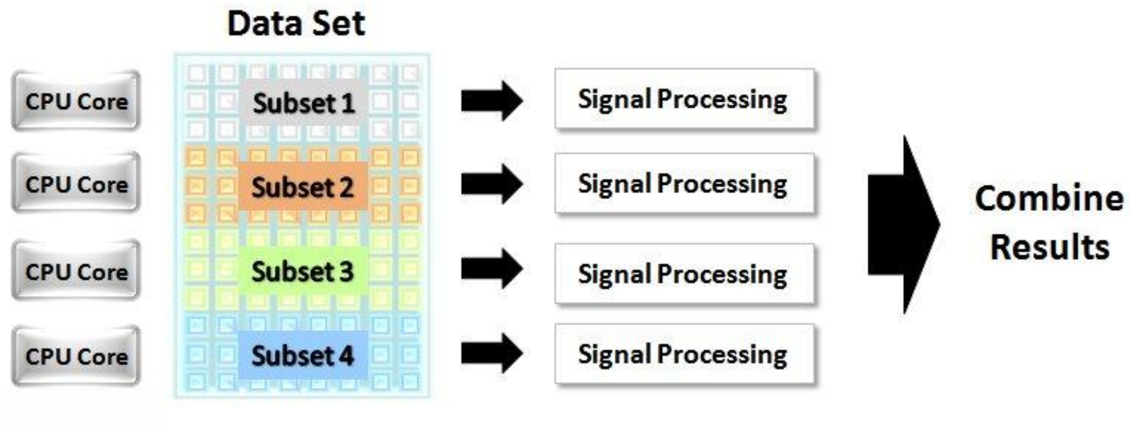


Figure 2: Processing a Large Data Set in Parallel on Four Cores

The graphical programming paradigm of National Instruments LabVIEW is ideal for parallel data architectures. Parallel processing in NI LabVIEW is intuitive and simple, in contrast to traditional text-based languages, which require advanced programming knowledge to create a multithreaded application.

Data Parallelism in LabVIEW

The code in Figure 3 shows a matrix multiply operation performed on two matrices, Matrix 1 and Matrix 2. This is a standard implementation in LabVIEW of multiplication of two large matrices.

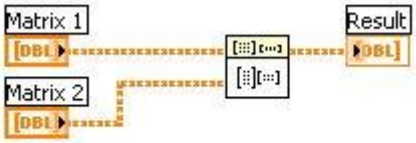


Figure 3: Matrix Multiplication in LabVIEW without Data Parallelism

An operation such as this can take a significant amount of time to complete, especially when large data sets are involved. The code in Figure 3 in no way capitalizes on the extra computer power offered by a multicore processor, unless by chance the Matrix Multiply VI is already multithreaded and fully optimized for a multicore processor. In contrast, the code in Figure 4 makes use of data parallelism and thus can execute significantly faster on a dual-core processor than the code shown in Figure 3.

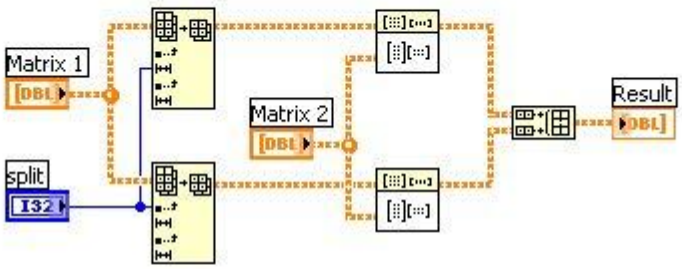


Figure 4: Matrix Multiplication in LabVIEW with Data Parallelism

When using this technique in LabVIEW to increase performance on a multicore processor, it is important to realize that a performance improvement cannot be achieved unless the Matrix Multiply VI is reentrant. If it is not reentrant, the separate instances of the Matrix Multiply VI cannot operate independently and concurrently.

An Application of Data Parallelism

At the Max Planck Institute in Munich, Germany, researchers applied data parallelism to a LabVIEW program that performs plasma control of Germany's most advanced nuclear fusion platform, the ASDEX tokamak. The program runs on an octal-core server, performing computationally intensive matrix operations in parallel on the eight CPU cores to maintain a 1 ms control loop. Lead researcher Louis Giannone notes, "In the first design stage of our control application programmed with LabVIEW, we have obtained a 20X processing speed-up on an octal-core processor machine over a single-core processor, while reaching our 1 ms control loop rate requirement."

Programming Strategies for Multicore Programming: Pipelining

Introduction to Pipelining

One widely accepted technique for improving the performance of serial software tasks is pipelining. Simply put, pipelining is the process of dividing a serial task into concrete stages that can be executed in assembly-line fashion.

Consider the following example: suppose you are manufacturing cars on an automated assembly line. Your task is building a complete car, but you can separate this into three concrete stages: building the frame, putting the parts inside (such as the engine), and painting the car when finished.

Assume that building the frame, installing parts, and painting take one hour each. Therefore, if you built just one car at a time each car would take three hours to complete (see Figure 1).

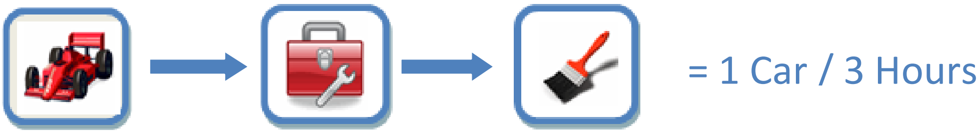


Figure 1. *In this example, building a single car on an assembly line takes three hours.*

How can this process be improved? What if we set up one station for frame building, another for part installation, and a third for painting. Now, while one car is being painted, a second car can have parts installed, and a third car can be under frame construction.

How Pipelining Improves Performance

Although each car still takes three hours to finish using our new process, we can now produce one car each hour rather than one every three hours - a 3X improvement in car manufacturing throughput.

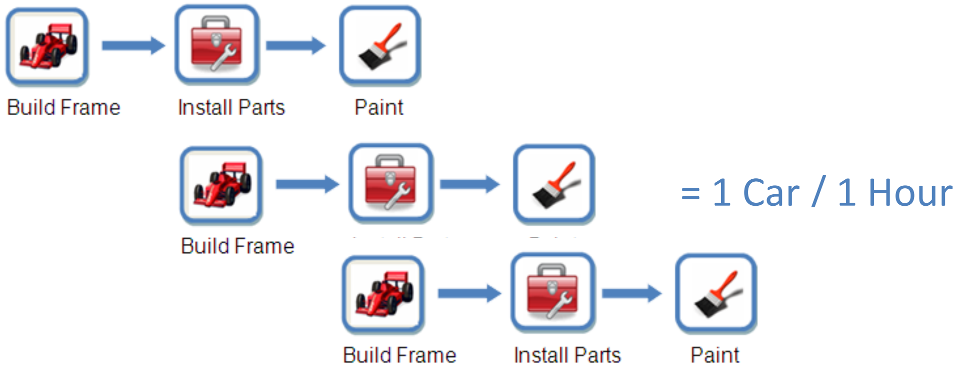


Figure 2. *Pipelining can dramatically increase the throughput of your application.*

Pipelining in LabVIEW

The same pipelining concept as visualized in the car example can be applied to any LabVIEW application in which you are executing a serial task. Essentially, you can use LabVIEW shift registers and feedback nodes to make an "assembly line" out of any given program.

For instance, consider an application in which you need to acquire data from a data acquisition device, perform fast Fourier transform (FFT) analysis, and then save the analysis results to disk. Notice that this is an inherently serial task; you cannot perform FFT analysis on a piece of data and save the analysis results for that data to disk at the same time. Instead, you can use pipelining to take advantage of multicore system architecture.

It is now useful to return to the concept of stages previously mentioned. Our example application can be divided into three concrete steps: acquisition, analysis, and storage. Assume that each stage takes one second to complete. This means that each iteration of the application takes three seconds to execute without pipelining. We can say that the application throughput is one computation every three seconds.

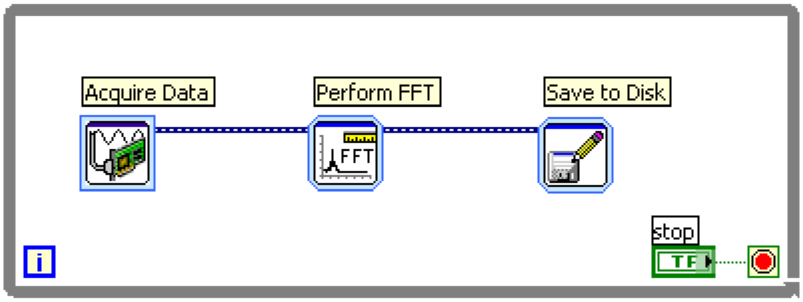


Figure 3. LabVIEW application executes in three seconds without pipelining

Figure 3. By placing a feedback node between each stage in LabVIEW, you can pipeline your application with very little programming effort.

After inserting the feedback nodes, each pipeline stage operates on the data provided by the previous stage (during a previous iteration). In this way, the pipeline can produce one computation during each iteration of the program - a 3X increase in throughput.

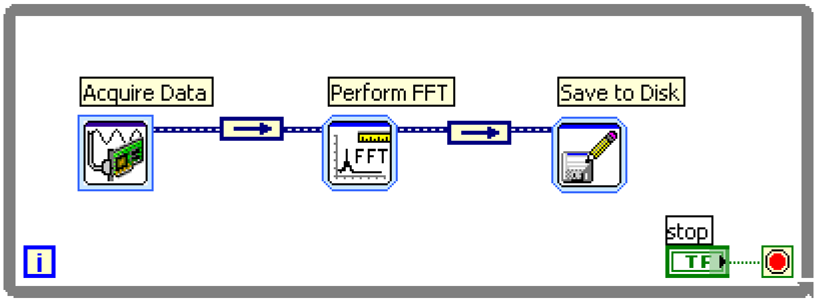


Figure 4. Pipelining application using feedback nodes increases throughput of 3x

Alternately, shift registers can be used instead of feedback nodes to implement pipelining in LabVIEW (the two methods are functionally equivalent).

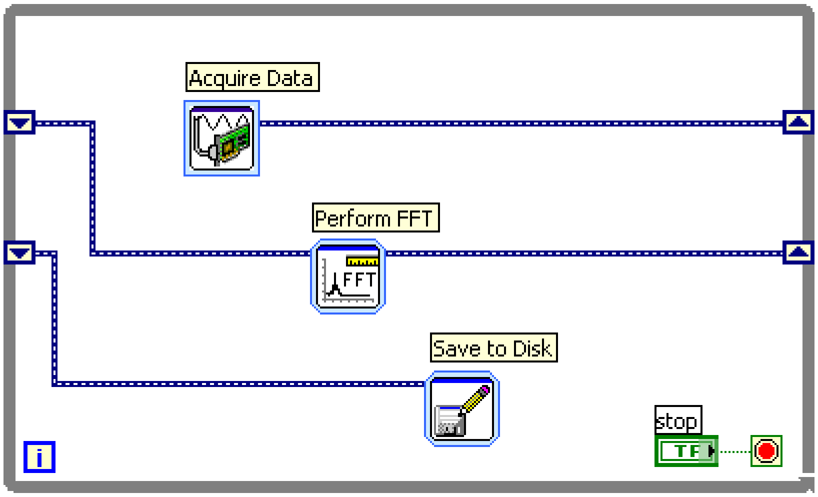


Figure 5. Pipelining application using shift registers increases throughput of 3x

Figure 5: Determining the execution time to identify well-balanced applications

To remedy this situation, the programmer must move tasks from Stage 1 to Stage 2 until both stages take approximately equal times to execute. With a large number of pipeline stages, this can be a difficult task.

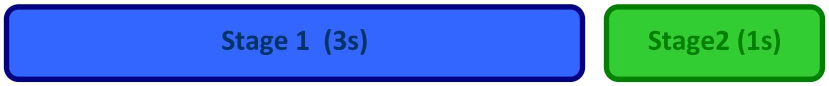
In LabVIEW, it is helpful to benchmark each of your pipeline stages to ensure that the pipeline is well balanced. This can most easily be done using a flat sequence structure in conjunction with the Tick Count (ms) function as shown below:

Important Concerns

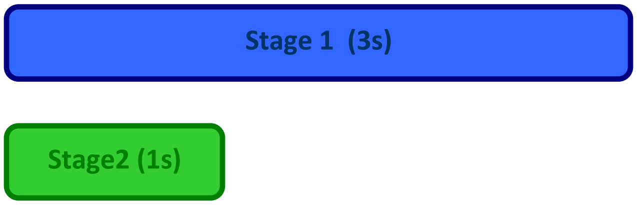
Balancing Stages

In both the car manufacturing and LabVIEW examples above, each pipeline stage was assumed to take an equal amount of time to execute; we can say that these example pipeline stages were *balanced*. However, in real-world applications this is rarely the case. Consider the diagram below; if Stage 1 takes three times as long to execute as Stage 2, then pipelining the two stages produces only a minimal performance increase.

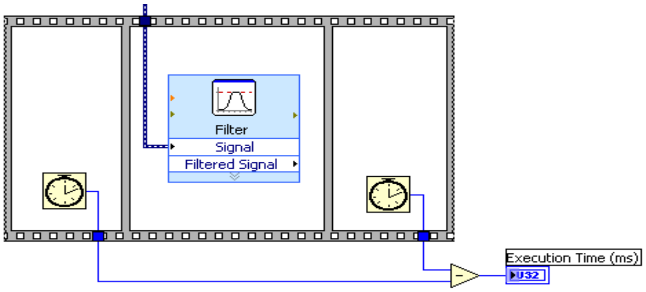
Non-Pipelined (total time = 4s)



Pipelined (total time = 3s): Speed-up = 1.33X (not an ideal case for pipelining)



To remedy this situation, the programmer must move tasks from Stage 1 to Stage 2 until both stages take approximately equal times to execute. With a large number of pipeline stages, this can be a difficult task. In LabVIEW, it is helpful to benchmark each of your pipeline stages to ensure that the pipeline is well balanced. This can most easily be done using a flat sequence structure in conjunction with the Tick Count (ms) function as shown below:



Data Transfer Between Cores

It is best to avoid transferring large amounts of data between pipeline stages whenever possible. Since the stages of a given pipeline could be running on separate processor cores, any data transfer between individual stages could actually result in a memory transfer between physical processor cores. In the case that two processor cores do not share a cache (or the memory transfer size exceeds the cache size), the end application user may see a decrease in pipelining effectiveness.

Conclusion

To summarize, pipelining is a technique that programmers can use to gain a performance increase in inherently serial applications (on multicore machines). In order to gain the most performance increase possible from pipelining, individual stages must be carefully balanced so that no single stage takes a much longer time to complete than other stages. In addition, any data transfer between pipeline stages should be minimized to avoid decreased performance due to memory access from multiple cores.

Introduction to LabVIEW Real-Time Symmetric Multiprocessing (SMP)

Symmetric multiprocessing, or SMP, is an OS-supported characteristic that allows for tasks to be easily moved between processors to balance the workload efficiently. Most modern OSs (Windows, Linux, MacOS,etc.) support SMP. However, this not commonly available in Real-Time Operating Systems.

With LabVIEW 8.5 Real-Time, support for SMP has been added to the operating system to allow for high-performance real-time performance based on multicore processor technology. Figure 1 below describes how Real-Time tasks can be automatically load balanced across the available processor cores in a system.

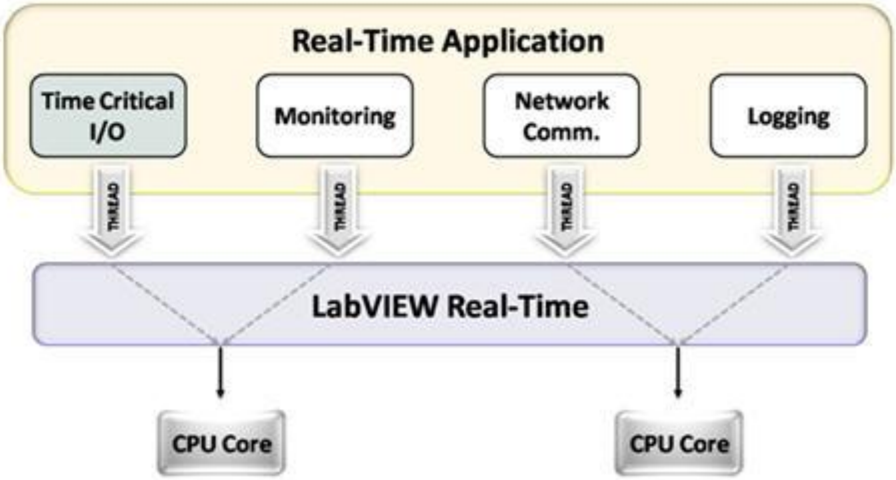


Figure 1 - LabVIEW Real-Time support for SMP

Assigning Tasks to Specific Processors with the Timed Loop

The default behavior of LabVIEW 8.5 Real-Time is to automatically schedules tasks to best utilize the processor resources of the system. However, in some scenarios, it may be advantage for developers to explicitly assign their code to specific processor core. For example, users can dedicate one core of their processor to perform the time critical control and isolate it from less important tasks that run on a different processor, as described in Figure 2 below.

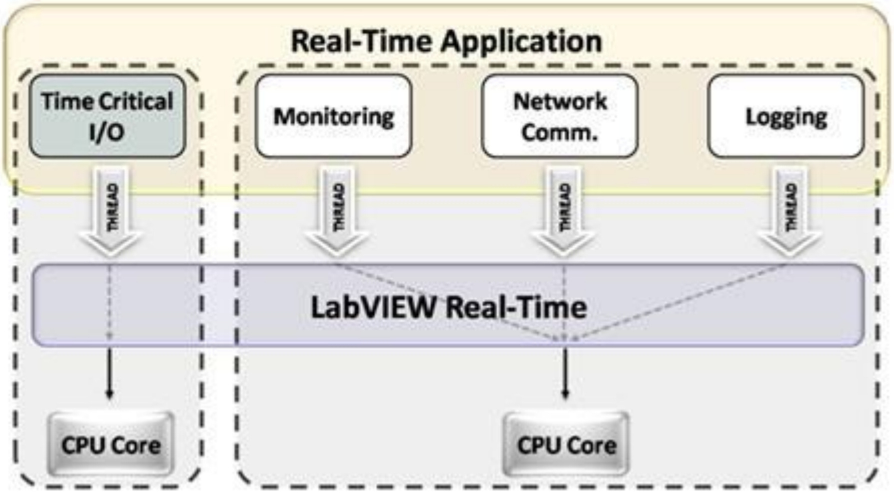
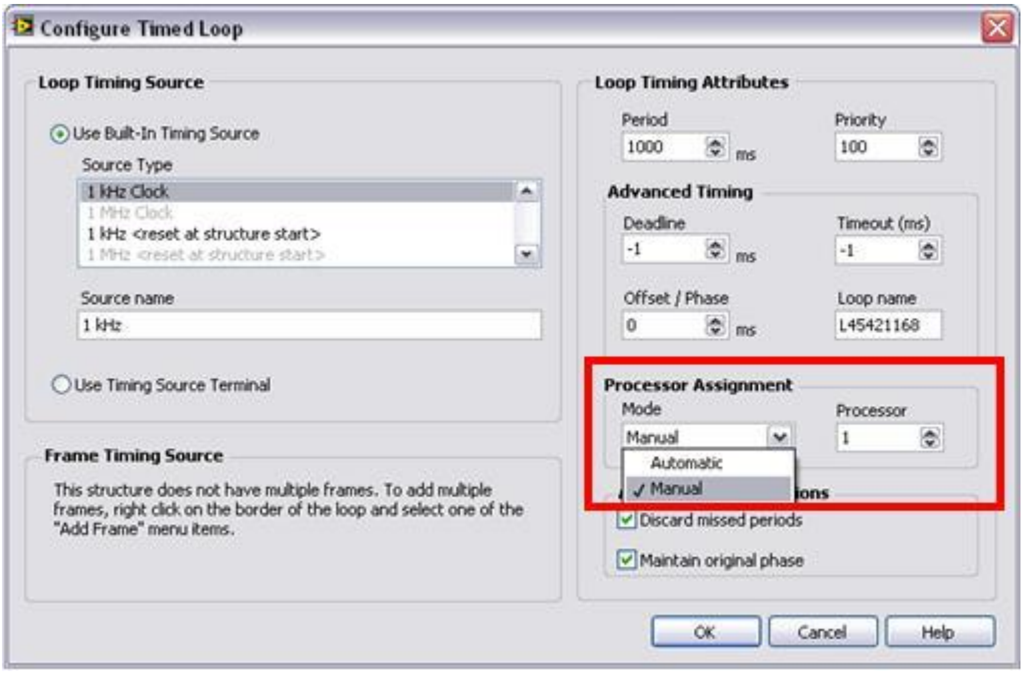


Figure 2 - Assigning a Time Critical Task to a Dedicated Processor

The timed loop structure in LabVIEW allows for this assignment to cores, using a function called processor affinity. Each timed loop represents a unique thread, and by setting processor affinity in the "configure timed loop" dialog box, a developer can designate the thread to only run on the specified processor. This configuration is outlined in Figure 3 below.



Configure Timed Loop

Loop Timing Source

● Use Built-In Timing Source

Source Type: 1 kHz Clock

- 1 kHz Clock
- 1 MHz Clock
- 1 kHz <reset at structure start>
- 1 MHz <reset at structure start>

Source name: 1 kHz

○ Use Timing Source Terminal

Frame Timing Source

This structure does not have multiple frames. To add multiple frames, right click on the border of the loop and select one of the "Add Frame" menu items.

Loop Timing Attributes

Period: 1000 ms

Priority: 100

Advanced Timing

Deadline: -1 ms

Timeout (ms): -1

Offset / Phase: 0 ms

Loop name: L45421168

Processor Assignment

Mode: Manual

Processor: 1

ions

☑ Discard missed periods

☑ Maintain original phase

Figure 3 - Assigning Processor Affinity with the Timed Loop

Debugging Multicore Real-Time Applications

As systems get more complex, it's important to understand at a low-level how code is being executed by the system. This is amplified by systems with more than one processor. The Real-Time Execution Trace Toolkit 2.0 provides a visual representation of both threads and functions executing on single core or multicore systems, so developers can find hotspots in their code and also detect undesirable behaviors such as resource contention, memory allocations, and priority inversion.

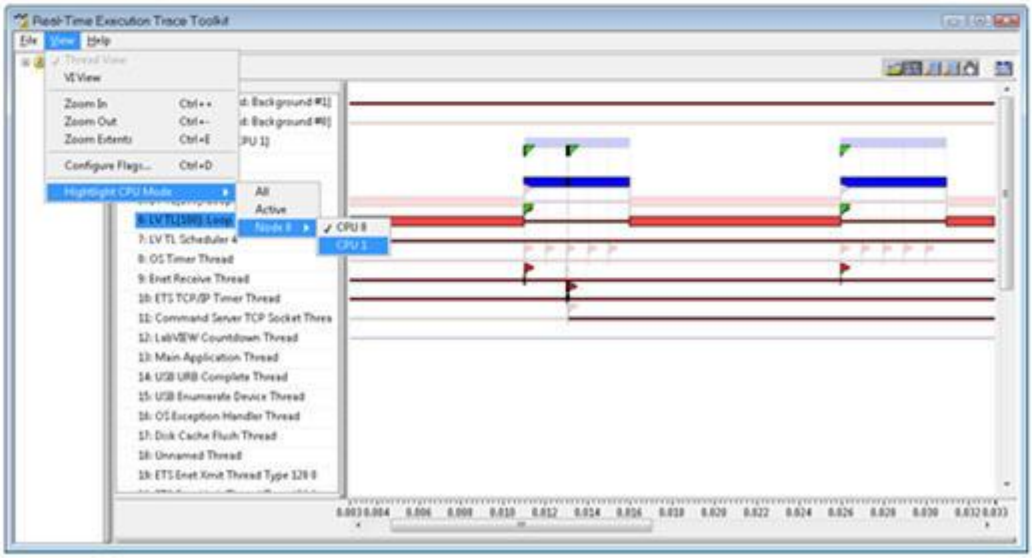
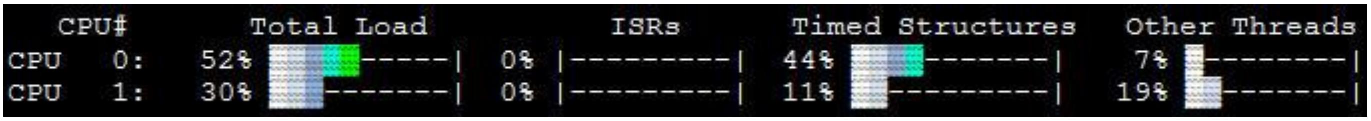


Figure 4 - The Real-Time Execution Trace Toolkit 2.0 with support for Multicore Debugging

Another utility that allows for inspection of CPU utilization on multicore real-time systems, is the On-Screen CPU Monitor in LabVIEW 8.5 Real-Time. This utility displays information directly to a display connected to the real-time target, with information such as Total Load, ISRs, Timed Structure utilization, and other thread utilization as shown below.



```
  CPU#        Total Load          ISRs      Timed Structures    Other Threads
CPU   0:    52% ▒▒▒▒-----|  0% |---------|  44% ▒▒▒▒-------|    7% ▒--------|
CPU   1:    30% ▒▒▒-------|  0% |---------|  11% ▒▒---------|   19% ▒▒-------|
```

Figure 5 - On-Screen CPU Monitor

Evaluating Multicore Readiness in Real-Time Systems

Companies migrating real-time software to a multicore processor experience varying levels of scalability, and should look at each layer of the stack for readiness. The real-time software stack consists of development tools, libraries, device drivers, and a real-time operating system. Many real-time applications from previous generation systems were built on a software stack intended for single-processor designs. Table 1 lists a few key considerations to help evaluate the readiness of the real-time software stack found in a typical control application.

Real-Time Software Stack	*What It Means to Be Multicore Ready*
Development Tool	Support is provided on RTOS, tool allows for threading correctness and optimization. Debugging and tracing capabilities are provided to analyze real-time multicore systems.
Libraries	Libraries are thread-safe and can be made re-entrant so they may be executed in parallel. Algorithms are in place so as to not cause memory allocation and induce jitter into system.
Device Drivers	Drivers are designed for optimal multithreaded performance.
Real-Time Operating System	RTOS supports multithreading and multitasking, and can load balance tasks on multicore processors with SMP.

Table 1 - Multicore Ready Real-Time Software Stack

A common bottleneck in real-time systems are device drivers that are not thread-safe or libraries that cannot be executed in a re-entrant fashion -- code may still function properly but it cannot executed in parallel on a multicore processor, so performance gains cannot be maximized. For LabVIEW developers, these pitfalls are overcome with thread-safe and re-entrant I/O drivers (such as DAQmx) as well as math libraries that can be executed in a re-entrant fashion. The LabVIEW Real-Time software stack meets all the requirements for multicore readiness.

LabVIEW Real-Time SMP Case-Study: Wineman Technologies Inc.

Wineman is a National Instruments partner that develops dynamometer control systems as well as hardware-in-the-loop simulators, utilizing PID control, profile generation, alarming, scripting, analog I/O, digital I/O, CAN, Modbus and TCP/IP communication. Their family of products including ***EASE*** and ***INERTIA*** applications all utilize an architecture based on LabVIEW Real-Time.

Wineman was challenged with how to overcome high CPU utilization. The high priority loop in one of their typical applications consumed the majority of the single core processor in order to meet the execution requirements. This left little time for the lower-priority loops to execute tasks such as user-interface communication and data logging. The result is that the applications were generally unusable and/or unstable due to the task starvation required to meet the specified control loop execution rates.

Using the SMP support in LabVIEW 8.5 Real-Time, Wineman was able to add more control loops to their existing system and enhance the performance by 40% with only minor changes to the LabVIEW code.

> *With LabVIEW Real-Time 8.5, we* ***increased our maximum loop rate by 40%*** *by utilizing both processor cores in our dual-core PXI real-time controller…"*

Support for multicore processors in LabVIEW Real-Time significantly improved the overall performance of Wineman's products. Most notably was the ability to take advantage of the multicore support without any significant changes to the code. By using the inherently parallel nature of LabVIEW, Wineman's application naturally migrated to the multi-core environment in a way that would take advantage of the additional processing power. This meant that Wineman is able to add functionality in their dynamometer control and hardware-in-the-loop simulators and meet the needs of their end-users.

Conclusions

LabVIEW 8.5 Real-Time opens the door for Real-Time developers to take advantage of higher-performance multicore processors in their real-time applications. In addition to support for SMP, LabVIEW 8.5 introduces enhanced debugging tools such as the Execution Trace Toolkit 2.0 to visually analyze traces executing on a multicore system.

Finally, as a recommendation for evaluating readiness for migrating code to a multicore system (regardless of programming language or tool of choice), it's a good idea for developers to consider their entire real-time software stack. The multicore-ready LabVIEW Real-Time software stack allows companies like Wineman to see immediate performance gains when migrating to multicore systems, with little programming modifications to the original code.

Debugging Multicore Applications with the Real-Time Execution Trace Toolkit

Debugging is the most time-consuming phase of software development, according to 56 percent of survey respondents at Embedded Systems Conference 2007, in a survey conducted by Virtutech. In addition, debugging is made more difficult as systems become more complex with more processors in a single design. Currently, the software industry has a gap in support for multicore debugging. 59 percent of respondents from the same survey mentioned above said the debugging tool they use does not support multicore or multiprocessor development. To help fill this gap, National Instruments introduces support for multicore debugging with the Real-Time Execution Trace Toolkit 2.0. This document provides an overview of the Real-Time Execution Trace Toolkit with examples of how to detect problem areas in code.

Overview of the Real-Time Execution Trace Toolkit

The Real-Time Execution Trace Toolkit provides a visual way to debug real-time applications by showing thread activity from traces on the real-time operating system (RTOS). Common use-cases of the tool include the following:

- Identifying shared resources and memory allocation
- Verifying expected timing behavior
- Monitoring CPU utilization and multicore interaction
- Learning about the LabVIEW execution model

The Real-Time Execution Trace Toolkit is an add-on for LabVIEW 8.5 Real-Time and also backward compatible with versions of LabVIEW Real-Time 7.1.1 or above.

There are two main components of the trace toolkit: Instrumentation VIs and the Trace Viewing Utility.

The Instrumentation VIs need to be added around the code whose execution you would like to trace.

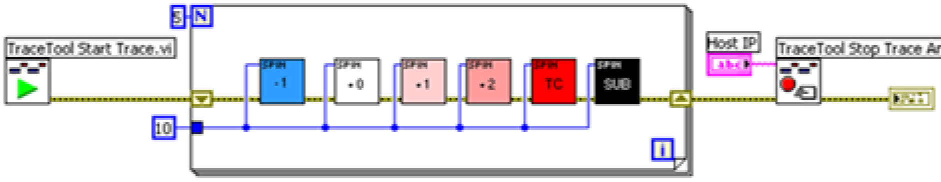


Figure 1. *Instrumenting a LabVIEW Real-Time with Start and Stop Trace Tool VIs for Debugging*

Events from the code are then logged to a memory buffer, and passed back to the host machine for viewing.

The Trace Viewing Utility is used to view the captured execution trace. Within the Trace Viewing Utility are separate views to look at a listing of all traces, a Thread View, and a VI View of the current trace.

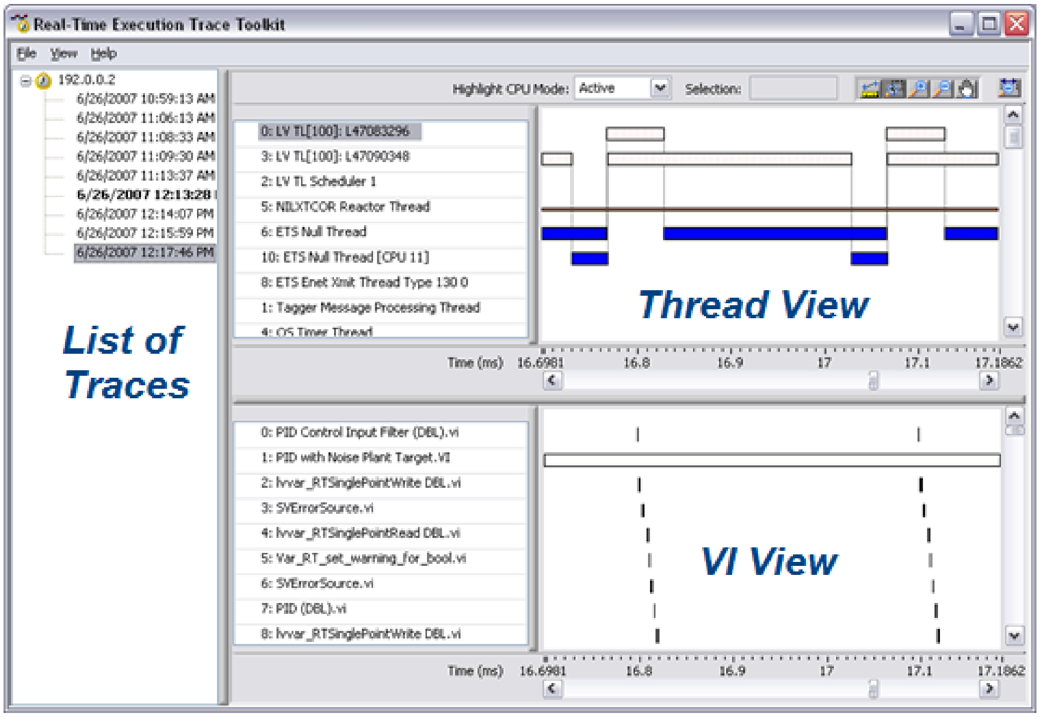
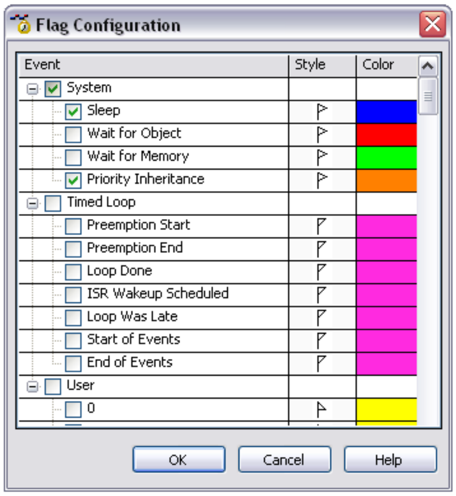

Real-Time Execution Trace Toolkit
File View Help
192.0.0.2
6/26/2007 10:59:13 AM
6/26/2007 11:06:13 AM
6/26/2007 11:08:33 AM
6/26/2007 11:09:30 AM
6/26/2007 11:13:37 AM
6/26/2007 12:13:28
6/26/2007 12:14:07 PM
6/26/2007 12:15:59 PM
6/26/2007 12:17:46 PM
List of Traces
Highlight CPU Mode: Active
Selection:
0: LV TL[100]: L47083296
3: LV TL[100]: L47090348
2: LV TL Scheduler 1
5: NILXTCOR Reactor Thread
6: ETS Null Thread
10: ETS Null Thread [CPU 11]
8: ETS Enet Xmit Thread Type 130 0
1: Tagger Message Processing Thread
4: OS Timer Thread
Thread View
Time (ms) 16.6981 16.8 16.9 17 17.1 17.1862
0: PID Control Input Filter (DBL).vi
1: PID with Noise Plant Target.VI
2: lvvar_RTSinglePointWrite DBL.vi
3: SVErrorSource.vi
4: lvvar_RTSinglePointRead DBL.vi
5: Var_RT_set_warning_for_bool.vi
6: SVErrorSource.vi
7: PID (DBL).vi
8: lvvar_RTSinglePointWrite DBL.vi
VI View
Time (ms) 16.6981 16.8 16.9 17 17.1 17.1862

Figure 2. *Three main components of the Trace Viewing Utility: Listing traces, Thread View, and VI View*

The bars in the viewing utility represent processing activity on the real-time target, and the traces can be zoomed in for more precise viewing or to measure the execution time of a thread or VI. In addition, Flags can be configured to observe specific events:

- System Events: Sleep, Wait, Priority Inheritance
- Timed Loop Events: Preemption Start/Stop, Done, ISR Wakeup
- User-Defined Events



Event	Style	Color
⊟ ☑ System		
☑ Sleep	⚐	🟦
☐ Wait for Object	⚐	🟥
☐ Wait for Memory	⚐	🟩
☑ Priority Inheritance	⚐	🟧
⊟ ☐ Timed Loop		
☐ Preemption Start	⚐	🟪
☐ Preemption End	⚐	🟪
☐ Loop Done	⚐	🟪
☐ ISR Wakeup Scheduled	⚐	🟪
☐ Loop Was Late	⚐	🟪
☐ Start of Events	⚐	🟪
☐ End of Events	⚐	🟪
⊟ ☐ User		
☐ 0	⚐	🟨

Figure 3. *Flag Configuration Dialog*

What to Look for when Debugging

The most common use-case for the Trace Tool is debugging code to detect unexpected behavior such as shared resources. Shared resources can introduce jitter into your real-time application, so it's important to understand how to detect this potential problem. Table 1 shows a few examples shared resources and the potential problems that may appear in an application.

Shared Resources	Potential Problems
• LabVIEW Memory Manager • Non-reentrant shared subVIs • Global variables • File system	• Priority Inversion • Ruined determinism

Table 1. *Shared resources and related potential problems.*

One such shared resource is the LabVIEW Memory Manager. It is responsible for dynamically allocating memory. When a normal priority program is in possession of the Memory Manager, the rest of the threads, including the time-critical thread, must wait for the shared resource to become available. In such cases jitter is inevitably introduced in the time-critical thread. To resolve this issue, the thread scheduler temporary boosts the normal-priority application to run in the time-critical thread so that it can finish up quicker and release the Memory Manager. This phenomenon is known as priority inheritance or priority inversion. To avoid such situations National Instruments advises to avoid using shared resources. A solution in the case of the Memory Manager would be to preallocate memory for arrays.

Figure 4 is the trace of a program which has a normal priority subVI and a time-critical subVI sharing the Memory Manager. The green flags show dynamic memory allocation, and the orange flag shows priority inheritance. The execution of the time-critical thread was interrupted so that the normal priority subVI can be boosted up to run in the time-critical thread, and therefore release the shared resource quicker. Because the time-critical thread was interrupted, this shared resource can harm the determinism of the time-critical subVI.

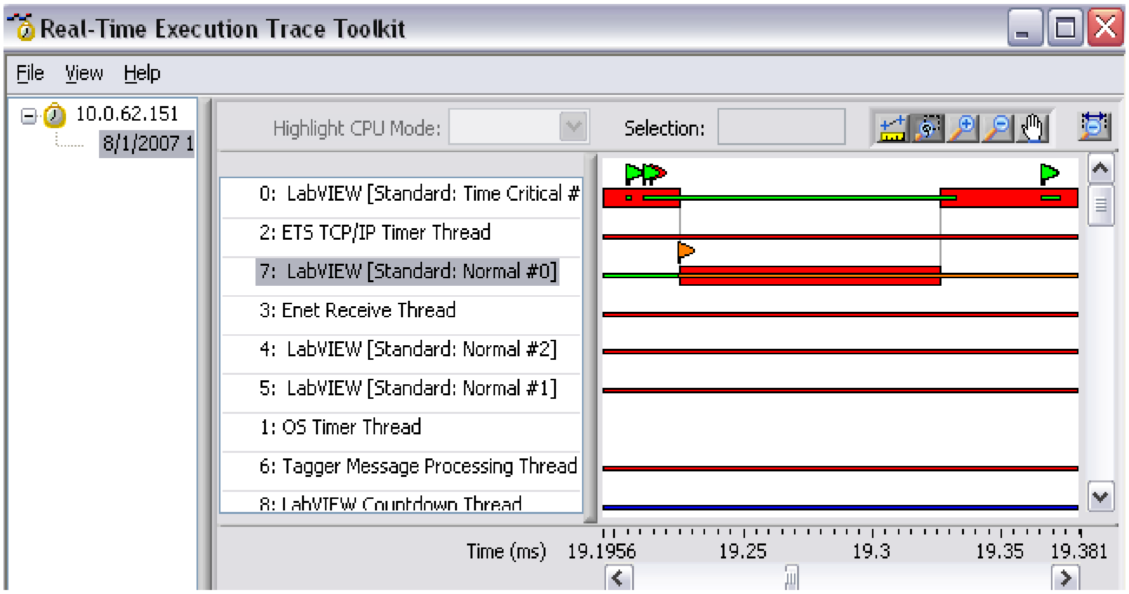

Real-Time Execution Trace Toolkit
File
View
Help
10.0.62.151
8/1/2007 1
Highlight CPU Mode:
Selection:
0: LabVIEW [Standard: Time Critical #
2: ETS TCP/IP Timer Thread
7: LabVIEW [Standard: Normal #0]
3: Enet Receive Thread
4: LabVIEW [Standard: Normal #2]
5: LabVIEW [Standard: Normal #1]
1: OS Timer Thread
6: Tagger Message Processing Thread
8: LabVIEW Countdown Thread
Time (ms)
19.1956
19.25
19.3
19.35
19.381

Figure 4. *The green flags show memory allocation, in other words, accessing the Memory Manager. The orange flag shows priority inheritance.*

Priority inheritance can also occur when mixing two priority scheduling schemes – at the VI level and at the Timed Loop level. For example, in Figure 5 we can see a time-critical subVI (red icon) and a Timed Loop fighting for the same memory resources. The race for shared resources between the time-critical subVI and the Timed Loop leads to priority inheritance.

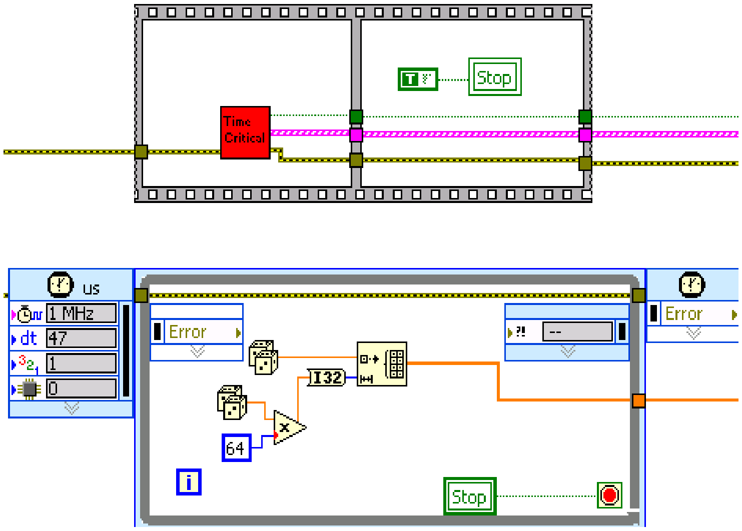


__Figure 5.__ Example of using two different priority assignment schemes.

In Figure 6 we can see that the Timed Loop started running first, then it was preempted by the time-critical subVI. The purple flags show the start and the yellow flags show the end of the Timed Loop preemption. At the end of the preemption, the timed loop inherited priority from the time-critical subVI. This is noted on the execution trace by the orange flag. To reduce such conflicts, try not to use shared resources in time-critical threads.

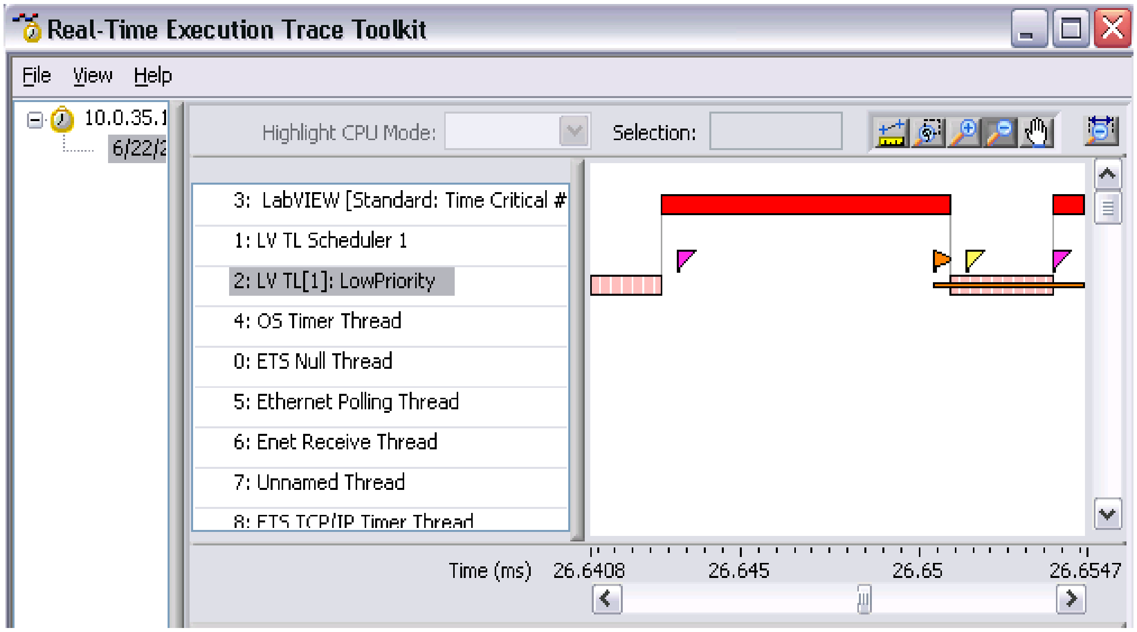


__Figure 6.__ Timed Loop preemption and priority inheritance

It’s recommended to use only one priority assignment scheme – either subVIs or Timed Loops of different priority levels.

Debugging Multicore Applications

When a trace is logged on a multicore real-time target, the Viewing Utility will have the option to select a “Highlight CPU Mode”.

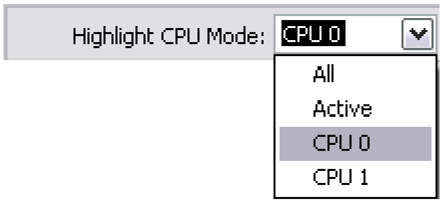
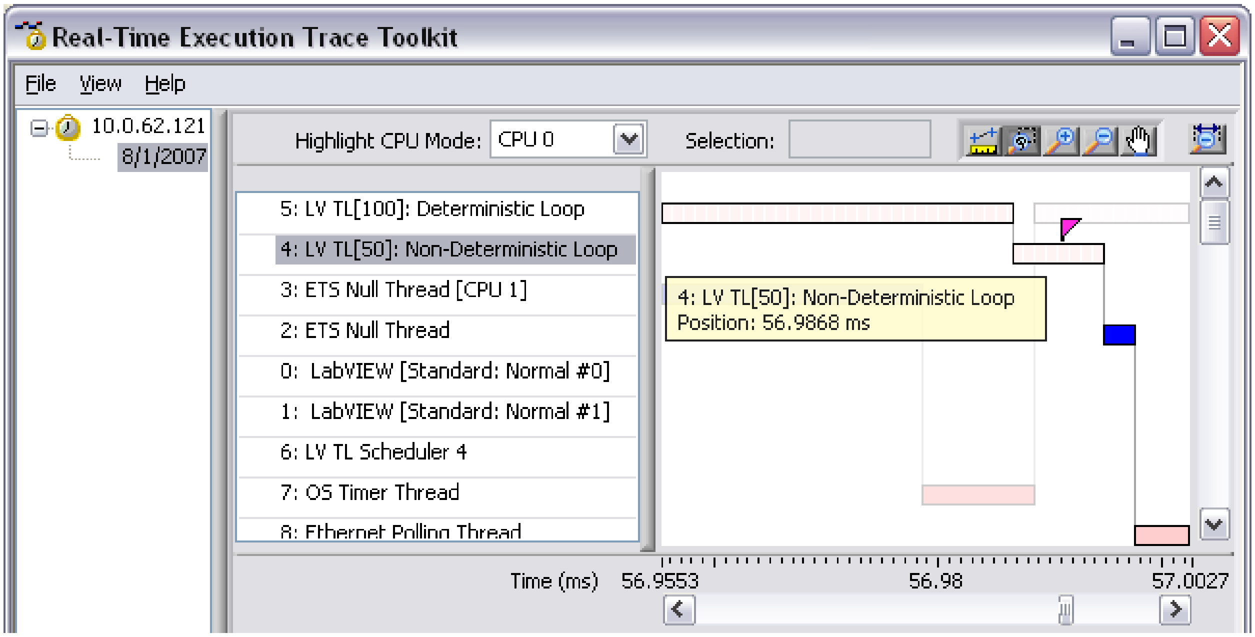

Highlight CPU Mode: CPU 0

__Figure 7.__ Selecting Highlight CPU Mode

The following trace shows a multicore application with two tasks called “Deterministic Loop” with a priority of 100, and “Non-Deterministic Loop” with a priority of 50.   Notice how the Highlight CPU Mode is selected to “CPU 0”, so the threads that are highlighted are all running on that same CPU. The threads running on CPU 1 are dimmed.



Highlight CPU Mode: CPU 0

5: LV TL[100]: Deterministic Loop
4: LV TL[50]: Non-Deterministic Loop
3: ETS Null Thread [CPU 1]
2: ETS Null Thread
0: LabVIEW [Standard: Normal #0]
1: LabVIEW [Standard: Normal #1]
6: LV TL Scheduler 4
7: OS Timer Thread
8: Ethernet Polling Thread

4: LV TL[50]: Non-Deterministic Loop
Position: 56.9868 ms

Figure 8. *Trace of a Multicore Application*

A common item to look for in a trace from a multicore application are the threads switching between processors.   The automatic scheduler will do it’s best to load-balance tasks across the available cores, but sometimes there are inefficiencies that must be manually controlled by the developer. The Processor Affinity assignment in the Timed Loop allows developers to assign a Timed Loop to a specific processor core.

Conclusions

The Real-Time Execution Trace Toolkit provides a visual way to debug real-time applications by showing thread activity from traces on the real-time operating system (RTOS). This whitepaper considered common use-cases of the tool , particularly the detection of shared resources. It’s important to use other standard debugging tools along with the Real-Time Execution Trace Toolkit, such as: Probes, breakpoints, execution highlighting, NI Timestamp VIs (RDTSC), RT Debug String VI, and User Code.

Key features of version 2.0 of the Real-Time Execution Trace Toolkit is the added support for more flag configurations (such as priority inheritance) and the ability to debug multicore real-time applications.