An Object-Oriented Software Framework for Execution of Real-Time, Parallel Algorithms*

J. Brent Spears and Brett N. Gossage

ERC, Inc.

555 Sparkman Drive, Executive Plaza, Suite 1622 Huntsville, AL 35816
{bspears, bgossage}@rttc.army.mil

Abstract. Infrared (IR) scene projection systems that employ micro-resistors can have significant, non-uniform spatial variation in their output. This non-uniformity causes unwanted artifacts in the projected scene sufficient to prevent accurate sensor testing. To compensate for this non-uniformity, high-speed digital hardware is used to apply non-uniformity correction (NUC) to the images. However, the hardware is very costly and must be custom-built for each projector. With high performance computers, NUC can be implemented in software at a fraction of the hardware cost while still meeting the real-time requirements. The purpose of this paper is to present object-oriented frameworks, implemented in C++, for executing NUC algorithms in parallel in real-time. The frameworks provide abstractions for multi-threading, parallel processing, shared memory, frame-based scheduling, variable interrupt sources, and scheduling disciplines. Results for NUC algorithms on an 8-processor SGI Onyx-2 will also be presented.

1 Introduction

To avoid costly field-testing, infrared (IR) sensors are tested by projecting an infrared scene into its optical aperture and running the system within a real-time simulation. The scene is generated by a micro resistor array that converts image pixel values into thermal energy. A difficulty in this conversion process arises from the fact that there can be a significant variation in the electrical resistance values for all of the micro-resistors. This *non-uniformity* causes unwanted artifacts in the projected scene. For the scene projector to be a useable device, a provision for non-uniformity correction (NUC) must be implemented. The current solution to the non-uniformity problem is to selectively adjust the intensity value of the input image on a resistor by resistor basis. This must be done for each pixel in the image at video frame rates. The present implementation of NUC relies on costly high-speed digital hardware. The goal of this effort was to develop a new class of scalable and reusable NUC algorithms which will perform real-time

* Funding for this project was provided by the Common High-Performance Software Support Initiative (CHSSI) in cooperation with the Redstone Technical Test Center (RTTC) Huntsville, AL.

NUC for scene projectors using generic, parallel computing resources. This paper describes object-oriented frameworks for applying NUC algorithms in real-time. We first present the general design of the system. Next, we will describe the libraries and classes used in the design, particularly the two main components - the real-time frame scheduler and the NUC algorithms. Finally, we present results from testing the algorithms on a multi-processor computer.

2 General Design

There are three main processes in the system - a scene generator, NUC applicator, and a display process. All three processes operate in parallel using a ring buffer with three images (see Figure 1). This creates a pipeline in which the scene generator provides an image, the NUC process then applies the algorithms to the image, and the display process takes the resulting image and displays it to the screen. The NUC process is further divided into threads that are responsible for a slice of the image. Each thread is run exclusively and without interruption on separate processors. A drawback to this design is a two-frame latency (typical for any pipeline). All three processes are synchronized to a particular frame rate (in this case, the vertical refresh rate).

Fig. 1. Processes and Ring-buffer

3 Libraries

The real-time NUC (RNUC) software is based on a core set of reusable libraries. The C++ utility library and the inter-process communication and control (IPCC) library are the core libraries reused by the real-time software system.

3.1 C++ Utility Library

The C++ utility library contains a variety of utility classes such as arrays and strings. One of the more important classes used in the real-time system is the ring buffer adapter. This class is an standard template library (STL¹) adapter which provides circular iterators (i.e., a ring). In the real-time software system, the ring buffer allows multiple images to be operated on in parallel. Other important features used by the system are the `MemoryPool` class and the `pool_alloc` template class. The `MemoryPool` class serves as an interface for a pool of memory (could be shared or not). It contains member functions such as `allocate` and `deallocate`. The `pool_alloc` template class is the allocator for a `MemoryPool`. The allocator was designed and modeled after the STL allocator in order to make it compatible with the STL. This allows STL constructs (such as lists, queues, and vectors) to be used in a `MemoryPool` (possibly shared memory) by instantiating them with this allocator. The allocator can also be used with other classes that use an STL-compatible allocator.

3.2 Inter-process Communication and Control (IPCC) Framework

The IPCC is a cross-platform library that provides multi-threading and inter-process communication support. The platform independence is achieved by using the *bridge pattern* [GE1]². The bridge pattern separates the interface from the implementation. This allows the programmer to code using the interface without worrying about the implementation. This pattern not only supports platform independence, but also allows different types of implementation for the same platform. Several basic inter-process communication constructs are used in the real-time software system. These include threads, mutexes, semaphores, and condition variables. The key capability used by the real-time system is the shared arena.

Shared Arena The `SharedArena` class provides platform-independent support for inter-process communication via shared memory. Platform-specific shared memory capabilities are provided by implementation classes as part of a *bridge pattern* [GE1]. A `SharedArena` supports dynamic memory allocation/deallocation and a single, process-shared user data pointer. The interface semantics of a `SharedArena` are similar to the memory-mapped file interfaces provided by POSIX. Since `SharedArena` is a subclass of `MemoryPool`, it can be combined with the `pool_alloc` template class (effectively creating a shared memory allocator) to support shared C++ objects and collections.

¹ The STL has been incorporated into the C++ Standard Library, but we will use "STL" for the sake of brevity.

² A variation of the pattern was used in cases where the constructs are allocated in shared memory.

4 Real-Time Frame Scheduler (Rt) Framework

Rt is a portable real-time frame scheduler library. It defines an interface that permits execution of threads on multiple processors at a specified frame rate. The *bridge pattern*[GE1] is used to separate the interface from the implementation. Therefore, the behavior of Rt can be modified with a new implementation. For example, we have developed a simulated real-time implementation based on the *pthread*s library. This implementation is useful when testing algorithms for correctness during development. The class diagram for Rt is shown in Figure 2.

4.1 Frame Schedulers

The Rt framework interface is based heavily on SGI's REACT/Pro [GE1] interface (our first implementation layer was based on REACT/Pro). A frame scheduler is created and assigned to control a unique processor. The scheduler will exclusively control the processor by restricting and isolating the processor so that only threads assigned to that scheduler will be permitted to execute on that processor. There is always one master frame scheduler and an optional number of slave frame schedulers (slaves are needed when using more than one CPU).

4.2 Interrupt Source

The master frame scheduler manages the frame rate for itself and any slave frame schedulers. The frame rate is defined by an interrupt source. Interrupt source availability is platform dependent. Rt supports several different types of interrupt sources, but all may not be valid on a given platform (availability may be determined at runtime with function calls). An example of an interrupt source is a real-time clock timer or an interrupt from some hardware device. The interrupt source defines the length of a minor frame. One or more minor frames define a major frame (the frame rate). During each minor frame, some work is allowed to be performed by the scheduler.

4.3 Threads

Threads accomplish the work to be done during a minor frame. The programmer simply derives his thread class from `RtThread` and defines his work in the `RtThread::work()` member function. All interaction between a thread and a frame scheduler occurs in the `RtThread::process()` listed below.

```
void RtThread::process()
{
// Wait for frame scheduler to enqueue the thread...
    m_enqueuedSemaphore.wait();

// Join the frame scheduler...
    m_scheduler->joinScheduler();
}
```

```
// Main loop...
while( !m_finished )
{
    // Work function hook...
    this->work();

    // Yield the cpu and suspend...
    m_scheduler->yield();

} // end while

} // end process
```

Threads are enqueued to minor frames of a scheduler and are executed during that particular minor frame. Multiple threads may be enqueued to the same minor frame and they are executed in the ordered enqueued. A thread must complete before the end of the minor frame it was enqueued to. If not, an **over-run** is said to have occurred. And if a thread is never started³ during its minor frame, an **underrun** is said to have occurred.

4.4 Scheduling Disciplines

By default, Rt will report overruns and underruns by calling an error handler installed by the programmer (or a default one which does nothing). The programmer can control whether overruns and underruns are reported on a per-thread basis using the scheduling disciplines described below.

Strict real-time Thread must be started and completed during a minor frame.

Overrunnable Thread is allowed to overrun.

Underrunnable Thread is allowed to underrun.

Continuable Allows a thread to be started just once and execute throughout a major frame.

Background Thread is started only after all other threads have completed.

4.5 Creating Schedulers

The basic steps for creating a frame scheduler are as follows:

- create slave schedulers
- create master frame scheduler
- enqueue worker threads to minor frames
- start master scheduler
- wait for worker threads to complete

³ A thread can underrun if the threads enqueued before this one use the entire minor frame

The following code demonstrates the steps for creating and using the Rt frame schedulers:

```
// Create slaves...
    RtSlaveFrameScheduler slave1( 2 );    // cpu 2
    RtSlaveFrameScheduler slave2( 3 );    // cpu 3

// Add to slave set...
    RtSlaveSchedSet slaveSet;
    slaveSet.add( &slave1 );
    slaveSet.add( &slave2 );

// Create interrupt source (50 milliseconds)...
    RtIntrSource intrSource( rt_intr_cctimer, 50000 );

// Create master scheduler on cpu 1 with 1 minor frame...
    RtMasterFrameScheduler master( 1, intrSource, 1, slaveSet );

// Create worker threads and add to thread set (for
    convenience)...
    ThreadSet threads;
    TestThread worker_thread1;
    TestThread worker_thread2;
    TestThread background_thread;
    threads.add( &worker_thread1 );
    threads.add( &worker_thread2 );

// Enqueue worker threads to slave schedulers...
    RtSchedDiscipline disc( (rt_disc_t)( rt_disc_realtime ) );
    slave1.enqueue( &worker_thread1, 0, disc );
    slave2.enqueue( &worker_thread2, 0, disc );

// Enqueue background thread to master scheduler...
    disc.set( (rt_disc_t)( rt_disc_background ) );
    master.enqueue( &background_thread, 0, disc );

// Start the master scheduler (this also starts slaves)...
    master.start();

// Wait for worker threads to finish (using the thread set)...
    threads.join();
```

4.6 REACT/Pro Implementation

As previously stated, Rt is based on SGI's REACT/Pro [CD1]. This implementation is a hard real-time implementation for SGI multi-processor machines

Fig. 2. Rt Class Diagram

with REACT/Pro installed. The interrupt sources available are the CPU-specific timer, real-time clock, vertical retrace interrupt, hardware interrupts, and a user-defined interrupt (recommended for debugging purposes only). This implementation requires certain privileges and kernel reconfiguration.

4.7 Simulated Real-Time Implementation

To test the correct execution of algorithms absent the need for real-time execution, it is only necessary to provide for the scheduling of threads. This capability is provided by implementing a *simulated real-time* frame scheduler. This is accomplished by creating a simulated frame scheduler implementation class within the bridge pattern structure already in place. The simulated real-time implementation allows testing of algorithms on multi-user and single user machines without the need for super user privileges and reconfigured kernels.

5 Non-uniformity Correction (NUC) Algorithms

The input response of each pixel over the resister array must be measured over a number of input values and its response function estimated. Lookup tables (LUT's), least squares fits to polynomials, or non-linear functions are potential models for the response function. In each case, a set of coefficients is given for each pixel over the entire array. While each pixel may have it's own response function, at present it is considered to be independent of its neighboring pixel's input. This means that image non-uniformity correction is a pixel-by-pixel transformation on an image.

5.1 Image Partitioning

Since the NUC process is pixel-by-pixel, the application of the correction is highly data parallel and the image may be divided into sub-images for processing by independent threads. But, that image partitioning cannot be completely arbitrary if false cache sharing and memory paging are to be avoided. Since C/C++ arrays are row-dominant, the image is divided into row strips so that each sub-image is composed of pixels that are contiguous in memory. In addition, the Rt framework restricts the thread CPU to only that thread and no load balancing schemes such as a work pile are necessary. The number of image strips is the same as the number of threads.

5.2 NUC Function Objects

The software must be able to apply arbitrary correction functions with minimal impact on the software. Due to conflicting requirements for both speed and accuracy, a wide variety of pixel mapping functions may be applied. To adapt to different NUC functions with minimal changes to the RNUC code, the RNUC

software employs *function objects* (a.k.a. *functors*) [BS1]. A function object maintains internal state (such as the NUC function coefficients) while providing the semantics of a function call. To separate the algorithms from the type of image data processed, the function objects are implemented as C++ templates. This represents an application of *generic programming* in combination with the object oriented techniques presented so far. The class template `NucOperator` is provided as a common base class for NUC algorithm classes. It has a single template argument that is the image type used by the imaging system. This allows easy adaptation to 8-bit or 16-bit imagery. `NucOperator` defines the pure virtual functions `apply` and `loadCoefficients` as hooks for derived NUC operator classes to define their own correction algorithm and coefficient input.

```
template< class ImageType >
class NucOperator
{
    public:
        typedef typename ImageType::value_type PixelType;
        typedef IP_ROI RoiType;

        /** Apply the action of this operator over a given roi. */
        virtual void apply( const ImageType& input, ImageType& output,
                           const RoiType& roi ) = 0;

        /** Define operator () to allow use as a functor. */
        void operator () ( const ImageType& input, ImageType& output,
                           const RoiType& roi )
        { this->apply(input,output,roi); }

        /** Load the correction coefficients.
        @param filename the name of the input file. */
        virtual void loadCoefficients( const cstring& filename ) = 0;
}; // end class NucOperator
```

A Look Up Table (LUT) image corrector can now be defined by subclassing from `NucOperator`.

```
template< class ImageType >
class LutCorrector : public NucOperator<ImageType>
{
    public:
        typedef NucOperator<ImageType> Base;

        /** Override NucOperator::apply. */
        void apply( const ImageType& input, ImageType& output,
                   const RoiType& roi );

        /** Override NucOperator::loadCoefficients. */
```

```
void loadCoefficients( const cstring& filename );

};// end class LutCorrector
```

5.3 Results

We now present results from our tests running on an 8-processor SGI Onyx-2. The NUC algorithms were applied to two images (16-bit pixel values) using up to 5 processors (see Table 1).

Table 1. RNUC Test Results

Image	Cpu's	Frame Rate (Hz)
256x256x1	1	78
	2	150
	3	222
	4	282
	5	340
512x512x1	1	20
	2	38
	3	60
	4	75
	5	90

6 Conclusion

We have demonstrated that, for our RNUC application, generic high-performance systems can remove the need for expensive, custom-built hardware. It is an often-repeated myth that making software object-oriented necessarily makes it slower. This is simply not the case and we have shown that high-performance, real-time requirements can be met using a combination of object-oriented and generic-programming techniques. In fact, it is their synergistic combination that can provide the best solution to the need for both flexible and efficient software.

References

[GE1] Gamma, Erich, et al.: Design Patterns: Elements of Reusable Object-Oriented Software (1995)
[CD1] Cortesi, David and Thomas, Susan: REACT Real-Time Programmer's Guide (1998)
[BS1] Stroustrup, Bjarne: The C++ Programming Language (2000)