Reconfigurable Computing Application Frameworks *

Anthony L. Slade, Brent E. Nelson and Brad L. Hutchings
Brigham Young University
Configurable Computing Laboratory
Provo, UT 84602, USA
{aslade, nelson, hutch}@ee.byu.edu

Abstract

FPGA-based Configurable Computing Machines (CCMs) offer powerful and flexible general-purpose computing platforms. However, development for FPGA-based designs using modern CAD tools is geared mainly toward an ASIC-like process. This is inadequate for the needs of CCM application development. This paper discusses an application framework for developing CCM-based applications beyond just the hardware configuration. This framework leverages the advantages of CCMs (availability, programmability, visibility, and controllability) to help create CCM-based applications throughout the entire development process (i.e. design, debug, and deploy). The framework itself is deployed with the final application, thus permitting dynamic circuit configurations that include data folding optimizations based on user input. The resulting system aids in creating applications that are potentially more intuitive, easier to develop, and better performing. An example application demonstrates the use of the application framework and the potential benefits.

1 Application Acceleration Using FPGA-based CCMs

Configurable computing machines (CCMs) are FPGA-based processing and computing platforms. Figure 1 shows the components of a typical CCM architecture. CCMs provide a versatile and powerful computing platform that may be integrated with a host system. The FPGAs in a CCM can be configured by the host system to perform virtually any type of computation or process. This makes CCMs attractive computing platforms with flexibility comparable to that of microprocessor-based systems and performance comparable to that of ASIC designs.

*This work is supported by the Defense Advanced Research Projects Agency (DARPA) and is funded via Grant No. F33615-99-C-1502.

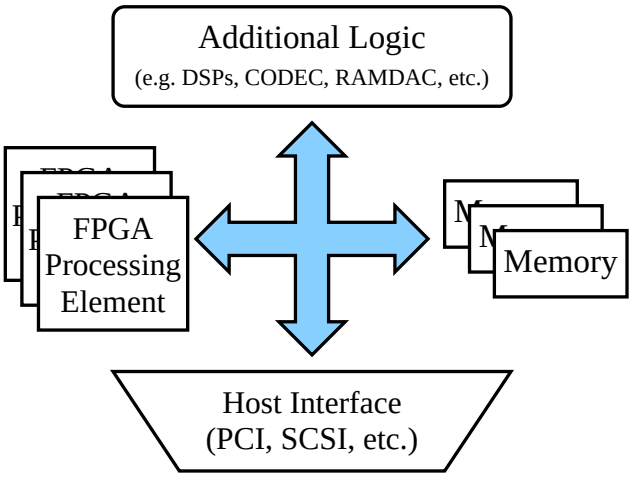

Figure 1. Configurable Computing Machine (CCM) Block Diagram.

Because CCM computation and processing occurs in hardware, applications targeted to CCMs may be able to run several times faster than if they were executed in software running on a microprocessor. In some cases, CCM-based applications may be more than an order of magnitude faster than alternative implementations[13]. CCMs are also capable of improving the performance-to-price ratio in many applications[12].

Although CCMs do offer a number of potential advantages, the cost of producing a full application for use on a CCM platform can still be high. One reason is the multiple types of design components found in a CCM-based application. (See Figure 2.) The interface drivers are often provided by the developers of the CCM hardware. However, application developers must still deal with the large tasks of developing the hardware configuration and application software.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **2003**	2. REPORT TYPE	3. DATES COVERED **00-00-2003 to 00-00-2003**
4. TITLE AND SUBTITLE **Reconfigurable Computing Application Frameworks**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Brigham Young University,Department of Electrical and Computer Engineering,Provo,UT,84602**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**10**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

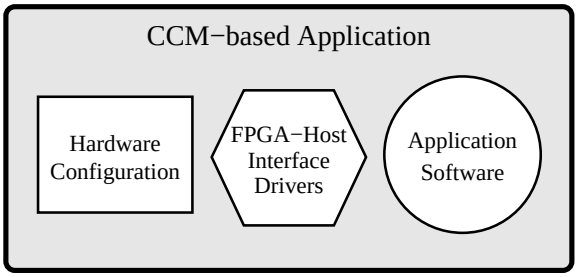

Figure 2. The hardware and software design components of a CCM application.

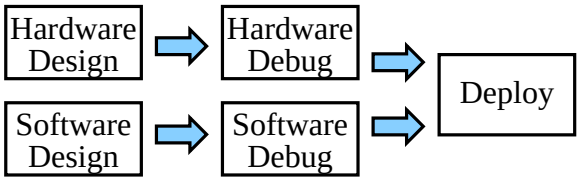

Figure 3. Traditional stages of CCM application production.

Figure 4. Combined stages of CCM application production.

2 Limitations of Current CAD Tools for CCM Application Development

In order to quickly generate a new market for their products, FPGA manufacturers have relied heavily on the existing electronic design automation (EDA) and CAD tool infrastructure. By offering libraries for use with major CAD tools, or information sufficient for others to create such libraries, FPGA manufacturers have allowed designers to use their favorite EDA/CAD tools to create designs that target FPGA devices. While this strategy has allowed FPGA producers to successfully build up a large market for their products, it has also created a development system that is ASIC-oriented[8]. This ASIC-oriented nature of CAD tools has been adequate for creating FPGA designs that serve as ASIC prototypes or replacements. However, focusing only on hardware design and simulation limits development for CCM applications by ignoring the benefits of CCM features and by not supporting the software development, which is crucial to the final application deployment.

The ASIC-orientation of EDA/CAD tools commonly used for FPGA configuration design results in a split development path for the hardware and software components of the application as shown in Figure 3. The disjoint, often sequential, development of the hardware design and the software design extends the time necessary to develop the whole CCM application. A combined development path, as shown in Figure 4, streamlines development of the hardware and software and reduces the complexity of the deployment stage.

At this point, it should be noted that this paper is not about the automated partitioning problem for hardware/-software co-design. That is, this paper is not about a tool for system specification, co-design, and co-simulation. Rather, this paper discusses methods of joining the hardware and software development environments in such a way that extends the benefits provided by an EDA tool beyond the design and debug stages of development. Although this new integrated approach to developing CCM applications can facilitate the partitioning problem, it also has benefits that simplify application development in general, and that may result in applications that perform significantly better.

3 Leveraging Features of CCMs in Application Development

Development of CCM applications may be improved by leveraging the features of FPGAs and FPGA-based CCM platforms. As noted by Hutchings and Nelson[7], three major features of CCMs are:

- Availability of the hardware at design time
- Programmability and re-programmability of the hardware
- Visibility into the hardware

To this list, Graham[4] adds the following:

- Controllability of the hardware state
- System execution controllability

Two major benefits come as a result of these features of CCM systems: a reduction in the need for simulation for hardware design verification, and tighter integration of the software and the hardware components of CCM applications, particularly in the deploy stage of development and beyond.

Traditional EDA/CAD tools have assumed that the design may only be put into hardware form after extensively simulating it in order to fully verify its functionality. Relying exclusively on simulation to verify hardware designs is resource- and time-intensive, making design verification a bottleneck in the development process. While simulation

does have its place in designing for FPGAs, making use of the target CCM platform for in-hardware execution can reduce reliance on simulation for circuit verification. Using target FPGA hardware as a replacement for full simulation is possible due to the extraordinary internal visibility made available by the hardware.[1] Unfortunately, traditional EDA/CAD systems typically do not offer support for in-hardware execution and debugging of designs; rather, they focus only on the hardware design.

Design and Debug

In-hardware execution is typically several orders of magnitude faster than simulation[4]. Integrating the information from in-hardware execution of incremental and final builds of the hardware design within the development environment allows the designer to verify the design in less time[7]. Shorter design and debug cycles may then permit further design exploration, and thus improve the design overall.

CCM features permit the hardware to be viewed as an extension of the application software. This gives a more useful view of the CCM application as a whole computing system, rather than two separate pieces. This enables further design exploration that helps to determine the optimal partition between the hardware and software components of the application. Also, integrated application development helps detect and eliminate design flaws that might not be evident in separated design flows. The integrated development approach also allows developers to focus attention on the interface with which the final user interacts. The result is an application that not only solves a particular problem, but that is also intuitive to the target audience. In summary, modern CAD tools lack integrated hardware and software development environments to support the deployment of complete CCM applications.

Deploy

Perhaps one of the greatest benefits of an integrated design process is the potential for improved application performance. When an ASIC design is fabricated, it usually must work with a wide range of data. Such generic designs come with the tradeoffs of larger size and slower speeds. The re-programmability of CCMs, however, allows for performance enhancements to be made by embedding specific data from the problem being solved by the application into the hardware design. For example, complex combinational logic may be replaced by faster lookup tables; large, slow, general-purpose multipliers may be replaced by more efficient constant-coefficient multipliers; etc. Such optimizations, known as data folding[3] or constant propagation[14], can be used in applications such as FIR filters (by embedding the filter coefficients in the multipliers) or pattern matching circuits (by creating constant-value comparators).

With high-performance CCM applications, when the data changes, so does the problem to be solved. That is, when the end user has a new set of information to be processed by the application, CCM applications have the capability to adapt to that new set of data for optimal performance. Such adaptation may come as a result of regenerating netlists and performing placement and routing again. Or the application may adapt by modifying FPGA configuration bit streams or some intermediate form of the hardware configuration[4]. Thus, performance improvements gained from data folding may require significant portions of the hardware design tool to be included in the final application. This defines a new role for EDA/CAD tools; that is, certain CCM applications may require the tools themselves to be deployed along with the applications. The benefits of hardware reconfiguration enabled by the tools may then remain with the final application in the deploy stage and beyond.

4 Reconfigurable Computing Application Frameworks

The traditional monolithic EDA/CAD tool system is insufficient to support the unique needs of CCM application design, debug, and deployment. This section explains how application frameworks offer a better alternative. *Reconfigurable computing application frameworks* (RCAFs) help designers use CCM features to easily develop a custom design and debug environment that is specific to a given application. Using an RCAF, the preparations for deployment of a CCM-based application will occur simultaneously as the whole application, including both the hardware and software components, is built up in an integrated fashion.

As general-purpose computing platforms, CCMs may be used as the processing cores for a wide variety of applications. Loading a new hardware configuration into the FPGAs of a CCM changes its functionality. Configuring the FPGA hardware of a CCM is much like loading in a new software program on a microprocessor-based machine. And much like software loading, CCM configuration and reconfiguration can be dynamic and interactive. RCAFs support the dynamic, software-like features and operation of CCMs. As shown in Figure 5, an RCAF implementation is a bridge between the CCM hardware and the CCM application software running on the host system. The RCAF is integrated and deployed with the CCM application. The scalable and modular nature of the RCAF makes this possible; this is typically not feasible with large, monolithic CAD tools.

[1] For example, Xilinx FPGAs offer a "readback" capability, which gives the current state of all register and memory elements in the FPGA device.

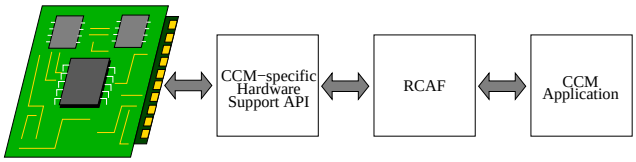

Figure 5. The RCAF interfaces between the CCM hardware and application software

4.1 RCAF Interface Between CCM Hardware and CCM Software

An RCAF offers simple and accessible interfaces to all of the important features of targeted CCM platforms. The RCAF takes care of the low-level details of configuring and operating the CCM hardware (i.e. FPGA configuration, reset, clock starting, stopping, and stepping, querying hardware state, etc.). Ideally, an RCAF will work with virtually any CCM hardware platform. The same interface is available to the application developer, no matter what the target platform may be.

In this way, the RCAF functions much like conventional board-support software packages such as are provided with commercially available CCM platforms (e.g. SLAAC API, Anapolis API, etc.). However, the RCAF actually works on top of such hardware support packages to provide much more in its application programming interfaces (APIs). The RCAF also includes a system that aids in the interpretation of the hardware state as it relates to the original hardware design. The RCAF maintains a reference mapping between the circuit design components and their corresponding circuit elements in hardware. That is, the RCAF takes care of the overhead necessary to give the application developer a transparent view of the circuit structure and state while the circuit is executing in hardware.

Because the hardware view of the circuit provided by the RCAF is the same as the design view, circuit interaction components (e.g. schematic viewers, circuit execution controls, custom circuit visualization components, etc.) may be consistent across the design, debug, and even deployment stages of application development. Beyond even that, the RCAF contains APIs that aid in establishing a simple architecture that creates a communications network that allows interactive components to send and receive messages to and from other interactive components as well as the hardware execution and state controllers. This architecture helps maintain a modular and scalable application system for easy maintenance and updating.

4.2 RCAF Support for (Dynamic) Circuit Design

The RCAF is not just an interface layer in a CCM-based application. It also contains the resources necessary for creating the hardware configuration for the CCM application. That is, the RCAF is also a full-featured FPGA circuit design tool. The circuit design, which is built up programmatically through the RCAF, may be netlisted and then translated by technology-specific tools into a configuration image for the CCM hardware.

Because the RCAF is a part of the deployed application, the circuit design need not be static. The RCAF may dynamically generate a custom version of the CCM hardware based on information provided by the application. As explained in Section 3, this enables CCM applications to optimize the hardware configuration by folding in data provided by the user. Whether by manipulating existing configuration images or by regenerating netlists and performing placement and routing, the RCAF can assist in providing dynamic, interactive circuit creation capabilities in the field to significantly improve application performance.

5 A Reconfigurable Computing Application Framework Implementation

The general architecture of a *reconfigurable computing application framework* implemented at Brigham Young University's Configurable Computing Laboratory was patterned after the model-view-controller architecture shown in Figure 6. The *model* is the fundamental data being manipulated by the system or the computation being performed. The *view* is the system component that determines how data is presented to the user. The *controller* is the portion of the system that receives user input and interprets it to perform system tasks. These tasks may perform manipulations of the data or they may change the way in which the view displays information about the data. The model-view-controller architecture divides the functionality of a system along boundaries that allow for simple system production, maintenance, and scalability[11]. It also establishes a natural communications network between the three components. The description of the RCAF that follows is divided along the boundaries established by the model-view-controller architecture.

5.1 The RCAF Model Component

The underlying *model* for the RCAF is implemented by JHDL. JHDL is a tool for designing and simulating digital circuits, and is especially well-suited for reconfigurable computing designs[1, 6]. JHDL performs the role of model for the RCAF by representing the structure and state of the hardware design. Manipulations to the model come in the

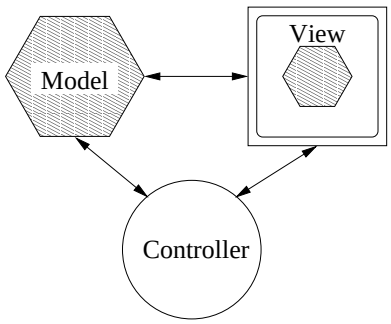

Figure 6. The model-view-controller architecture for user interfaces[11].

form of stepping the circuit clock and applying values to the circuit inputs.

Figure 7 shows the general architecture of a JHDL design as well as the types of APIs exposed to external systems. These APIs are used to manipulate the model as well as to extract information about the model's state. Each API is explained separately below.

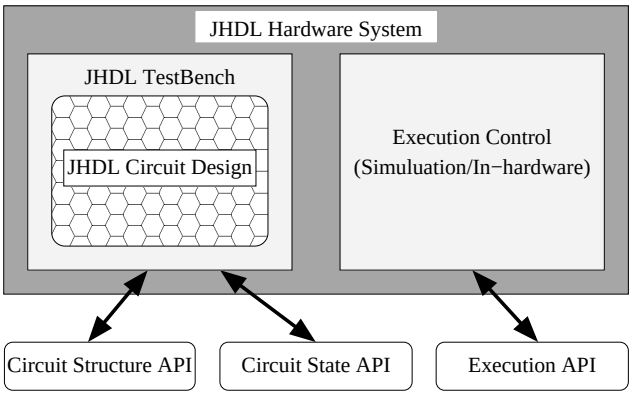

Figure 7. JHDL APIs offer access to circuit information and control.

Circuit Structure API

The circuit structure API gives access to the components of a circuit design (cells and wires) and their relationships to each other. The structural view exposed by the JHDL model is that of the original circuit design. This design may differ from the physical circuit in hardware. For example, the circuit model on the left in Figure 8 contains two wires that are absent from the physical FPGA hardware configuration represented on the right. However, the JHDL model maintains information that maps structural components of the design model to their counterparts in the physical circuit in the CCM hardware[4].

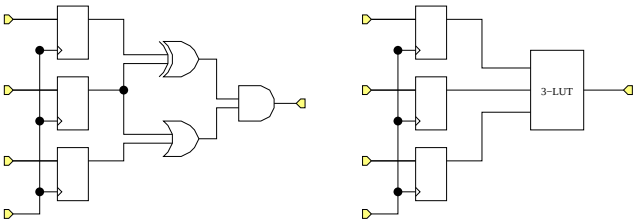

Figure 8. Mapping a circuit model to hardware often loses intermediate logic and wires.

Circuit State API

The circuit state API provides a means of determining the values on wires and in memory devices. The same API is used whether the circuit is being simulated or is being executed in actual hardware. Both methods of circuit execution require the presence of the JHDL simulator as part of the circuit model. This is because the state of the hardware (obtained, for example, via hardware "readback" features present in certain Xilinx FPGAs) usually only contains state information from non-combinational logic elements. Therefore, for the circuit design shown in Figure 8, hardware readback would only return the state of the three flip flops. The JHDL simulator then uses those values to interpolate the remaining three wire values, including the two that were subsumed into the configuration of the FPGA 3-LUT. Such information can be crucial when debugging CCM applications. This also means that the application may provide hardware visibility without requiring the hardware configuration to contain additional logic meant only to route signals to external pins. Thus, JHDL makes added circuit information available without added circuit complexity or latency.

Execution API

The execution API provides a means of controlling circuit simulation and execution. Since the simulator is used in both simulation-only and in-hardware execution of the circuit design, it is the agent that performs the data model manipulation for both modes of execution. Therefore, the execution API is also consistent across execution modes. Besides providing methods for stepping and cycling the system clock, the execution API also provides a mechanism that allows components external to the JHDL design to receive notification of the completion of circuit execution events,

i.e. clock steps, multi-step runs, and circuit resets. Application viewers use this portion of the execution API to know when the circuit has settled so that they may query circuit state information.

Integrating External Components

JHDL is based on a general-purpose programming language (Java). This allows the JHDL APIs to be accessed *directly* from other systems, giving JHDL an advantage over proprietary languages and HDLs. Elements of a JHDL design are represented by objects. Some example JHDL API methods provided by these objects are displayed in Table 1. The RCAF controller and viewers are seamlessly integrated with the JHDL model through these and other methods of the JHDL APIs.

5.2 The RCAF Controller Component

To receive and interpret user input and execute corresponding system-wide tasks, the RCAF has a centralized *controller*. The principle RCAF controller is contained in a class named `Broker`. The `Broker` class gets its name from its role as an intermediary between the other major components of the RCAF system. This class is the only component of the RCAF that has a system-wide view of all of the other components. Having a single class with this information instead of various components with a network of references to each other makes the RCAF simple and easy to maintain. This also makes the system modular and scalable. That is, only one class needs to be modified to change the set of features and components included for a given application.

The `Broker` class contains all of the methods to perform system-wide tasks. These include creating application viewers, disposing unused viewers, managing communication between system components, and controlling the execution of the hardware.

The `Broker` is aided by a command interpreter. The interpreter accepts text-based commands from the application user. The result is the invocation of corresponding methods of the `Broker` class. These text commands may be stored in a history log and executed again later, making the RCAF system scriptable. This system of text commands also helps to make the RCAF controller modular and extensible, as will be shown in the example in Section 6. In graphical environments, the `Broker` creates listeners that translate low-level events from the viewers (e.g. button presses, mouse clicks, and key strokes) into text commands to give to the command interpreter. This listener system for inter-component communication is based on the listener model commonly used in user interface architectures, notably in the Java Swing classes. Figure 9 shows how these various controller components work together as well as how they fit into the overall RCAF architecture.

5.3 The RCAF View Component

RCAF *viewers* offer application-specific presentations of the circuit design. They also offer a portal through which the user may apply inputs to be passed to the RCAF controller. While a default RCAF configuration for general-purpose circuit visualization includes a standard set of viewers (e.g. schematic viewers, circuit hierarchy viewers, memory viewers, etc.) each unique CCM application may easily add to or remove from this default set of viewers as appropriate for the application. Each custom viewer merely needs to follow the RCAF system's method for generating events. In the current implementation of the RCAF this is done by generating *action events* as in the Java Swing user interface model. Each viewer should also provide a listener interface, implementations of which may be used to translate action events into commands for the command interpreter.

5.4 Integrating the RCAF Model, View, and Controller

Figure 10 shows an example interaction between the various components of the RCAF architecture. In this example, the user double-clicks with a mouse pointer on a portion of the schematic viewer that displays a wire in the circuit design. This generates an event that is picked up by the GUI listener registered with the schematic viewer. This listener then translates that event into a textual command that is passed to the command interpreter. The interpreter then makes the appropriate method call in the `Broker` class. The `Broker` knows of two other viewers that should be affected by that command and tells them to update their output to include the new wire. Those viewers then query the necessary information from the circuit model and update their displays.

In general, the `Broker` holds the central position in the RCAF system. Modifications to the `Broker` change the set of available RCAF viewers. Since the `Broker` knows which viewers may exist in the system, and their relationships to itself and to each other, it knows how to configure the command interpreter and the viewer listeners. Having just this one class with information that makes up a system-wide view makes the RCAF easier to maintain and update. The single system controller simplifies removing and adding viewers to the system and managing system communication.

Table 1. Some JHDL methods for accessing circuit design information and control

JHDL Class	Method Name (arguments)	Return Type
`HWSystem`	`findNamed(String name)`	Returns an arbitrary member of circuit model.
`Cell`	`getDescendants()`	Returns a list of the children cells and other descendants of the given cell
`Wire`	`get()`	Returns the current value of the given wire
`HWSystem`	`step (int numSteps)`	Steps the circuit clock for a specified number of clock edges

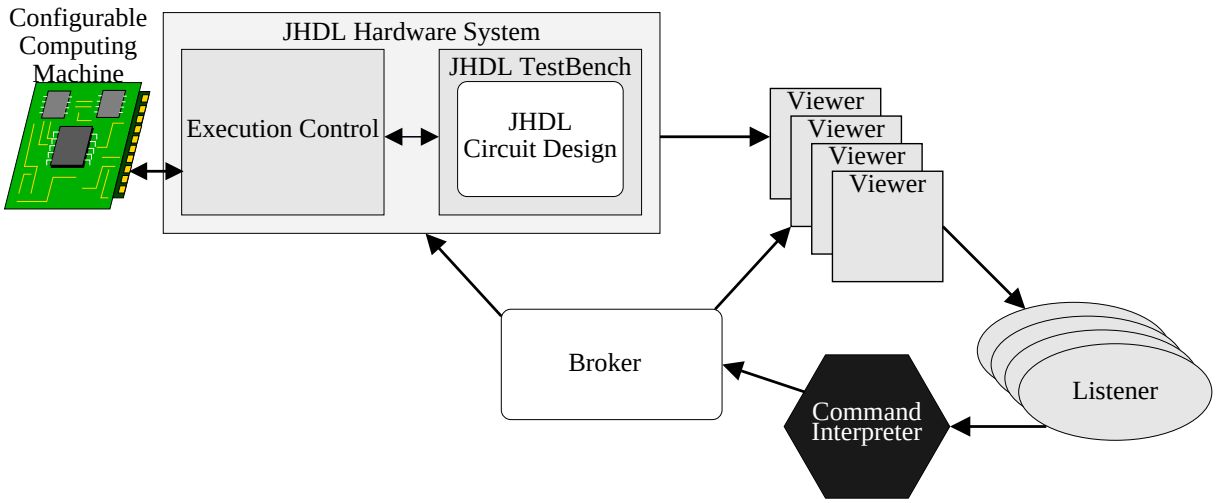

Figure 9. The RCAF system architecture.

6 Building a CCM Application

The RCAF provides a basic architecture on which to build a full CCM-based application. The developer needs only only to create viewers appropriate for the given application and extend the capabilities of the `Broker` class to interact with those viewers. The following example demonstrates how this may be done. This example application implements an edit distance calculator.[2] The circuit that calculates the edit distance in this application is built from a linear systolic array, a simple architecture easily optimized for FPGA hardware[10, 5, 2]. Each processor in the array has a character of the target string embedded in the configuration of the FPGA LUTs of which it is made.

The following description of this example application is in terms of the model, view, and controller components of the system. This shows how the various application components correspond with the architectural components of the RCAF. It also shows how the development of the application may be subdivided into easy-to-manage portions. Thus a group of designers may easily develop various parts of a significant application concurrently.

[2]The edit distance between two strings is the number of character insertions, deletions, and substitutions, which may be weighted differently, that it takes to change a source string into a target string.

6.1 Edit Distance Application Model

The model for the edit distance application is a parameterizable JHDL circuit design. Figure 11 shows a portion of the code for the `EditDistance` class that builds the circuit model. The parameters that determine the circuit structure and function are the width of the output wire `distanceOut` and the contents of the string `targetStr`. The output wire is connected to a counter that tallies the total number of operations needed to convert the source string to the target string. Therefore, the width of this wire determines the maximum edit distance value the circuit may detect. The user provides the desired target string parameter when starting the application. This means that the circuit will be optimized for the data at execution time, even if it changes from one use of the application to the next. Although this specific example only demonstrates the RCAF system in simulation mode, the deployment-time optimizations for in-hardware execution could be performed in a variety of ways. The hardware design could be used to produce a netlist which could then be processed by placement, routing, and configuration bitstream generation tools. Or a pre-existing, low-level, FPGA-specific configuration could be modified by a tool such as JBits to change the embedded constants. Previous work at Brigham Young

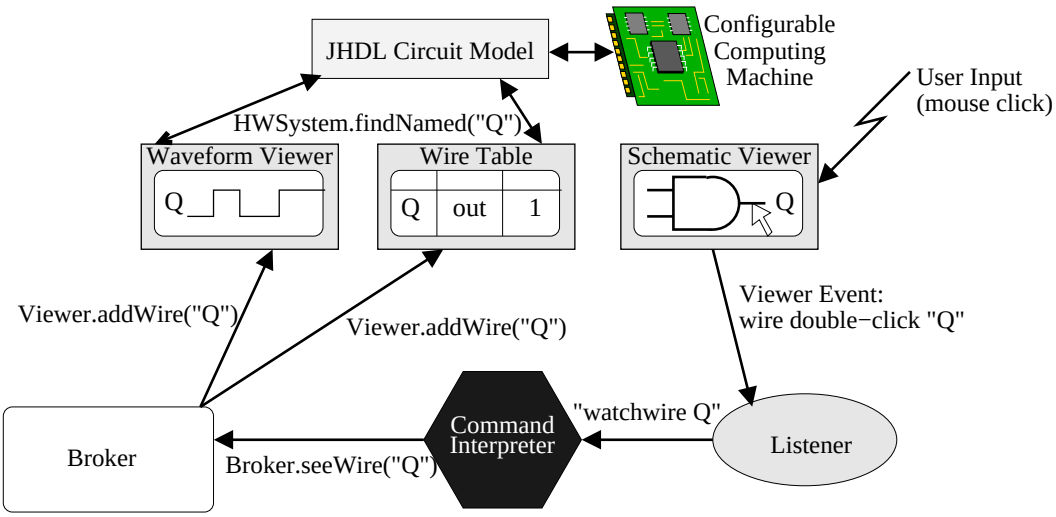

Figure 10. Example communication in the RCAF model-view-controller architecture.

University has also shown that such optimizations can be performed by directly modifying pre-bitstream or bitstream configuration files[9]. The execution-time modifications could take anywhere from seconds to several minutes. The method of updating optimizations between execution runs could be determined by the RCAF or the application code built on top of it, making the whole process transparent to the user.

```
public class EditDistance
       extends Logic {
  public static CellInterface[]
    cell_interface = {
      in("charIn",4),
      out("distanceOut",PARAM_WIDTH),
      param("outputWidth",INTEGER),
      param("targetString",STRING),
  };
  public EditDistance(
          Node parent,
          Wire charIn,
          Wire distanceOut,
          String targetStr) {
     // constructor code ...
  }
} // end class EditDistance
```

Figure 11. The `EditDistance` class creates the circuit model for the edit distance application.

6.2 Edit Distance Application View

The view for the edit distance application (Figure 12) provides a simple interface that allows the user to input a source string, control the execution of the circuit, and view the progress of the string as it cycles through the processors of the systolic array. The inputs are in the form of text fields. The circuit visualization shows information about each processor in the circuit. On the top of the view of each processor is displayed the character that is embedded in that processor. Below that is a symbol that indicates whether the current source character in that processor is equal or not. Below that is the character from the source string. Finally, on the bottom is the output value from the processor, which is passed down the array to the accumulator connected to the circuit output.

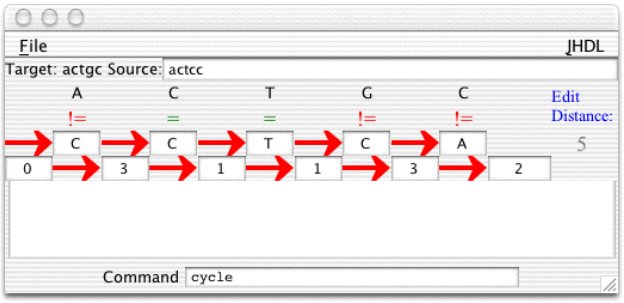

Figure 12. The graphical user interface for the edit distance application.

Between cycles of the system clock, the viewer updates the values displayed. The values are obtained by using methods shown in Table 1. References to the circuit wires that contain the values of interest are obtained with the `findNamed` method. The actual values are then obtained through the `get` method.

Each of the processor views in the edit distance application is a separate, parameterizable GUI panel. This GUI subpanel was developed along with the hardware design for an individual processor. This custom view of a single processor was used to help debug the hardware design. When the processor design was verified, both the processor and GUI subpanel were duplicated in parameterizable arrays in the hardware design and complete custom viewer respectively. Thus the development of the custom viewer not only built up the final application interface, it also aided in the hardware design and debug from the beginning of application development.

6.3 Edit Distance Application Controller

The controller for the edit distance application is built by extending the features of the default RCAF `Broker` class. Additional, custom commands are also added to the RCAF command interpreter. The code for the custom `Broker` class is shown in Figure 13. A custom edit distance viewer is created in the `EDBroker` constructor on line 6. Line 13 creates and registers an instance of the `EDCLICommand` class. This class implements a command for the RCAF command interpreter (the `CLInterpreter` class). This command is tied to the text input field in the custom viewer in Figure 12. Entering a new source string in the text field causes the custom command to be generated and sent to the command interpreter, and hence the `Broker`. This results in an updated data schedule for the source string input values that are streamed into the system. This example of custom interactive circuit input and control shows how the deployed application can be built in such a way that is most intuitive to the final user, not just the hardware designer.

```
public class EDBroker
  extends Broker {
  public EDBroker(HWSystem system,
                  CLInterpreter interp) {
    super(system,interp);                    5
    new EDFrame(system,interp);
  }
  protected void registerCLICommands(
                  HWSystem system,
                  CLInterpreter interp) {    10
    super.registerCLICommands(system,
                              interp);
    EDCLICommand edc
         = new EDCLICommand(interp);
  }                                          15
} // end class EDBroker
```

Figure 13. The `EDBroker` class extends the default RCAF controller for the edit distance application.

6.4 Completing the Edit Distance Application

The edit distance application is executed by a JHDL testbench class. This class is the final deployment and execution point of the application; in this class, the system components (the circuit model, the viewers, and the controller) are built and finally integrated into the complete CCM application. The code for this class is shown in Figure 14. It contains a `main` method that receives user data on lines 21 and 23. That information is then used to create the testbench, and hence the circuit design itself, on line 26. This is the point at which the RCAF dynamic circuit generation features are activated to create the optimal circuit configuration for the user. Finally, the application creates an `EDBroker` on line 32 to start up the interactive components (i.e. the custom viewer and the controller elements).

```
public class tbEditDistance
  extends Logic implements TestBench {
  public tbEditDistance(Node parent,
                  int width,
                  String target) {           5
    super(parent);
    Wire charIn
      = wire(4,"charIn");
    Wire distance
      = wire(width,"distance");              10
    new EditDistance(this,
                 charIn,
                 distance,
                 target);
    Stimulator stim                          15
        = new Stimulator(this);
    stim.addWire(charIn);
  }
  public static void main(
        String[] args) {                     20
    int width
      = Integer.parseInt(args[0]);
    String target = args[1];
    HWSystem system
        = new HWSystem();                    25
    new tbEditDistance(
         system,
         width,
         target);
    CLInterpreter interp                     30
        = new CLInterpreter();
    new EDBroker(system,interp);
  }
} // end class tbEditDistance
```

Figure 14. The `tbEditDistance` class builds and integrates the application components.

This example application demonstrates how simple it is to extend the existing RCAF to quickly build up a CCM application. The source code for this example application is only 1321 lines long, including the circuit design (696

lines), controller extensions (116 lines), and the custom viewer (507 lines). After compilation and inclusion with the RCAF, the total size of the application is from two to three megabytes compressed, depending on features included. Of that, the application layered on top of the RCAF is a mere twenty-two kilobytes. Given the features that could be included with this application—dynamic circuit model generation, netlisting, platform control, etc.—the result is a small (relative to most other full-featured CAD tools) simple, deployable application. The end user needs only to execute the application in an appropriate runtime environment (i.e. CCM-specific drivers must be present, a Java runtime environment must be available, and FPGA-specific placement and routing tools may be required for design reconfiguration).

7 Conclusion

Existing tools for FPGA design are inadequate to support the design, debug and deployment of configurable computing machine applications. This paper presents a method of developing applications for configurable computing machines based on an application framework. This framework, a *reconfigurable computing application framework*, uses sound architectural principles for interactive applications to create a foundation for CCM applications. Using this framework can simplify the process of developing and verifying new CCM applications. This includes the ability to verify and execute the application in the actual CCM hardware.

The framework also simplifies the task of creating a custom interface that best meets the needs of the application. The framework does not constrain the application to a predefined system like an ordinary CAD tool. Rather, the application is built on a foundation that offers the resources required to create a hardware configuration and corresponding model, and that allows the application to use its own custom interface. The framework can also scale to allow the application to include or exclude whatever it needs to be complete.

Because the RCAF itself is deployed with the final application, the underlying data model also allows the application to dynamically create the hardware configuration. This allows the application to be optimized for the user. Whenever the user-supplied data changes, the application can change the hardware configuration to improve overall application performance.

References

[1] P. Bellows and B. L. Hutchings. JHDL - an HDL for reconfigurable systems. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 175–184, Napa, CA, April 1998. IEEE Computer Society, IEEE Computer Society Press.

[2] I. M. Bland and G. Megson. The systolic array genetic algorithm, an example of systolic arrays as a reconfigurable design methodology. In K. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 260–261, Napa Valley, CA, USA, 1998.

[3] P. W. Foulk. Data-folding in SRAM configurable FPGAs. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 163–171, Napa, CA, Apr. 1993.

[4] P. S. Graham. *Logical Hardware Debuggers for FPGA-based Systems*. PhD thesis, Brigham Young University, Provo, UT, USA, December 2001.

[5] D. T. Hoang. Searching genetic databases on Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, Napa, CA, Apr. 1993.

[6] B. L. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A CAD suite for high-performance FPGA design. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–24, Napa, CA, April 1999. IEEE Computer Society, IEEE Computer Society Press.

[7] B. L. Hutchings and B. E. Nelson. Unifying simulation and execution in a design environment for FPGA systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9:201–205, February 2001.

[8] F. J. Kurdahi, N. Bagherzadeh, P. Athanas, and J. L. Munoz. Guest editors' introduction: Configurable computing. *IEEE Design & Test of Computers*, 17(1):17–19, January–March 2000.

[9] W. J. Landaker. Using hardware context-switching to enable a multitasking reconfigurable computer system. Master's thesis, Brigham Young University, Provo, UT, USA, August 2002.

[10] R. J. Lipton and D. Lopresti. A systolic array for rapid string comparison. In H. Fuchs, editor, *1985 Chapel Hill Conference on VLSI*, pages 363–376. Computer Science Press, 1985.

[11] D. R. Olsen, Jr. *Developing User Interfaces*. Morgan Kaufmann, 1998.

[12] S. Salamone. Is acceleration hardware the wave of the future? *Bio-IT World*, July 2002.

[13] J. Villasenor and B. L. Hutchings. The flexibility of configurable computing. *IEEE Signal Processing Magazine*, 15:67–84, September 1998.

[14] M. J. Wirthlin and B. L. Hutchings. Improving functional density through run-time constant propagation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 86–92, Monterey, CA, Feb. 1997.