A Java Development and Runtime Environment for Reconfigurable Computing

Don Davis, Michael Barr, Toby Bennett, Stephen Edwards,
Jonathan Harris, Ian Miller, Chris Schanck

TSI TelSys, Inc.
7100 Columbia Gateway Drive
Columbia, MD 21046
http://www.tsi-telsys.com

Abstract. Fast runtime reconfigurable hardware enables system designers to swap hardware into and out of an FPGA much as the pages of virtual memory are swapped into and out of virtual memory. Java provides a powerful object-oriented language with constructs to support multiple threads. In this paper, we discuss a method for developing reconfigurable hardware object class libraries, a runtime environment to manage these hardware objects and techniques for controlling such designs from the Java programming language.

1 Introduction

Field Programmable Gate Arrays are used for a wide variety of functions from implementing “glue-logic” to ASIC replacement. Perhaps one of the most compelling uses for these parts is as Reconfigurable Processing Units (RPUs) to implement changeable algorithms on “native silicon.” This is accomplished though reconfiguring the FPGA to implement a processing architecture that is optimized for the tasks at hand.

The newest generation of FPGA address provide a more robust platform for reconfigurable computer development by implementing some key enabling features such as fast runtime reconfiguration, partial reconfiguration, on-the-fly reprogrammability and the ability to read the internal state of the device. Given this minimum feature set, a component-based approach to algorithm design is achievable. There is nothing particularly new about component-based design. FPGA designers as well as software designers all make use of this approach. However, combining the productivity of software design with the performance of hardware design has not been effectively exploited. In this paper we describe an powerful development and runtime environment for developing and executing reconfigurable applications in a multi-threaded, multi-user environment.

2 Hardware Objects

Hardware designs that provide discrete functions and contain their own state and configuration information are referred to as hardware objects. These objects make up the components that a designer can use to implement a given design. One key feature of these hardware objects is that they are relocatable; they can be moved to different RPUs on a board or to different parts within a given RPU. A developer can use these hardware objects to implement specific functions in hardware which provides higher performance for the user. The runtime environment allows the developer to use hardware objects and RPUs easily without having to deal with issues that arise from the introduction of multiple threads and multiple users.

3 Goals and Approach

In order to move reconfigurable computing technology into the mainstream, application development must get easier rather than harder. An application programmer would rather spend their time and energy developing algorithms instead of the issues involved in managing the RPUs such as loading, executing and swapping hardware objects. This implies an intelligent abstraction layer for the RPUs that the programmer can interface to using high level commands. This software abstraction layer (the runtime) hides the details of the particular devices from the application programmer and allows him or her to use high-level requests to access the RPUs resources.
Based on these goals, our approach follows. There are three things that need to be done to create an application for an RPU-based platform:

1. The hardware objects required by the algorithm must either be created in-house or purchased.
2. Algorithms using those hardware objects need to be developed.
3. The available reconfigurable devices must be managed. The makes the loading, executing and swapping of hardware objects possible.

There are a wide range of hardware objects that can be reused by many different algorithms and algorithm developers. These objects, such as multipliers, FFTs, DCTs, etc. could be provided as libraries of hardware objects that can purchased by the application developer, saving time and effort. In the absence of ready-made hardware objects, traditional hardware design methods can be use to create hardware objects. The algorithms are known only to the application developer, so it makes sense that he or she should be the one responsible for developing them. Implementation of the algorithms will rely on the use of robust hardware objects to provide ease of design and performance. The management of the RPU resources is the responsibility of the runtime environment. The lowest layer of this environment are the specific to the particular RPUs used and their physical interconnection. However, the highest layer (the layer that the application programmer interfaces to) is a specified standard to allow for portability and the evolution of RPUs and associated technology.

4 HardwareObject Class

There are a number of common characteristics that all hardware objects support. They must be able to be relocated and loaded into an RPU. There must be some initialization and start sequence. They must have a method of assigning inputs and outputs and they must be able to be unloaded so that another hardware object can be swapped into the same space.

From a software perspective, the closest analogy to a hardware object is a thread in a multitasking operating system. Table 1 summarizes these similarities.

Thread	**Hardware Object**
Begins execution at a predefined location	Begins execution in a predefined state
Single-minded, executes linearly from start to end	Single-minded, executes continuously until complete
Independent of other threads	Independent of other objects
Executed in pseudo-parallel	Executed in parallel

Table 1: Threads vs. Hardware Objects

It is advantageous to capture as much of this common functionality as possible in a single Java class. That way, all of hardware objects can inherit these characteristics and then extend them with their own unique functionality. The parent Java class to capture this functionality is called HardwareObject. It inherits and extends Java's built-in Thread class. This allows it to leverage the similarities between hardware objects and threads as well as provides a thread to control the hardware object.

```
public class HardwareObject extends Thread {
       public HardwareObject (String name);
       public load();
       public unload();
       public enable();
       public disable();
       public start();
       public stop();
}
```

There are six methods in the Hardware Object class. The first two are for loading and unloading bitstreams. The second two are for controlling the clock input to the hardware object, therefore they control when the hardware object is running since all hardware objects need to be clocked in order to process data. The final two methods are start() and stop(). These methods override Thread.start() and Thread.stop() to provide them with knowledge about hardware objects.

5 Runtime Environment

Figure 1 shows our system architecture. ACEruntime is the runtime environment for the ACEcard and is in some sense the operating system for the ACEcard. ACEruntime is a Java application that runs on either the ACEcard's embedded processor or the host workstation. It is written in Java, so there are no issues with moving it to either processor.

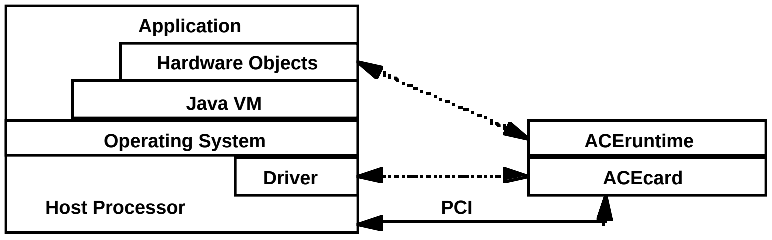

Figure 1: System Architecture

In our implementation, ACEruntime executes on the ACEcard embedded processor and controls the reconfigurable resources on the card. It is logically connected to the ACEobjects™ running on the host workstation. that is, the threads associated with the hardware objects are executed on the host processor while the actual hardware objects (and the requests to load(), unload(), enable() and disable()) are executed by the runtime environment on the embedded processor.

The primary purpose of the runtime environment is to provide a standard method for interfacing to a reconfigurable coprocessor. In addition, this interface must implement enough intelligence to handle high-level requests as well as issues of resource management, object placement and routing, object swapping, etc.

The design of ACEruntime is layered and modular. Layering allows appropriate abstractions to be implemented and assists in the modularity of the runtime. In addition, the modular nature of the runtime allows individual pieces of functionality to be changed and improved as new technologies and ideas emerge. In addition, support for different hardware architectures and devices is simply a matter of implementing the appropriate modules and inserting them into the ACEruntime framework. There are three layers: the Device Abstraction Layer, the Place and Route Layer, and the Hardware Object Scheduler.

5.1 Device Abstraction Layer

This is the lowest layer of ACEruntime. Its primary function is to isolate the other layers of the runtime from the particulars of a given RPU or FPGA and is used whenever the next layer needs to read or write configuration or state information to or from the RPU.

5.2 Place and Route Layer

This layer provides placement and routing of the hardware objects. Since hardware objects are already relationally placed and pre-intra object routed, this layer only has to deal with the coarse-grained placement and routing. That is, placement of entire hardware objects into the hardware page structure and only the routing necessary to connect multiple objects or connect the object to the framework. The place and route layer understands the basic structure of all hardware objects and has to make decisions about where to best place a needed object.

5.3 Hardware Object Scheduler

This is the highest layer of the runtime and provides the interface to the HardwareObject class and its descendants. It translates these high-level requests into lower-level requests for the place and route layer. One of its core functions is to handle the scheduling of Hardware Objects. Since the RPUs are a shared system resource, there are potentially multiple threads and multiple users all requesting access to those resources simultaneously. The scheduler must accept those requests and ensure that they are all serviced in a fair and timely manner. This may involve swapping hardware objects into and out of the RPUs to ensure equal access.

6 Example Application

An example application may help to illustrate how all of these elements work together in a system. A JPEG encoder design is straightforward. It involves a discrete cosine transform (DCT), a quanitizer (Q) and a variable length coder (VLC).

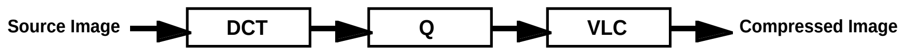

Figure 8: JPEG Encoder

The first step is to acquire the three blocks necessary to implement this algorithm. It is relatively easy to convince yourself that of each of these blocks can be considered a hardware object. A hardware designer must implement the logic design necessary for each of the required functions. The output of this process is three bitstreams. For each of these bitstreams, we must provide a Java class. To do this, you reference the bitstream data, give the object a name, write constructors, create input and output assignment mechanisms and develop a software thread. The DCT class is shown below (the Q and VLC classes are similar):

```
public class DCT extends HardwareObject {
       public static String relocHardwareObj = “DCT”;
       public DCT ();
       public DCT (HardwareObject inputObject, OutputPort source,
                           InputPort target);
       public setInput (int[] inputData);
       public int[] getOutput ();
       public void run ();
}
```

Notice that there is a constructor in the design that accepts another HardwareObject-derived class as input. This allows a direct connection of the output signals from that hardware object to the input signals of another hardware object. Once the HardwareObjects are created, the JPEG algorithm is simple.

```
int[] jpegEncoder(int[] SourceImage) {
     AtomicObject dct = new DCT();
     AtomicObject q   = new Q(dct);
     AtomicObject vlc = new VLC(q);
     dct.input(SourceImage);
     return vlc.output();
}
```

The first step is to instantiate the three hardware objects. When this is complete, the hardware object scheduler has scheduled them for execution on the reconfigurable processor. The place and route layer has been tasked to find space for the objects. In addition, the hardware object scheduler has been instructed to route the output of the DCT to the input of Q and the output of Q to the input of VLC. The scheduler passes this information to the place and route layer and that layer will handle routing the hardware objects together. If there is not room in the physical device to fit all three objects at the same time, then the objects will be scheduled to be instantiated in sequence. In this case, the inputs and output will not be physically routed together on the RPU, but instead logically routed together by passing the output data from the DCT to Q and the output of Q to VLC. To actually create the compressed digital image, input is provided to the DCT object and the results appear at the output of the VLC object. In a typical implementation, the call to VLC.output() will block until the software thread associated with the VLC object receives all of the output data.

7 Conclusion

In order for reconfigurable computing to become a mainstream technology, there must be a range of available platforms to address various price/performance needs with a common and easy to use application development method. We believe that by combining the power and user base of Java with the performance of reconfigurable processors and tying this together with a robust runtime environment that supports threading and multiple users, a major step has been taken.

In the future, new reconfigurable platforms that provide more flexibility and higher levels of performance will emerge. Perhaps more importantly, they will also contain native silicon support to address many of the issues inherent in using these devices as reconfigurable processors. In addition, runtime environments will become faster and more optimal; a designer will incur less of a penalty in terms of overhead when instantiating or swapping objects. Tools for developing hardware objects will become easier to use and provide more automatic functions while libraries of reusable hardware objects will address more application domains.

Acknowledgments

This material is based upon work supported by DARPA/ITO under contract No. DABT63-97-C-0044.