Framework for architecture-independent run-time reconfigurable applications

David I. Lehn*, Rhett D. Hudson, Peter M. Athanas

Bradley Department of Electrical and Computer Engineering,
Virginia Tech, Blacksburg, VA 24061-0111

ABSTRACT

Configurable Computing Machines (CCMs) have emerged as a technology with the computational benefits of custom ASICs as well as the flexibility and reconfigurability of general-purpose microprocessors. Significant effort from the research community has focused on techniques to move this reconfigurability from a rapid application development tool to a run-time tool. This requires the ability to change the hardware design while the application is executing and is known as Run-Time Reconfiguration (RTR). Widespread acceptance of run-time reconfigurable custom computing depends upon the existence of high-level automated design tools. Such tools must reduce the designers effort to port applications between different platforms as the architecture, hardware, and software evolves. A Java implementation of a high-level application framework, called Janus, is presented here. In this environment, developers create Java classes that describe the structural behavior of an application. The framework allows hardware and software modules to be freely mixed and interchanged. A compilation phase of the development process analyzes the structure of the application and adapts it to the target platform. Janus is capable of structuring the run-time behavior of an application to take advantage of the memory and computational resources available.

Keywords: RTR, run-time reconfiguration, CCM, configurable computing, FPGA, automated design

1. INTRODUCTION

Typical computational applications for field programmable gate arrays (FPGAs) have traditionally been static. In such applications, the FPGAs are programmed once at system startup and remain in that one configuration. One obvious limitation of this approach is that there are always computations that require more resources than are available on a fixed configuration of hardware. Limited hardware resources can be overcome by taking advantage of the ability to reprogram FPGAs using runtime reconfiguration. Applications that require more resources than are available on a CCM can use a divide and conquer technique, where an application is broken into smaller computations, each of which can execute on a CCM individually. Over time the configurations change in order to complete the application execution. The first step of an RTR application performs computations and stores the results into memory. The array is then reconfigured for the next step, which can compute results based on the results of previous computations. This process continues until all the computations have been performed[5,10]. Using these techniques, the primary limitation on the size of problem that an RTR application can solve is the size of the memory required to store intermediate results[4].

Typical design tools are good for targeting single FPGAs with a static design. CCM tools often differ considerably due to the wide variety of specialized configurations. Interconnections between the components, memory sizes, even the components themselves vary widely between platforms. Application designers retarget their work to individual platforms to deal with this variation. This results in optimized custom applications but design reuse is difficult. RTR CCM tools have additional design issues. They must manage the reconfiguration process and the configuration state over time. Reconfiguration cost is expensive in current devices. Tools must be designed to minimize this overhead.

This paper presents a tool that attempts to address some of these problems for the application developer. The Janus framework makes the following contributions to RTR CCM application design flow:

- a demonstration of a unified design framework for developing platform independent RTR CCM applications,
- an object-oriented approach that allows developers to build applications without knowledge of the architecture, compilation methodology or run-time implementation of the target CCM,
- a stage-based design paradigm that provides an intuitive approach to designing RTR applications, and

* Correspondence: Email: {dlehn,rhudson,athanas}@vt.edu, WWW: http://www.ccm.ece.vt.edu/

- an evaluation of the successes and shortcomings of the approach.

This paper provides a brief overview of the Janus RTR development environment. Section 2 provides background to related RTR tools and hardware. Section 3 presents an examination of the Janus application development framework, along with a discussion of the object-oriented structure of the tools. Section 4 presents the implementation of an image interpolation application as a case study. Section 5 draws some conclusions about the success of the tools, points out some shortcomings, and discusses the future of the Janus tool set.

2. BACKGROUND

The primary advantage of a CCM is that it is configurable. A microprocessor has a limited number of specific computational units. It may have one multiplier, one adder and a barrel shifter. If an application never uses the barrel shifter, the silicon that implements the barrel shifter goes to waste. If a CCM has an application that needs ten adders, the CCM can be configured to provide them. If an application needs four multipliers, the FPGAs can be configured to provide them. Microprocessors often have single data paths to memory. CCMs are often designed with each FPGA having it's own memory port. This provides a larger aggregate data path.

These features of CCMs provide the means to gain the significant performance increases demonstrated in the literature. Any successful development paradigm for RTR CCMs must allow applications to exploit these underlying properties of the target architecture. The Janus tools present an abstraction of the underlying CCM that allows applications to be automatically adapted to take advantage of the resources available on a target platform. The developer creates sets of operations, called stages, that need to be performed in a particular order and the tools automatically determine how to schedule parallel instances of them on the target hardware.

The current state of the art requires the application developer to build all of the components of a RTR design separately. FPGA configurations are designed with traditional tools. The user interface is designed using the designer's choice of a programming language. The user interface controls the CCM hardware using a proprietary application-programming interface supplied by the vendor. The developer manages all of the interfaces between the components. Since there are no standard design representations for these interfaces, the resulting interfaces are created ad hoc on a per application basis[1].

2.1. Software

Software design tools for CCMs have taken two main approaches. One method with a large body of research is to attempt to map high-level languages such as C to hardware constructs[11,16]. Other tools such as SPARCS and CHAMPION, described below, use specialized dataflow and control graphs to represent user designs. In either case, automated spatial partitioning of designs for FPGAs is a NP-complete problem whose heuristic solutions are extremely sensitive to the topology of the computational dataflow graph and the target CCM[20].

Projects such as the Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems (SPARCS)[15] take a design specification at the behavioral level in the form of a task graph. Temporal partitioning is used to temporally divide and schedule the tasks on the target architecture. Spatial partitioning is used to map the tasks to individual FPGAs in a multi-FPGA CCM. A high-level synthesis tool creates an efficient register-transfer level design for each set of FPGA tasks. Commercial tools are used to perform the logic synthesis, placement, and routing for each FPGA configuration.

The University of Tennessee presents a design environment called CHAMPION[13]. It uses the Khoros Cantata visual programming software as its design front end. Users are provided with a set of image processing primitives that can be composed to perform image-processing operations. The CHAMPION software provides automatic retiming to match paths through the dataflow and performs multi-FPGA design partitioning. It is unclear how well the CHAMPION system abstracts the underlying hardware system and allows the user to function without detailed knowledge of the architecture. The system does provide an automatic partitioner, that allows the user to function without knowledge of the architecture, but no data has been presented on the tool's ability to map arbitrary designs to varying hardware architectures.

---

[1] Recently, attempts have been made to increase the portability of host code for configurable computing platforms. One such proposal is a standard interface for distributed adaptive computing systems[12].

2.2. Hardware

Reconfiguration time is a considerable design issue in many RTR applications. Typical FPGAs have a significant configuration time[21]. As the frequency of reconfiguration increases it is possible to overwhelm the computation time with configuration overhead. This situation should be avoided so that the application can effectively utilize its resources.

Several different strategies for handling the run-time overhead of reconfiguring large FPGAs have emerged in the literature. One very promising strategy is the multiple-context FPGA[19], originated by Ong[14]. The Context Switching Reconfigurable Computing (CSRC) FPGA[18] is an example of such a device. Additional configuration memory is used to support multiple configurations. In the CSRC device, four separate configurations can be maintained on chip simultaneously with one active configuration controlling the configurable units. The CSRC device is capable of changing its configuration, e.g. its context, on a single clock edge. In addition, the CSRC device is capable of loading new configurations while another configuration is actively computing.

For simple applications, with few stages, multi-context devices provide a perfect environment. If a multi-context device supports four on-board configurations, then up to four stages of an application can be present on the device at the same time and the overhead for changing configurations is on the order of a clock period. When the number of stages in an application exceeds the number of configurations available on the multi-context device, new configurations need to be loaded onto the device during execution. The ability to load new configurations onto the device in the background, while computation is going on in the foreground, drastically reduces the apparent reconfiguration overhead.

2.3. Just Another Hardware Description Language

Just another Hardware Description Language (JHDL)[1] is an object-oriented hardware abstraction that allows hardware developers to create FPGA hardware with a Java development system. Java was chosen because it is an intuitive, easy to learn object-oriented language[7].

The goals of JHDL are to provide a unified way for hardware developers to describe the structure of digital logic circuits, guide placement of the circuits on actual FPGAs, and execute the design on the target platform. JHDL allows users to describe circuit construction at the same level of abstraction as purely structural VHDL. In addition, the JHDL developer has all of the constructs available to a high-level software programmer at their disposal for creating structural models. The abstraction used to create logic primitives is also independent of the target device. This allows designs to be portable between FPGAs. The Java language has a standardized interface for native code called the Java Native Interface (JNI)[8]. This allows Java programmers to provide portable interfaces to CCM platforms. While this could be done in VHDL the mechanisms for implementing native extensions are not widely used and are generally proprietary to particular VHDL vendors.

JHDL provides a simulation environment that is tailored to FPGA development. The JHDL simulation environment allows users to perform software only simulations of their structural hardware, but with the press of a button, the simulator can run the same application on the hardware platform itself. Readback capabilities of target architectures allow the simulation tools to perform comparisons between the simulated and actual values of signals in the system. Memory can be initialized and interrogated during simulation. Schematics of the designs can be called up with the current simulation or actual hardware signal values.

JHDL was chosen as the hardware description paradigm for this work because of its inherent flexibility for the application developer and the tool developer. It allows the application developer immense flexibility in terms of producing reusable adaptable hardware modules. In addition, it allows the tool developer to extend JHDL classes and utilize existing interfaces to provide tool integration that would be impossible using VHDL. The Janus tools also have support for integrated simulation with JHDL. The JHDL and Janus simulation environments can work together to provide a complete system simulator.

3. JANUS

RTR application design using current tools is an ad hoc process. Commercial tools that support the RTR design flow do not exist. RTR developers currently must maintain a delicate balance between several hardware designs, software control of the CCM platform, and run-time interaction between the hardware and the software. Janus tools present a unified design flow that allows the hardware, software and interaction between them to be completely described using Java. Computational hardware is specified in Java using JHDL. Software components of an application are built within the Janus framework using

the Java language and any of its ancillary programming interfaces. The Janus tools provide hardware access. Interaction with the CCM vendor's application programming interface (API) is performed through JNI.

Since the architecture of RTR CCM platforms can vary widely from platform to platform, Janus tools provide an architecturally independent view of RTR CCM technology to the application developer. Using the Janus framework, application developers can work without detailed knowledge of the target architecture. The developer can recompile an application to any target architecture supported by the system. This provides the ability to port designs between different CCM architectures. Furthermore, since the life cycle of configurable computing platforms is relatively short and the development cycle for custom software tools for a platform is relatively long, a useful design framework must support architectures that have not yet been developed.

Prior design attempts conducted as part of the exploratory work[10] for this framework had encountered numerous computationally challenging problems. This prior research indicated that an attempt to create a set of RTR design tools whose input specification was a high-level language would result in the need to solve multiple NP-complete sub-problems. The initial effort indicated that a greater understanding of the overall design process was required before such a task should even be attempted. The initial goal of the Janus tool set is to provide a useful, complete design environment that avoids solving any NP-complete problems. Based on the algorithms for high-level language translation presented in the literature, avoidance of NP-complete problems precludes the adoption of a high-level language as the input specification to the Janus tools.

3.1. Janus Design Goals

Based results reported in the literature, previous experience with RTR design automation and a general pragmatic approach to tool development, the design of the Janus tools is governed by these goals.

3.1.1. Unified RTR Application Description

A CCM-accelerator architecture is composed of two connected halves: the host side and the CCM side. The host side is generally a microprocessor-based workstation. The CCM side is a computational array of FPGAs. The design efforts for each side are often completely separate. Janus' primary design goal is to allow RTR application developers to specify their designs using a single design flow. One of the primary reasons for selecting JHDL as the structural representation for the CCM side is to place the circuit design in the same design space as the host side. VHDL is not a convenient medium to write user interfaces or to provide database access, but Java is. By bridging the gap between the host side and the CCM side, the Janus tools not only provide a unified design description for an application they provide a means for the two designs to interact.

3.1.2. Time-Multiplexed Virtual Hardware

The size of the problem that many applications can address is limited by the amount of configurable resources available on the CCM. Run-time reconfigurable applications do not have this fundamental limitation. Temporally multiplexing the FPGA resources can significantly increase the scope of designs. The Janus tools provide an abstraction of this process that allows users to easily specify reconfigurations within the unified design.

3.1.3. Architecture Independence

The Janus tools are not bound to a particular architecture. There are a wide variety of CCM platforms in use by the research community. The advantages of being able to port applications between different CCMs are reason enough to create an architecture independent system. Therefore, it is one of the goals of the Janus tools to be able to design applications that can be automatically targeted to different CCMs. Caution must be taken, however, to insure that in making applications portable, their ability to exploit the advantages of the underlying architecture is not hindered. Applications built with the Janus tools must automatically adapt to exploit their target architecture.

3.2. Application Design

A view of the structure of the Janus abstraction itself is presented in Figure 3-1. Figure 3-1 is an UML[17] package diagram describing the division of the classes in the Janus system among the various packages and the dependencies between the various Janus packages. Also diagramed are the dependencies between user applications, shown in the **apps** package, and the Janus packages.

The **hardware** package contains classes that provide abstractions of hardware architectures, processing elements and memory. The **runtime** package provides data structures for describing the sequence of run-time events and interfaces to the hardware architecture for executing them. The **schedulers** package contains the compile-time code for translating the user's description of an application into a sequence of run-time events. Note that user applications are not dependent upon the **hardware**, **runtime**, or **schedulers** packages. Applications are independent of the underlying hardware model, compiler components, and run-time structures.

The **stages** package defines the interfaces to the components that the application developer will use to model their application. The **base** package contains the application framework that the user extends to create their own application. It defines the interface between Janus' compiler and run-time system and the user's application. Finally, the **app** package contains the code for creating Janus' user interface. User interaction can be provided as a graphical user interface (GUI), a command line interface (CLI), or even as a network service.

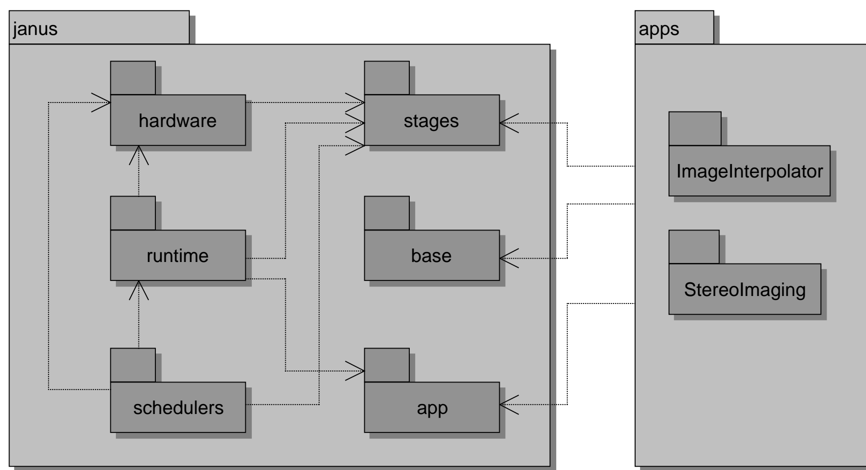

Figure 3-1: Package Diagram for Janus Abstraction.

The developer must create an application model that defines the sequencing of the computation and the nature of the computation itself. Software stages can be used to prototype portions of the hardware design or for application specific host processing. Portions of the design that require hardware-accelerated operations must be designed and implemented. Some type of user interface must be built. Finally, the finished application must be compiled so that it can be executed on a target platform.

3.3. Creating the Application Model

The **stage** package defines the basic structures the user will utilize while creating an application model. The design of the **stage** package is highly influenced by previous work on a more ambitious toolset[10]. Based on these observations and others made from looking at applications like wavelet compression[3] and SAR imaging[2], the approach taken in the Janus tools is based on *stages of computation*. This approach allows applications to be described in the natural divisions that occur between significant reorderings of data. Intermediate results are retired to memory between each stage. This allows the following stage to simply read its data from memory in the order that it requires. No explicit data reordering is necessary.

Given this computational model, the Janus abstraction defines a set of container objects called stages. The stage containers allow the application developer to group together the computational operations that form an application. A class diagram of the **stages** package is given in Figure 3-2.

The package defines an interface, called ***Stage***, and three implementations of that interface, **StageSoftware**, **StageOrdered**, and **StageUnordered**. These three classes, and another interface called **Operation**, form the basic building blocks that an application developer uses to model their computation. The developer describes their computation by creating a hierarchy of computational stages. The **StageOrdered** object maintains an ordered list of references to other ***Stage*** objects. Each

StageOrdered object has at least one reference to another ***Stage***. This allows the user to create a tree of ***Stage*** objects. The leaves of this tree structure are instances of the **StageSoftware** and **StageUnordered** classes. The **StageSoftware** class allows the user to create software implementations of computational stages. **StageUnordered** is a collection class that allows the user to build up a set of hardware-accelerated operations that need to be performed during a given stage. These hardware-accelerated operations are implementations of the **Operation** interface.

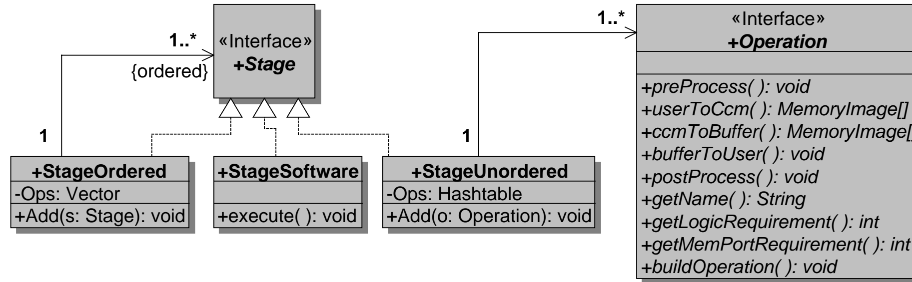

Figure 3-2: Class Diagram of the Stages Package.

Each of the leaf nodes in an application model describes a set of events that will occur during run time. In the case of **StageSoftware** objects, this run-time behavior is specified as a concrete implementation of **StageSoftware**'s *execute()* method. This ability to implement software stages provides the ability to create a rapid prototype of a system before implementing hardware-accelerated objects. As hardware versions of each stage are created, they can be easily substituted for their software implementations. It also provides the ability to incorporate software implementations of algorithms that are not well suited to CCM development.

StageUnordered objects contain an unordered set of hardware-accelerated operations. Operations are entities that span the boundary between the host computer and the CCM. These operations are implementations of the **Operation** interface shown in Figure 3-2. Operations can be thought of as hardware accelerated processes. All operations are associated with at least one memory interface. Operations read their input data through a memory interface and retire their results back through the same interface or possibly another memory interface. Operations that are grouped together in a **StageUnordered** collection should all be computationally independent of each other. If two operations are computationally dependent upon one another, they should be relegated to separate **StageUnordered** objects that are sequenced in the appropriate order by a higher-level **StageOrdered** object.

The software portion of an operation describes how data is transferred between the host-side application and the processing elements on the RTR CCM. A number of methods are used to do setup, cleanup, and host side processing of data. Extra memory copying is avoided where possible but applications can do custom processing when necessary to perform tasks such as memory reordering.

The *buildOperation()* method provides the hardware description of an operation. The *buildOperation()* method uses JHDL to build a structural model of the operation that can be netlisted and placed-and-routed on the target FPGA. The application code is free to do whatever is needed during this build. It can use parameters from data loaded through a user interface or otherwise adapt to its environment. Current place and route tools take a significant amount of time to run. This does not allow "instant" execution of an application. In the current tool you must run an initial pass over the application to write out netlists. Then the netlists must be turned into bit streams with external tools. A second run is then necessary to execute the application. When JHDL or other tools allow real time generation of FPGA configurations, Janus will be able to take advantage of this capability.

3.4. Hardware Model

Figure 3-3 presents a class diagram that depicts the hardware abstraction. The abstract class ***HardwareArchitecture*** provides a software interface to the target RTR CCM. It is composed of one or more associated ***ProcessingElements***.

A concrete implementation of a ***HardwareArchitecture*** models the behavior of a specific hardware platform. For instance, the **WildForceXL_4062** class models the behavior of an Annapolis Micro Systems WILDFORCE XL populated with XC4062s. The **WF4_4062** class models the processing elements available in the **WildForceXL_4062** class.

HardwareArchitectures and ***ProcessingElements*** both have methods that allow the compiler to query information about the available hardware resources and to assign hardware operations to memory ports on individual processing elements. These methods allow the compiler to interact with the architecture and the processing elements to determine which operation should be scheduled on which element. The compiler acts within the constraints of available configurable logic and memory ports.

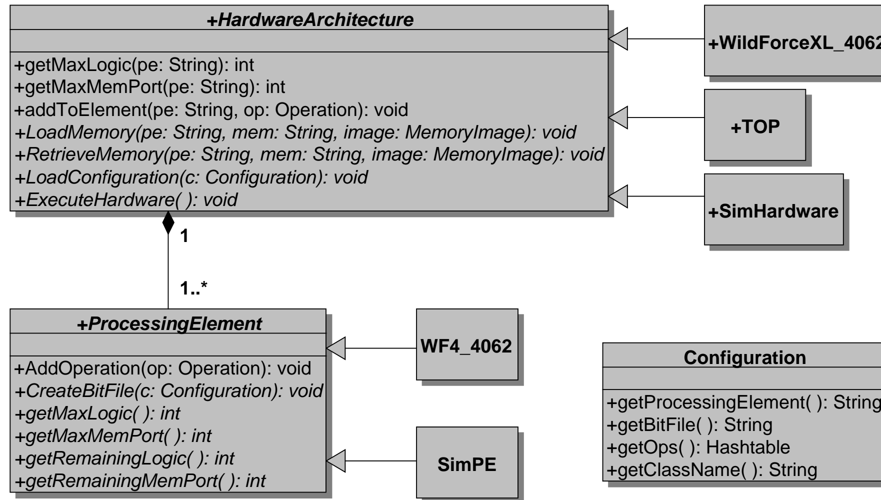

Figure 3-3: Janus Hardware Abstraction.

Three concrete implementations of the Janus ***HardwareArchitecture*** class have been implemented as part of this research. One allows Janus applications to target an Annapolis Micro Systems WILDFORCE populated with XC4062XL FPGAs. Another targets Virginia Tech's Tower of Power[12] (TOP) computing engine. The tower of power computing engine is a collection of sixteen Dell workstations networked together with Myrinet. Each workstation has a WILDFORCE board. Communication between the host application and the various TOP processing elements is performed over the network. Data must not only be moved from host to CCM, but from the central control point, to another computer and from that computer to the CCM. All of this extra control is transparent to the Janus developer. The final architecture is a software only simulation mode. This takes advantage of the JHDL simulation environment to preset an arbitrary hardware configuration to the application. The difference between targeting the different platforms is a single menu selection during the compilation process.

3.5. Scheduling

The ***Scheduler*** **t**akes a reference to an application model and a ***HardwareArchitecture*** and produces a **RunTimeScript**. A **RunTimeScript** is an ordered list of events that need to occur at run time during the execution of the application. The run-time events are represented by the various ***RunTimeEvent*** classes:

- **LoadConfiguration** – loads a set of hardware configurations onto the hardware processing elements,
- **FetchMemory** - copies data from the host application to a specific memory location on the CCM,
- **RetireMemory** – copies data from one of the memories on the CCM to a location in the host application,
- **HardwareInterlude** –executes some subset of operations from a **StageUnordered** on the CCM. This involves starting the clock on the hardware platform, waiting for a signal from the CCM indicating that the computation has been completed, and stopping the clock,
- **SoftwareInterlude** – calls the *execute()* method of a corresponding **StageSoftware**.

The scheduler can perform an arbitrarily complex algorithm to best map stages and operations to the target platform. The current version in Janus takes a simple approach. First the stage hierarchy of the application is flattened to leave an ordered list of unordered stages. Due to current tool constraints each of these unordered stages will have operations of similar type. This puts hardware configuration events at the boundary of each unordered stage. Next the scheduler iterates through the list of stages to create a **RunTimeScript**. For each stage the scheduler will add the appropriate **RunTimeEvents** to complete each operation. The scheduler uses the ***HardwareArchitecture*** to determine resource availability. The available resources are reduced as operations are assigned to the hardware. Operations must meat the logic and memory port limitations of the

remaining hardware. Operations of the same type are grouped together to minimize hardware configuration events. When no more operations can be executed due to limited resources or a need to reconfigure the device the scheduler will add hardware configuration events. Each operation added to the **RunTimeScript** has both memory events and an associated **HardwareInterlude**.

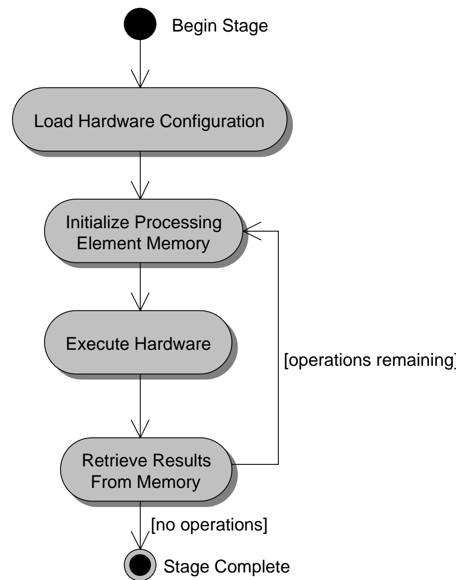

Figure 3-4: Activity Diagram for Compute Cycle.

3.6. Application Execution

The Janus abstraction specifies a run-time environment that executes the series of ***RunTimeEvents*** generated by the compiler. Figure 3-4 shows how the execution is conceptually performed on a per-unordered stage basis. When the user requests the application to run, it iterates through the entire list of ***RunTimeEvents***. Alternatively, the user can proceed through the application one run-time event at a time using single step options. During execution the application's own graphical user interface is the primary point of interaction with the user. The application can provide progress indicators, intermediate results, and parameter adjustment controls through software interludes.

4. EXAMPLE APPLICATION

The B-Spline based image interpolation algorithm[6] provides a good example of how to exploit the feature of the Janus framework. The interpolation procedure increases the resolution of an image by deriving new pixels between the existing ones. The approach presented accomplishes this using 2-5-2 splines. The Image Interpolator breaks down into four stages: inverse filter on the rows, inverse filter on the columns, Fast-Spline Transform on the rows, and Fast-Spline Transform on the columns. Figure 4-1 illustrates the computational structure of the image interpolation process. Each gray circle represents either a row or column. The rows and columns of each filter are separable and thus mutually dependent on each other. All the columns must be computed before any of the rows can be computed or vice versa. In addition, the diagram indicates the FST is completely dependent on the results of the IF. All the rows and columns of the IF must be computed before any of the rows or columns of the FST can be computed.

The diagram also exposes inherent parallelism in the computation. For both the IF and the FST, each row is independent of all the other rows and each column is independent of all the other columns. This allows simultaneous calculation of any number of rows or any number of columns.

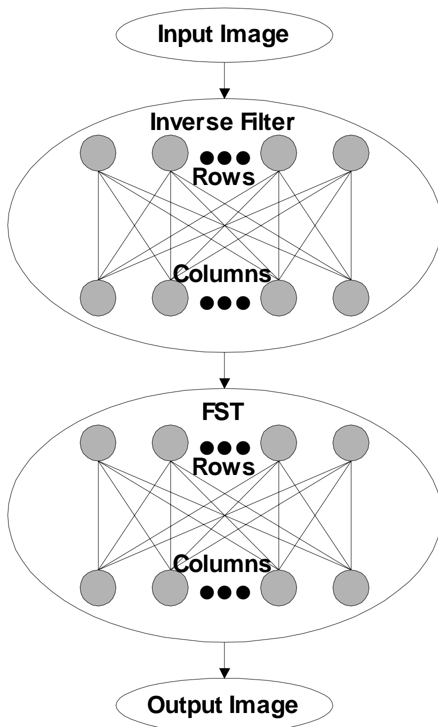

Figure 4-1: Computational Structure of Image Interpolation.

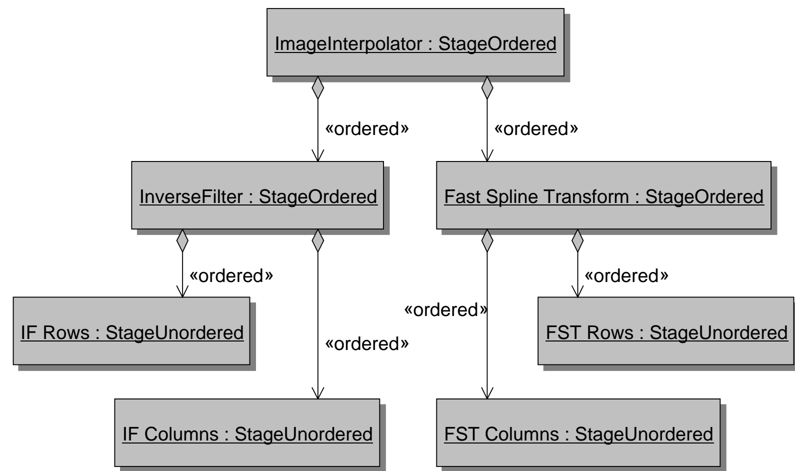

Figure 4-2: Object Diagram of Image Interpolator.

Figure 4-2 is an object diagram showing an application model of the image interpolator. The structure is an ordered hierarchy. The leaf **StageUnordered** objects contain the Operations. Since all of the hierarchical objects maintain an ordering of their aggregates, there is an overall ordering maintained on the leaf nodes. If we assume that the association markers in **Error! Reference source not found.** are ordered spatially from left to right in the diagram, the overall order of the leaf nodes is **IF Rows**, **IF Columns**, **FST Columns** and **FST Rows**. The overall ordering defines the sequence of events that will happen at run time. Java code is written to construct this structure for an input image. The stages are filled with operations representing low level hardware logic.

Further partitioning might be necessary if the FPGA resources required by the computations in each of these partitions exceed those available on the platform. Alternatively, if the resources required represent a fraction of the available resources, then the required hardware may be duplicated within a partition to take advantage of the parallelism discovered in the computational structure. The exact choice of partitions and parallelism cannot be made until some implementation decisions have been reached.

The image interpolator's graphical user interface is shown in Figure 4-4. It has a **File** menu that allows the user to select an input image for interpolation and a **Test** menu that allows selection of various test patterns. Some image zoom controls are provided to allow detailed examination of image features. The interface has two progress bars that monitor report the status toward completion of individual stages and the overall process respectively. The lower portion of the interface displays the initial image, output image and, optionally, intermediate images produced between each stage.

Figure 4-3 shows the Image Interpolator after it has been loaded, built, and scheduled. The application designer has visual feedback to the design they built and how it will be executed. This GUI also has an interface to execute the application in one step or one event at a time.

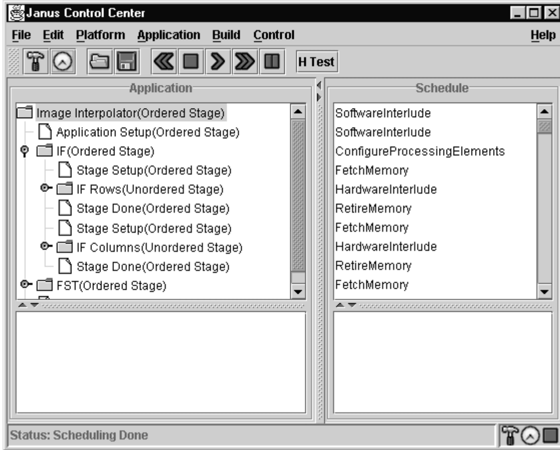

Figure 4-3: Janus Control Center.

Figure 4-4: Graphical User Interface for the Image Interpolator.

5. RESULTS

There are two major contributors to overall runtime: reconfiguration and memory transfers. When data sets are small and memory transfers are short the reconfiguration time has been more than 80% of the total time. Configuration time is a constant for a given platform. Increasing the computation time can make the ratio of computation time to configuration time reasonable. Using larger data sets to increase the computation time is not always the best solution. It can quickly lead to memory bandwidth problems. In the Image Interpolator the data produced from a computation increases exponentially with the image size being processed. With extremely large images the configuration and computation time can be insignificant compared with the time due to memory bandwidth alone.

Reconfiguration overhead can also become a serious issue due to platform characteristics. The WILDFORCE board does not allow parallel configurations. In the case where all the FPGAs are programmed with the same configuration this is a considerable loss. The constant reconfiguration time will increase linearly with the number of FPGAs used. This is somewhat balanced out for ideal applications that can reduce computation time linearly with the number of processors used. Targeting Janus applications to a platform with many networked nodes, such as the Virginia Tech Tower of Power, made the need for distributed configuration caching apparent. Inter-host bandwidth and latency issues overwhelmed any benefit of distributed computation.

Many lessons were learned from the aborted tool design described by Hudson[10]. One of the most important was that the Janus tools are exploring a new area of design space. So much is unknown about the design of RTR applications that it is easy to become engaged in an effort to solve problems that, after much effort, turn out to be unrelated to the core design issues. The initial version of the tools, documented here, provides a fully functional working design system. This fully functional system was made possible by a steadfast belief that a working tool would provide a stepping stone to understanding which further problems should be explored. Along the way, opportunities to solve many different NP-complete problems were put aside until a firmer understanding could be developed of which problems were worth solving.

The design and implementation of the Janus tools has attempted to examine the RTR application design process with a pragmatic approach. The first and foremost goal was to create a useful design framework forming a basis for further automation research. The primary purpose of the Janus tools is to study the nature of run-time reconfigurable applications and determine what sort of automation is useful to the design flow. The examples studied showed that some classes of applications can be rapidly developed and easily retargeted. Further research is needed to determine how to optimized the results of applications using the Janus framework.

A benefit was seen by the use of a fully functional programming language as unified representation of the application. The unified representation allows the user to create hardware that interacts with portions of the design that were previously

defined by a completely separate design flow. The fully functional programming language gives the user the ability to describe hardware modules that can adapt to their environment. This flexibility opens a wide opportunity for optimizations and future RTR exploration.

6. REFERENCES

1. Peter Bellows and Brad Hutchings, "JHDL - An HDL for Reconfigurable Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 175-184, April 1998.
2. D. Castanon, M. Petersen, S. Ponnuswamy, B. VanVoorst and C. Nanavati, "Constant False Alarm Rate Benchmark Specification," *Technical Information Report*, Honeywell Technology Center, March 1998.
3. A. Cohen, I. Daubechies, J. Feauveau, "Biorthogonal Bases of Compactly Supported Wavelets," *Comm. Pure Appl. Math.*, vol. 45, 1992.
4. J.G. Eldredge and B.L. Hutchings, "Run-Time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-Based FPGAs," *Journal of VLSI Signal Processin*g, Volume 12, pp. 67-86, 1996.
5. J.G. Eldredge and B.L. Hutchings, "RRANN: The Run-Time Reconfiguration Artificial Neural Network," *Custom Integrated Circuits Conferenc*e, pp. 77-80, San Diego, California, May 1994.
6. Leonard Ferrari and Jae Park, "An Efficient Spline Basis for Multi-Dimensional Applications: Image-Interpolation," *IEEE International Symposium on Circuits and Systems*, vol. 1, pp. 757-760, 1997.
7. James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*, Addison-Wesley Publishing Company, 1996.
8. Robert Gordon, *Essential JNI: Java Native Interface*, Prentice Hall Computer Books, 1996.
9. Rhett Hudson, David Lehn, and Peter Athanas, "A Run-Time Reconfigurable Engine for Image Interpolation," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 88-95, April 1996.
10. Rhett Hudson, David Lehn, Jason Hess, James Atwell, David Moye, Ken Shiring, and Peter Athanas, "Spatio-Temporal Partitioning of Computational Structures onto Configurable Computing Machines," *SPIE Proceedings*, Vol. 3526, p. 62-71, October 1998.
11. Tsuyoshi Isshiki and Wayne Wei-Ming Dai, "Bit-Serial Pipeline Synthesis for Multi-FPGA Systems with C++ Design Capture," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 38-47, April 1996.
12. Mark Jones, Luke Scharf, Jonathan Scott, Chris Twaddle, Matthew Yaconis, Kuan Yao, Peter Athanas, and Brian Schott, "Implementing an API for Distributed Adaptive Computing Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1999.
13. S. Natarajan, B. Levine, C. Tan, D. Newport and D. Bouldin, "Automatic Mapping of Khoros-based Applications to Adaptive Computing Systems," *Proceedings 1999 Military and Aerospace Applications of Programmable Devices and Technologies International Conference (MAPLD)*, pp. 101-107, 1999.
14. R. Ong, *Programmable Logic Device Which Stores More Than One Configuration and Means for Switching Configurations*, U.S. Patent 5,426,378, 1995.
15. I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, R.Vemuri, "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures," *5th Reconfigurable Architectures Workshop (RAW'98)*, Orlando, Florida, USA, March 30, 1998
16. James Peterson, R. Brendan O'Connor and Peter Athanas, "Scheduling and Partitioning of ANSI-C Programs onto Multi-FPGA CCM Architectures," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 178-179, April 1996.
17. James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley Publishing Company, 1998.
18. S. Scalera, J. Vazquez, "The Design and Implementation of a Context Switching FPGA", *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 78-85, April 1998.
19. Steve Trimberger, Dean Carberry, Anders Johnson and Jennifer Wong, "A Time-Multiplexed FPGA," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 22-28, April 1997.
20. Gopalakrishnan Vijayan, "Partitioning Logic on Graph Structures to Minimize Routing Cost," *IEEE Transactions on Computer-Aided Design*, pp. 1326-1334, vol. 9, no. 12, December 1990.
21. Xilinx Inc., *XC4000E and XC4000X Series Field Programmable Gate Arrays*, Nov. 10, 1997.