Programming Approach with Parallel Capable Instruments for Test Efficiency

Tan Van, Rodney Singleton
Assembly Test Division
Teradyne, Inc.
North Reading, MA, U. S. A.

Abstract— Throughput demands and cost-down pressures involved with modern functional testing necessitate a trend towards a parallel test paradigm.

Parallel test techniques can be employed by test engineers to overcome these challenges. The benefits of conducting tests on multiple units/subsystems concurrently are to maximize instrumentation utilization, increase throughput and reduce execution time.

This paper will outline a strategy for implementing parallel tests and provide some practical program development guidance and techniques which will allow test program developers to take full advantage of parallel capable systems.

Keywords-Parallel testing; Test efficiency; Analog test instrumentation

I. INTRODUCTION

Traditionally, test systems were set up to test one unit under test (UUT) at a time. As a result, test programs were written to program a single instrument in sequential steps from initializing the instrument to analyzing the captured data. This sequential strategy is so familiar to test engineers that it is even used on test systems with parallel capability.

With advancement in technology of integrated circuits and new instrument design, test instruments today provide parallel test capability. Multiple, independent source and measurement channels are now available. While test instruments have been revolutionizing, instrument drivers have not, especially IVI drivers. IVI drivers only support single channel programming. For example, the IviFgen_ConfigureOperationMode [1] function configures operation mode on one channel at a time on a multi-channel function generator. Since parallel test programs need to set up multiple stimulus and measurement channels, handling this task with IVI drivers is a big challenge.

This paper explores a software infrastructure design that helps overcome instrument driver limitations when developing parallel test programs. The design also discusses the ability to facilitate discrete channel debug.

II. DEFINITION AND DISCUSSION

A. IVI Drivers Programming Model

With the current IVI software architecture, IVI drivers only support a single instrument and single channel programming. Each instrument must be initialized one at a time with its resource name. After the instrument is initialized, each channel is programmed by passing the channel information along with hardware settings to the instrument driver.

Using this programming model on a parallel capable test system would require test programs to repeat every driver call for all channels on all instruments. As a result, programming multiple channels to generate stimulus signal or make measurement simultaneously is complex and lengthy.

Shown below is pseudocode for setting up multiple channels on a digitizer to perform concurrent measurements:

For Dig_Channel = 1 : n

    Configure:
    Frequency,
    Voltage Range,
    ...

End

This programming model increases the development time of new test and creates program logic that is more difficult to follow which leads to maintainability issues.

B. Multi-Channel Software Layer

As previously shown, using an instrument driver directly is unfavorable. Therefore, test infrastructure should include the Multi-Channel Software Layer (MCSL) to aid with multi-channel programming.

MCSL is a software layer consisting of software components that provide a simplified Application Programming Interface (API) [3]. These components enable users to generate parallel test programs when only single channel drivers are available. Each software component in the MCSL is a wrapper for an instrument driver, which controls an instrument that is capable of performing actions in parallel. It defines a new set of functions and properties that encapsulate the complexity of setting up multiple channels for testing.
As shown in Fig. 1, test programs control test instrumentation by using multi-channel components. Direct access to the respective instrument driver is not needed. For tests that require an increased amount of control, direct driver access can be utilized.

The most significant advantage of the MCSL is that test programmers do not have to manage the complexity of setting up hardware on multiple channels. This is all dealt with by multi-channel components. Another benefit to this design is reusability. Once a multi-channel component is created, it can then be reused by all future test programs. As a result, program development time is reduced. This structure also decouples test programs from the instrument drivers. If a change occurs in the instrument driver, only the components in the MCSL need to be changed. While this design simplifies the development of parallel test programs, it still allows user accessibility to the instrument drivers as illustrated in Fig. 1.

Figure 1: Test Program Structure

III. DESIGN MULTI-CHANNEL API

A typical test program consists of four major tasks: setting up the hardware; acquiring the data; analysis of data; and logging of results. Of these four tasks, setting up the hardware and acquiring the data take up most of the execution time.

The main features when creating a multi-channel API are:
- Initialize instrument
- Set up hardware
- Initiate measurement
- Wait for acquisition complete
- Fetch result

The implementation considerations for the first two features, instrument initialization and hardware setup, apply to both stimulus and measurement instruments. The remaining features, measurement initiation, acquisition, waiting, and fetching the results, are only applicable to measurement instruments.

A. Initialize Instrument

Test instruments used to occupy one system slot per instrument. For instance, the digitizer takes one slot, while the function generator takes another slot. Automated Test Instrument design has revolutionized such that multiple instruments are combined into one slot [2]. Some instrument configurations consist of different types of instruments. While others may consist of multiple, same type instruments, such as two digitizers or two function generators. The IVI drivers, however, allow only one instrument to be initialized at a time. With a test system that has two or more same type instruments, this would require the test program to make multiple calls to the instrument driver init function for each unit. To simplify this step, the multi-channel API should provide an initialize function that allows a list of resource names to be passed in as a parameter. As shown in the following example, this function uses IVI-C to initialize multiple function generators:

```c
void Init(ViRsrc resourceName[], ViSession sessionID[], int instrCount)
{
    for(int i=0;i<instrCount;i++)
    {
        IviFgen_init(resourceName[i],false,false,&sessionID[i]);
    }
}
```

Following execution, the Init function above returns a list of instrument handles for the specified instruments.

B. Set up Hardware

A typical instrument driver interface provides a large number of methods and properties for programming hardware. There are methods that set multiple parameters which belong to a subsystem. For example, it is possible for a user to program the sample rate and sample size in a single function call. However, there are a good number of parameters which can only be programmed individually, whether by using a method or property. As a result, setting up hardware for a channel may take many lines of code. If this code is repeated often in every test program then it not only affects development time but also causes maintenance issues.

This problem can be resolved by implementing the multi-channel setup function such that it provides the client application the ability to set up multiple subsystems, from the input path to the back-end digitizer, in a single function call. With this implementation, any changes in instrument drivers can be updated in only one location.

C. Initiate Acquisition

Instrument driver for a measurement instrument may provide different methods for initiating a measurement cycle. Generally, these functions have two different behaviors. In the first behavior, the driver initiates the acquisition and returns
program control back to the calling function immediately. In the second behavior, the driver initiates the measurement and then waits for the acquisition to finish before returning control. Using the IViDmm class [4], for example, the Read function waits until the acquisition completes while the Initiate function does not.

In a parallel test program, all measure channels need to be initiated so that they can start acquiring data simultaneously when triggered. Therefore, when implementing a multi-channel initiate acquisition function, it is important to use the appropriate driver function, namely, a function that returns immediately after initiating measurement so that the other channels can then be initiated.

D. Waiting for Acquisition Complete

After a measurement is initiated, the test program needs to know when data is ready for fetching on all enabled channels. To allow code reuse and to simplify this task, the multi-channel component should provide a method that returns the acquisition status on all enabled channels.

When querying for acquisition status using instrument driver, it is important to set some delay time between queries so that it does not use up the CPU time while waiting for the measurement. This allows the CPU to process other requests while test hardware is still making measurement.

E. Fetch Result

Most test cases need only one measurement. However, there are test cases that require multiple measurements before the data can be analyzed. Fetching data should be functionally partitioned from the analysis activity. With this separation, test complexity is reduced because functional overlap does not occur. This also improves maintainability, reusability, and unit testing.

Implementation of a fetching function should consider simplifying reading back results. Measured data contain many different characteristics such as voltage level, frequency, duty cycle, etc. Each of these values is read back one at a time when using the instrument driver directly. These driver calls can be wrapped in the multi-channel fetch function so that test programs will get all the required results with one function call.

F. Support Debug Capability

All test programs are required to provide debug capability for users to troubleshoot hardware failures. When debugging a failure, the user needs the ability to program the failing channel or instrument to a specific hardware state so that the failure can be recreated and diagnosed. This level of debug capability is still considered an essential requirement for any test program developed with multi-channel components.

When designing a multi-channel component, whether it is for a function generator or a measurement instrument, each subroutine of the multi-channel API must be defined such that a list of channels, along with hardware settings associated with each channel, can be passed in as a parameter. This will allow test programs to execute in parallel during a normal run while still enabling users to debug a test by programming only a single channel.

For illustration purpose, below is the Configure function definition using IVI-C. The first parameter, settings, is an array which contains the hardware settings of the channels to be programmed. The second parameter, channelCount, specifies the number of channels.

```c
void Configure(FgenSettings settings[], ViInt32 channelCount);
```

where FgenSettings is defined as:

```c
struct FgenSettings{
    ViSession SessionID;
    ViInt32 Channel;
    ViReal64 Amplitude;
    ...
};
```

The actual function definition may be different for different programming languages. However, they all serve the same purpose, that is to give test programs the flexibility to program any number of channels.

IV. Test Program Implementation

To demonstrate the similarity between two implementations, sequential and parallel, and to show the advantage of parallel programming, a test program was created to measure the frequency of two input signals with a multi-channel timer counter instrument.

A. System Configuration

The system configuration consists of the following equipment:

- 13-slot VXI chassis
- Slot 0, MXI controller
- 8-channel Analog Test Instrument.
- 2-channel function generator (UUT)

As shown in Fig. 2, each UUT channel is connected directly to a Timer Counter channel using a coax cable.
Figure 2: Test configuration with a multifunction analog instrument and a function generator.

**B. Implementation: Sequential vs Parallel**

As shown in Fig. 3 and Fig. 4, the steps in the two implementations are similar. In the sequential implementation, the test program accesses the instrument drivers directly. On the other hand, the parallel implementation uses the multi-channel components.

Comparing the two methods of implementation shows that parallel testing employing multi-channel components does not add additional complexity to the test program.

```plaintext
Step 1: Initialize instruments
MultiChannel_Source.Init(...);
MultiChannel_TimerCounter.Init(...);

Step 2: Program stimuli channels
MultiChannel_Source.Configure(...);

Step 3: Program measure channels
MultiChannel_TimerCounter.Configure(...);

Step 4: Start stimuli signals
MultiChannel_Source.InitiateGeneration(...);

Step 5: Initiate measurement
MultiChannel_TimerCounter.InitiateAcquisition(...);

Step 6: Check acquisition status
do
do
MultiChannel_TimerCounter.AcquisitionStatus(...);
while not complete

Step 7: Fetch data
MultiChannel_TimerCounter.FetchData(...);

For Test_Channel = 1: n

Step 2: Program stimulus channel
IviFgen_ConfigureOutputMode(...);
IviFgen_ConfigureStandardWaveform(...);
...

Step 3: Program measure channel
terAiChannel_ChannelsItemConfigure(...);
terAiChannel_ConfigureTimerCounterMeasurement();
...

Step 4: Start stimulus signal
IviFgen_InitiateGeneration(...);

Step 5: Initiate measurement
terAiChannel_InitiateTimerCounterMeasurement(...);

Step 6: Check acquisition status
do
derAiChannel_FetchTimerCounterMeasurementStatus(...);
while not complete

Step 7: Fetch data
derAiChannel_FetchTimerCounterMeasurement(...);

Next Test_Channel
```

Figure 3: Parallel test using multi-channel API

C. **Results**

As shown in Fig. 5, the execution time in sequential implementation grows linearly as the number of channels increases. Conversely, the execution time remains relatively unchanged in the parallel implementation.

Figure 4: Sequential testing using instrument driver directly

Figure 5: Execution time comparison
The efficiency achieved with parallel test varies depending on the test signal and type of test. The efficiency increases significantly for tests with longer acquisition time.

V. CONCLUSION

Test programs are required to reduce runtime without increasing budget on test instrumentation. This demand has led to the transition from sequential to parallel test. While test instruments have advanced to provide parallel capability, instrument drivers still support single channel programming. This is a challenge for creating parallel test programs. This paper has discussed the approach to address the instrument driver limitation. The following was discussed:

- Design of the multi-channel components which provide simplified API for parallel test programming.
- Implementation considerations of the subroutines that enable multiple channels to perform testing simultaneously with ability to debug on individual channels.
- A comparison between sequential programming using instrument directly and parallel programming using multi-channel components.

Whether or not IVI drivers or instrument specific drivers are used for creating parallel test program, the design techniques explored in this paper can be applied on different types of instrument drivers which lack the support for parallel testing.

ACKNOWLEDGMENT

The authors would like to thank the following people for their support and contributions: Steven Conte, Tarra Marchetti, and Aaron Schmaelzle.

REFERENCES