

A development environment for configurable computing

B. Boysen, N. DeBardeleben, K. M. Hazelwood, W. B. Ligon III,
R. Sass, D. C. Stanzione,jr., K. D. Underwood

Parallel Architecture Research Lab
Holcombe Department of Electrical and Computer Engineering
Clemson University
105 Riggs Hall
Clemson, SC 29634-0915

ABSTRACT

As FPGA density increases, so does the potential for configurable computing machines. Unfortunately, the larger designs which take advantage of the higher densities require much more effort and longer design cycles, making it even less likely to appeal to users outside the field of configurable computing. To combat this problem, we present the Reconfigurable Computing Application Development Environment (RCADE). The goals of RCADE are to produce high performance applications, to make FPGA design more accessible to those who are not hardware engineers, to shorten the design lifecycle, and to ease the process of migration from one platform to another. Here, we discuss the environment architecture, the current set of agents, and other agents to be developed.

Keywords: design environments, configurable computing, FPGA components, field programmable gate arrays

1. INTRODUCTION

As Field Programmable Gate Array (FPGA) density increases, so does the potential for configurable computing machines (CCMs). Unfortunately, the tools to develop applications for CCMs are not maturing as quickly as the technology. This problem manifests itself in several areas: initial application development is complex and time consuming, and migration from one generation to the next requires starting over almost from scratch. These problems leave many unwilling to migrate their applications from a software implementation to hardware. We propose an environment called the Reconfigurable Computing Application Development Environment (RCADE) for the development of CCM applications which will offer higher performance than software solutions, allow non-experts in FPGA design to develop applications, will simplify the design and maintenance process for those who are experts, and ease the process of migration from CCM platform to the next.

1.1. Problem description

The primary benefit of using FPGA technology in a CCM system is the increase in performance. Rather than implementing applications in software, FPGA technology allows the execution of applications at hardware speeds without the inordinately high cost of creating custom silicon.

To gain these hardware speeds, using CCMs presents a problem that is as intricate as hardware design. This presents a barrier which has to date made CCMs inaccessible to many users. Our environment seeks to remove this barrier and make CCM available to a larger user community by allowing the user to operate at a higher level of abstraction than traditional hardware design. The user enters their initial design in a platform-independent, algorithm-oriented way.

Even if a user spends the time to learn the intricacies of hardware design, it is difficult to verify and debug an application. Compared to a traditional software approach, the debugging process suddenly expands from merely testing and repairing an application implemented in C to debugging both the application and the individual operations used. Simulation helps logically debug the application, but simulation is slow and often does not deal with timing issues. The component-based approach used in our environment hides many of the timing and control issues associated with interconnecting the components. This approach serves to greatly reduce the time required to debug or to update an existing application.

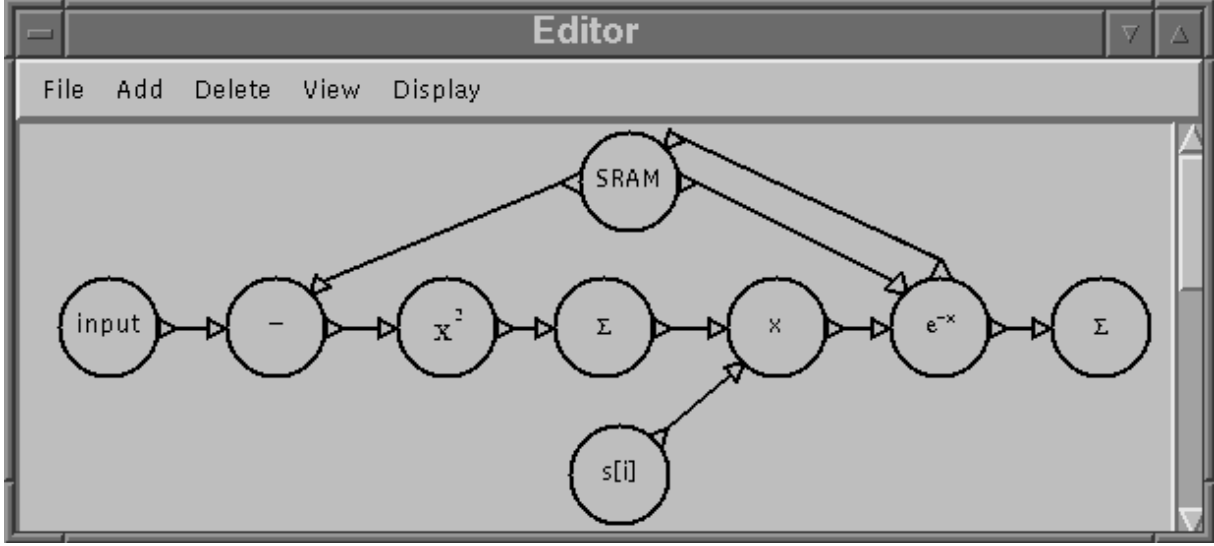


Figure 1. The PNN algorithm as represented in ADF and displayed by the editor.

The RCADE environment also addresses the problem of portability among CCM platforms. In traditional FPGA system design, each new generation of device and each new CCM board architecture required in essence throwing out old designs and starting a new design from scratch. Technology is progressing in these areas so quickly in fact, that the time from release to obsolescence of many devices is shorter than the design time for sophisticated applications! Our environment addresses this problem by allowing initial design entry in a platform independent representation, and by building implementations on a standard set of components. Migrating the components from system to system will allow all the applications based on them to migrate more easily.

1.2. Our approach

To address these issues, we use a technology independent description of a user design which contains all pertinent information about the algorithm in a data flow graph. RCADE is built upon the CECAAD base, which provides a format for describing the graph and a model for independent agents to interact with the graph. The agents encapsulate the expertise needed for successful design of high performance CCM applications.

The nodes in the data flow graph represent abstract operations in the design and are created by agents in the front end. A set of analysis agents activated by the user refines and enhances the data flow graph by adding parameters which constrain the final implementation. In the final step, back-end agents map the design to components.

Components in our environment are efficiently implemented blocks of logic that conform to a well-defined interface and correspond to operations in the design. In addition, our component-based approach hides many of the timing and control issues typically associated with interconnecting components. This serves to greatly reduce the time required to debug a new application or update an existing application.

To illustrate this process, we include the following example. The Probabilistic Neural Net (PNN) is an image classification algorithm for satellite telemetry.^{1,2} The algorithm has N classes, each with P_k vectors ($1 \leq k \leq N$). An important computation in the algorithm is

$$f(\mathbf{X}|S_k) = \frac{K}{P_k} \sum_{i=1}^{P_k} \exp \left[-\frac{(\mathbf{X} - \mathbf{W}_{ki})^T (\mathbf{X} - \mathbf{W}_{ki})}{2\sigma^2} \right] \quad (1)$$

where $K = (2\pi)^{-d/2} \sigma^{-d}$, \mathbf{W}_{ki} is the i^{th} training vector from the k^{th} class, and d is the dimension of each vector. (The σ is a smoothing parameter.) One of the N classes is assigned to every pixel by computing values from Equation 1 and selecting the class with the highest resulting score. Fig. 1 shows an initial abstract representation of the algorithm. By invoking a series of agents, this graph is transformed into a working hardware design.

Our goal is to create an environment for creating high performance configurable computing applications which shortens the design cycle for experienced users, makes migration from one CCM platform to another simpler, and allows non-experts access to the performance of dedicated hardware, though an expert with a longer design time may produce a smaller, faster solution. To achieve this, we based RCADE on a technology independent description of an algorithm. To this, we added agents which assist the user in describing the application and enhancing that description. Agents are also provided to map the description to a component-based design approach and create the final application.

1.3. Overview

The remainder of this paper is organized as follows. In Section 2, we discuss the infrastructure, specifically the organization of the system and our component-based approach. Several important agents are described in Section 3. There are many researchers actively pursuing ways of using FPGAs to do computation. In Section 4, we show how our project's goals and work are related to other projects in the area. Finally, we conclude with future work and a summary of our work to date in Section 5.

2. SYSTEM ARCHITECTURE

The fundamental structure of RCADE is shown in Fig. 2. At the core of the environment is the design, which is initially represented in an implementation independent format. A number of different tools, known as *agents* have concurrent access to the design. Agents perform a variety of functions, including aiding the user in initially describing their algorithms, guiding the selection of suitable components for the target CCM, and analysis and optimization of the design. Agents are distinct entities that utilize to a well-defined interface to the shared design. It is easy to extend or modify RCADE by adding new agents. The key to the success of the environment is that all the agents share access to the same representation of the design. This allows the user to initially describe the algorithm with the front-end agents of the environment at a high level of abstraction. These design descriptions may be re-used by various back-end agents to re-target the applications to ever-changing CCM architectures. Rather than attempting to implement the design at a gate level for each target architecture, the back-end agents use a component-based approach to implement the applications on the FPGAs. In this way, changing from one architecture to another requires a new set of components, rather than a new design.

The following sections describe in detail the infrastructure used to construct the environment, some of the agents that have been created to meet the goals of the environment, and the advantages and implementation of our component based approach.

2.1. Environment infrastructure

The reconfigurable computing environment described here is built upon the Clemson Environment for Computer-Aided Application Design (CECAAD).³ CECAAD is an effort to provide a base environment which can be extended to create environments for a variety of technologies and application domains. CECAAD provides support for the agent collaboration model, as well as a common format for describing the dataflow graphs which represent the design. The format used is the *Algorithm Description Format* (ADF). The fundamental unit of the ADF is the *Design Unit* (DU), which represents a single graph. The vertices in the graph are called nodes, and each node contains a number of input and output ports. The edges in the graph are known as links and connect the ports. Each node and port in the system as well as the DU itself contains a list of attributes. Support for hierarchy is provided in ADF by placing an attribute on a node which references to another DU. Since DUs are implemented as object-oriented structures, the interface they present to the user keeps the graph in a consistent state.

The *ADFmanager* is responsible for loading and saving DUs through an interface to a relational database. This allows the manager to search the DUs based on the value of any attribute, or any logical combination of attributes. For instance, an agent can pass a request to the ADFmanager to locate any DU that performs a single-precision multiply, is implemented on a Xilinx, and does not use a CORDIC algorithm. The ADFmanager is also responsible for coordinating the actions of the agents on the DU. When an agent requests that a new DU be loaded, the ADFmanager informs each running agent that a new DU is to be loaded, and checks if any agent needs to save the open DU.

CECAAD also provides support to ease the creation of new agents. A class is provided which supplies the functions to interface with the ADFmanager and the DU. Support for creating a user interface to each agent is

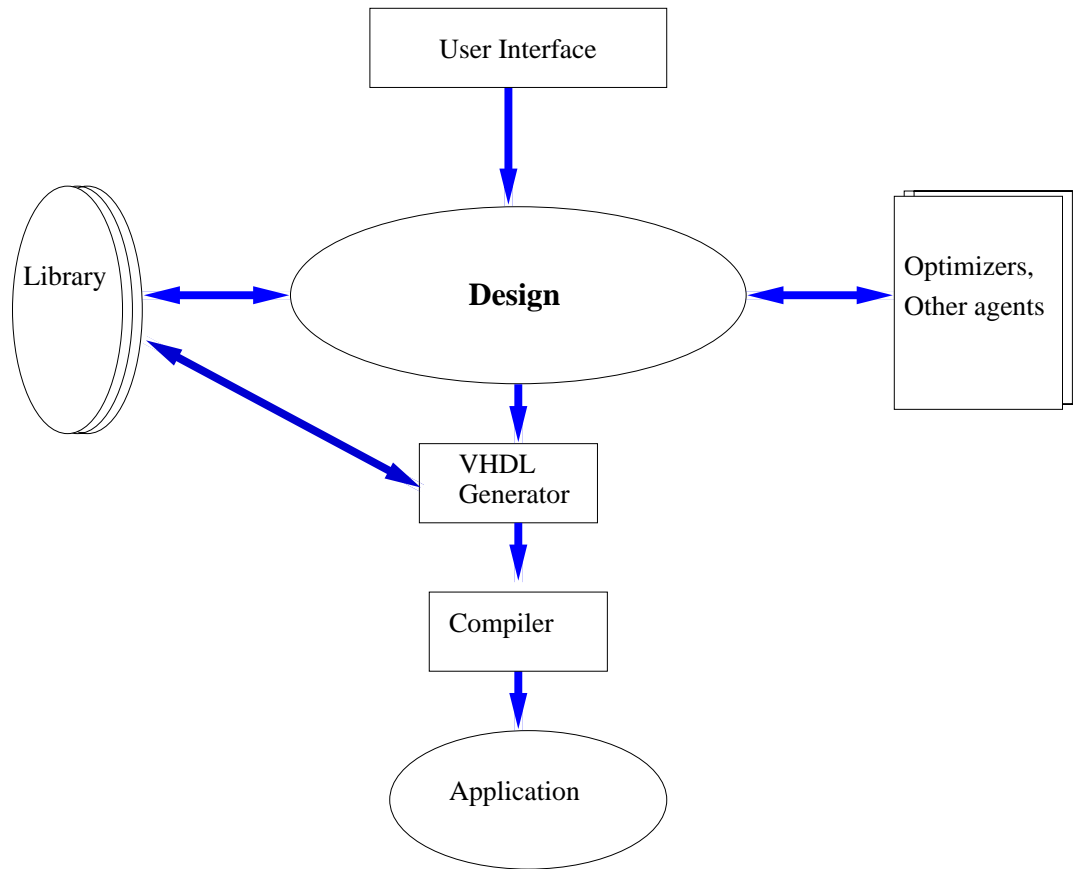


Figure 2. The basic structure of RCADE

supplied through the *view* and *display engine*. A view describes the way a window looks to the user and the way the user interacts with that window. An agent has one view for each window it wants to open. The display engine is responsible for processing the view onto a window. CECAAD allows agents to share windows, and the display engine allows for a seamless transition between one agent's view and the next. Each view and each part of a view is reusable. For instance, the ADF Editor provides a view which displays the DU graphically and allows users to perform basic graph editing functions. The precision analysis agent (described in Section 3 displays the graph in the same way the editor does, but user actions are interpreted differently. The precision analysis agent reuses the display portion of the editor's view, but overrides the functions associated with user interaction

2.2. Component based design

One of the reasons application design for CCMs is more difficult than software design is the lower level of abstraction. Programming languages typically provide support for most arithmetic operations. CCM design typically involves working at the logic block level, and each arithmetic operation must be recreated for each type of FPGA device. This greatly increases both the design and verification burden on the user. Similar to software component reuse, designers have started reusing individual components. In addition, vendors are starting to distribute pre-designed cores. However, there are many implementations of each operation, and the designer is required to select the best implementation of an operation, for their application. Furthermore, it is necessary to design and debug control logic to make the components interact. Changing one component requires changing the control logic and re-verifying the design.

To address these issues, RCADE allows users to design at the level of abstract operations, such as subtraction, multiplication or summation. Initially, these operations are not bound to an implementation. After the analysis agents discussed in Section 3.2 have placed constraints on the graph, a back-end agent selects an appropriate

implementation from a library for each node based on these constraints.

To prevent changing the control logic whenever a component is changed, RCADE components are all designed to interact in a specific way. Each component in the RCADE library contains both the operation and the control logic to interface to any other component. By merging the control logic with the operation, we can achieve designs with no global control logic. This permits an important step forward in new design tools by limiting the need for the designer to understand timing issues, and virtually eliminating timing related design failures. In addition, it reduces the complexity of the back-end agents. It also simplifies the redesign process since there is no longer a state machine to regenerate and reverify.

To achieve complete control independence, we chose to design the component interface using a single register with a bit indicating whether the register is full or empty, a full-empty register. Effectively, this is a one word asynchronous FIFO. The depth is constrained to one or two words to allow for an implementation based on registers rather than RAM which offers savings in both area and routing complexity. Using a single register, this scheme requires two cycles to pass a data item between two components. Alternatively, for a slightly higher area cost, a two register version can be used which allows data to be passed every cycle.

This type of distributed control enables alternative design techniques with minimal impact on overall control complexity. Three particularly interesting cases include: components that require multiple cycles to produce a result, components that require a variable number of cycles to produce a result, and components which execute at different clock speeds relative to the rest of the design. Though it would seem to degrade performance to use components which require multiple cycles per result vs. fully pipelined components, this is not always the case. Multi-cycle components tend to be much smaller in area than their pipelined counterparts. Wherever data cannot be produced fast enough to supply a pipelined component every cycle, an area gain can be achieved by using the smaller multi-cycle part with no performance penalty. This scenario is explained more fully in Section 3 in the discussion of the throughput analysis agent. In certain instances, even smaller variable cycle components are also acceptable as long as the short term average rate is acceptable. Similarly, components that run at different clock rates can be valuable tools to reduce area usage. Again considering the PNN example, the clock rate on the memory system is limited to 33 MHz, thus limiting the rate of the subtractor. The component performing the square, however, can run at 66 MHz, so we can choose a component with a throughput of one data item for every two cycles.

This component-based approach directly addresses many of the goals of RCADE. Ease of use for scientists and a shortened design life cycle are addressed by limiting the need for timing verification of the final design. Code reuse is made easier. This approach helps with portability issues since a hand placed component will always perform well regardless of the choice of device. Unfortunately, components still do not work well across different families of FPGAs, but this is mitigated by the fact that the design is stored in an implementation independent fashion. To migrate to a new family of reconfigurable devices, the back-end tools in essence recompile the same unchanged design for the new platform with a new set of components.

3. AGENTS

All of the functionality of the system is provided through a set of independent agents which interact through the shared design representation. Each agent is intended to independently transform the graph or enhance it with attributes. Front-end agents create an algorithm in ADF. Analysis Agents enhance and constrain the design with attributes describing everything from data formats to area constraints on individual components. Back-end agents generate a working application from the specification by selecting appropriate components and generating the code to connect them.

3.1. Front-end agents

To create an application in RCADE, the user specifies the design with a design entry agent. We describe two ways of entering designs in this section. Users primarily interact with the environment via the *ADF Editor* which can be used to create new designs as well as modify existing ones. We also provide an agent which reads an ASCII file and generates a design.

The *ADF Editor* provides a graphical means of directly manipulating the attributed graph. In addition to the ability to view and perform normal editing functions on the graph, the ADF Editor has an import feature which allows concatenation of existing graphs. The editor operates on the same shared design that other agents see, so

changes made to the graph by other agents are immediately visible in the editor. Through the View mechanism, the “look and feel” of the editor is completely configurable. By default, nodes are represented as circles and the links between them as lines, but this can easily be changed to suit a particular application.

The text translator uses a file format that reflects the underlying design unit format but is structured with keywords and symbols to make it readily understood by humans. For users familiar with block-structured languages (C, Ada, Pascal, *etc.*), the syntax is simple to understand: a design unit is a block that has a list of nodes and links. Nodes are blocks that list their input and output ports. Links connect the ports of arbitrary nodes. The syntax provides a uniform mechanism for adding attributes to design units, nodes, and ports. It also supports an expansion notation that introduces a name for a set of attributes. Thus entities that have attributes in common can simply refer to the name rather than repeat the set of attributes.

Like other agents in our environment, the design entry agents are software components with a well-defined interface to the rest of the system. It is relatively simple for an agent writer to add alternative design entry agents. Generating designs from other applications or high level languages involves developing a parser for the language which builds the ADF graph using methods provided by CECAAD.

3.2. Analysis agents

Analysis agents set constraints for use in optimization, perform graph transformations, and gather statistics from the ADF graph. These agents are intended to be invoked by and interact with the user as necessary to transform the graph and set constraints to produce a viable implementation.

3.2.1. Precision analysis agent

CCMs have not yet matured to the point where the use of standard floating-point formats is practical in general applications.⁴ Users are required to design their applications using a less familiar fixed-point implementation. The *precision analysis agent* provides the ability to propagate precision information through the graph to determine the bit width and format of each value. Frequently, users may have *a priori* knowledge of required or adequate precision at certain points in the algorithm they are implementing. The precision analysis agent gives the user the ability to adjust the precision of a data value on any port in the graph. The effects of each adjustment are then propagated throughout the entire system, and stored as attributes on the ports of each node. The back-end agents of the environment attempt to take advantage of any bit-width savings generated by the precision analysis agent by selecting a part from the library that can be scaled to the correct bit width. Precision analysis used intelligently can greatly increase the number of operations that can be implemented using a given reconfigurable platform. For instance, if the inputs of a particular multiplication are known to only require 8 significant bits, a pipelined 8-bit multiplier can be implemented using less than 10% of the area required if the inputs were assumed to be a standard 32-bit integer.

Upon initialization of the precision analysis agent, precision from the external inputs to the graph is propagated forward to every port within the graph. When the calculated precision is manually altered at some point within the graph, the precision is recalculated from that point forward in the graph. The precision analysis agent also attempts to determine if any savings in bit width can be achieved by propagating the effect of the altered precision backwards from output to input through the graph. Automatic backwards precision calculation is an incompletely specified problem for many arithmetic operations. However, the agent contains a mode which allows the user to interactively enter constraints to aid the backwards propagation.

Operations which the precision analysis agent currently support include addition, multiplication, division, sine, cosine, natural log, square root and buffer. Algorithms for computing the precision for additional operations can be added easily by adding a class when a new operation is added to the system. The precision analysis agent also supports hierarchical designs by computing the precision of subgraphs in the design, then attaching the precision of the external inputs and outputs of that subgraph to attributes of the corresponding node in the parent graph.

3.2.2. Throughput analysis agent

Another way of achieving space savings in FPGA designs is through *throughput analysis*. Throughput analysis is the process of estimating the rate at which each piece of the design will consume and produce data, and eliminate any excess capacity by replacing large and fast parts with smaller slower ones. The *throughput analysis agent* focuses on achieving this goal. This agent traverses the graph and estimates the maximum throughput for each input and

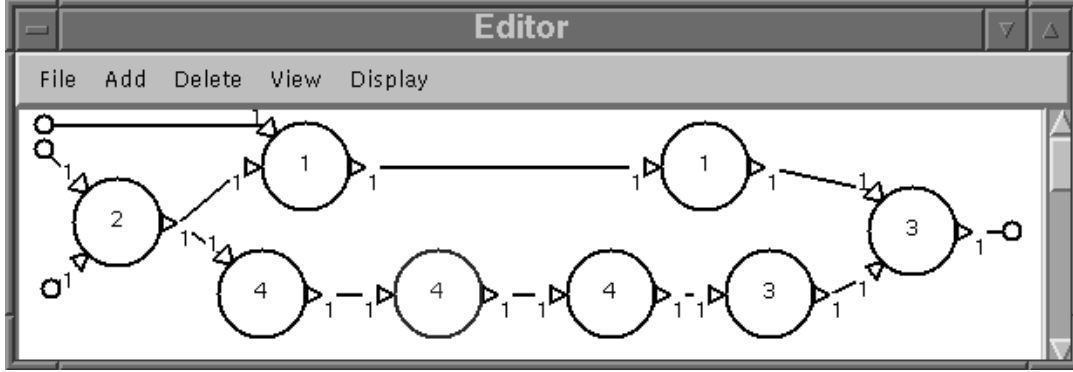


Figure 3. A simple example to illustrate pipeline balancing issues.

output of every node. Realizing that a node cannot, and need not, consume data faster than previous nodes are producing it, the throughput analysis agent then propagates any throughput limitations created by slower nodes throughout the graph. This information is stored as an attribute which is later used by the component selector discussed in Section 3.3.1. For instance, in the PNN example of Section 1, there is a subtraction feeding a square, then a summation, followed by a multiplication. Assuming that the subtraction and square are fully pipelined and the summation sums 16 elements, optimal application performance can still be achieved if the multiplication produces a result only once every 16 cycles. Such a component is a factor of 10 smaller than its fully pipelined counterpart.

3.3. Back-end agents

Back-end agents perform similar functions to the back end components of a compiler. They provide the platform specific details of the implementation. Back-end agents convert an algorithm specification into a working application. At minimum, this requires that a component be bound to each node and that application code be generated.

3.3.1. Component selection agent

The front-end tools express an algorithm in an ADF graph in terms of abstract operations. The analysis agents place constraints on the implementations of these abstract operations. The purpose of the component selector agent is to traverse the graph and select a specific implementation for each node, based on the abstract operation, the constraints, and the target platform. For each node, the component selector queries the database for all functions of that type. It then selects the implementation best meets the constraints.

3.3.2. Pipeline balancing agent

Once all of the components have been selected, the *pipeline balancing agent* traverses the graph and inserts additional buffers (FIFOs) in instances where the simple full-empty register interface would impede performance. These buffers are critical in two scenarios to maintain performance. First, they are needed to maintain a steady data flow into and out of a variable speed part, which may consume or produce data in bursts. The second scenario is illustrated in Fig.3. In this oversimplified illustration, each node is labelled with a latency in the center and a maximum throughput in terms of data items per clock cycle on the ports. We see two branches which start and end at common points. The difference is that one path has a latency four times higher than the other. Though both branches have a throughput of one data item per clock cycle, the effective throughput without pipeline optimization will be an average of one data item per four clock cycles, effectively cutting performance by a factor of four. This occurs because the shorter path cannot continue to consume data, preventing the longer path from consuming data.

To address these problems, we replace the interconnect between selected components with FIFOs. For variable rate components, the interconnects entering and leaving the node are replaced by FIFOs to assist in properly managing data flow for the component. For converging paths, FIFOs are used to provide additional buffering to increase the number of data items stored in one branch to equal the other. For a given branch, m , the number of elements, E_m , stored along the path can be calculated as shown below, where for each component j , the latency is l_j , the throughput is t_j , and L_m is the number of links in the branch, and C_m is the number of components in the branch.

For a set of k parallel branches, the number of additional buffers, N_i , needed in branch i can then be calculated as shown in Eq. 3.

$$E_m = \sum_{j=1} C_m \frac{l_j}{t_j} + L_m \quad (2)$$

$$N_i = \max_k \{E_k - E_i\} \quad (3)$$

The placement of the FIFOs is determined by port attributes and interpreted by the code generator. In the multipath case, all FIFOs are placed at the last link in the path to minimize their collective space requirements. logic and should therefore be kept together. If the number of extra buffers needed exceeds sixteen (a sixteen stage FIFO can be implemented in a single block of a Xilinx 4000 series FPGA), additional buffer stages are added. The result is that the total number of data buffers along any possible path is identical. Therefore, the pipeline balancing agent keeps latency from having an undue effect on throughput.

3.3.3. Code generation

Code generation is the final stage of the design process. A code generating agent is provided to convert the specification of a design in ADF to a VHDL implementation. This task requires instantiating the components, instantiating the interconnection buffers, and interconnecting components through the buffers. At present, RCADE only produces code for single chip designs. Partitioning issues will be addressed in the future.

The first target architecture is an Annapolis MicroSystems WildForce XL series board. Since we only provide for a single chip design, our initial model of the architecture corresponds to PE1 with a single input path, a single output path, and a single port to SRAM. We generate additional code to multiplex outputs onto a single path and demultiplex inputs from a single path. The code generated is suitable for use with the Synopsys FPGA Compiler, and an appropriate compile script is generated. Additional target architectures can be added by providing a new code generator and a new set of components. The rest of the environment remains unchanged.

4. RELATED WORK

Although many research projects are using FPGAs to do computation, their specific objectives vary and often their goals are not the same as ours. In this section, we present a brief overview of some related projects and describe how they address the problem described in Section 1.1.

In general, most research projects have opted for a source language similar to C.⁵⁻⁷ The Transmorgrifier C⁵ developed at the University of Toronto compiles a subset of C and is limited to pragma directives to control the bit widths of integers. The VT project⁶ and a project at NEC⁷ offer more control over what may be constructed in hardware but lack the interaction with the user necessary to develop high performance designs. While the syntax of C is likely to be familiar to new users, the language lacks a direct means for specifying timing or ranges of precisions or multiple storage formats. By itself, C is not a complete solution to our problem but it may be a helpful start. Indeed, nothing in our approach prohibits adding a front-end agent to translate from C to ADF.

The languages Lola⁸ and JHDL⁹ do recognize the need for a central data format and interactions with the user to produce high performance designs. In the case of Lola, the target user is an expert and the language focuses on giving the designer as much control over the implementation as possible. The JHDL project has taken a top-down approach by first working from a host software development perspective, whereas we emphasize building the hardware specifically.

Our use of components is not unique. Others have also used component-based design approaches^{7,10-12} and FPGA vendors, such as Xilinx, are acknowledging the importance of components. Xilinx ships tools with their new software releases to build components. There has been a significant movement toward using third-party components (or cores) as well as module generation tools.^{11,13,14} However, these approaches focus only on minimal area and optimal performance. While this is appealing to some users, it is not ideal for building large designs with numerous components where some flexibility in the component results large gains. Furthermore, these components are usually designed to be centrally controlled whereas we use distributed control for the advantages previously described.

5. CONCLUSIONS AND FUTURE WORK

RCADE provides a configurable computing application development environment which is accessible to a broader range of users than traditional FPGA design techniques, shortens the design life cycle, eases migration of application to new platforms, and yields high performance design. RCADE achieves this by employing a platform independent algorithm description process. The use of independent agents and a common design representation give the environment the flexibility to change with the technology and the needs of the user. The current set of agents allow the user to specify the design, enhance the design with precision information, constrain the design to improve performance and limit area usage, map the specification to a set components, and produce binaries for a current architecture.

In the future, we plan to add agents to provide a functional simulator and debugging support. Library enhancements will consist of more operations, more implementations for each operation, and the ability to integrate cores available from vendors. Algorithm validation can be further simplified by incorporating a runtime debugger capable of executing portions of the design in hardware and single stepping the clock.

On the front end, additional agents will give the user more flexibility in the specification of the design. Initially, we intend to add manual partitioning support to utilize more of the FPGA resources on a CCM and later augmenting this with some automated partitioning support. Additionally, we intend to support the use of higher level language constructs for various elements of design entry. This would involve automatically translating a limited subset of C into ADF, to specify such things as memory access patterns and algorithms for a node. The final front end enhancement being considered is a mechanism for allowing the user to specify the host software and the interaction between the host software and the CCM.

Other analysis agents currently being considered include agents to support functional unit re-use, and hardware/software co-design. An analysis agent supporting functional unit re-use would analyze the graph and determine if cases exist which would allow a more efficient implementation by using a single, faster component rather than multiple smaller, slower components. A hardware/software co-design agent would evaluate the trade-offs of moving components between hardware and software implementations, including issues such as the communication overhead and the performance impacts.

ACKNOWLEDGMENTS

The Planet Earth Research Lab is supported by NASA GSFC under grant number NAG 5-3053. Donations were received from Xilinx under donation number 0000CU7-1978 and Synopsys under agreement number UPS-70004334. Keith Underwood is supported under an NSF Graduate Research Fellowship.

REFERENCES

1. S. R. Chettri, Y. Murakami, I. Nagano, and J. Garegnani, "Comparing the computational complexity of the PNN, the PDM and the mixture model neural network (MMNN)," in *26th AIPR Workshop: Exploiting New Image Sources and Sensors*, 1997.
2. S. R. Chettri and R. F. Crompt, "Probabilistic neural network architecture for high speed classification of remotely sensed imagery," *Telematics and Informatics* **10**(3), pp. 187–198, 1993.
3. "<http://ece.clemson.edu/parl/pse/>," Aug. 1998. This is the CECAAD web site.
4. W. B. L. III, S. P. McMillan, G. Monn, F. Stivers, K. Schoonover, and K. D. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., (Napa, CA), Apr. 1998.
5. D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek, eds., pp. 136–144, (Napa, CA), Apr. 1995.
6. J. B. Peterson, R. B. O'Connor, and P. M. Athanas, "Scheduling and partitioning ANSI-C programs onto multi-FPGA CCM architectures," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., pp. 178–187, (Napa, CA), Apr. 1996.
7. T. Yamauchi, S. Nakaya, and N. Kajihara, "SOP: A reconfigurable massively parallel system and its control-data-flow based compiling method," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., pp. 148–156, (Napa, CA), Apr. 1996.

8. S. W. Gehring and S. H.-M. Ludwig, "Fast integrated tools for circuit design with FPGAs," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 133–138, (Monterey, CA), Feb. 1998.
9. P. Bellows and B. Hutchings, "JHDL - an HDL for reconfigurable systems," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., (Napa, CA), Apr. 1998.
10. D. A. Clark and B. L. Hutchings, "Supporting FPGA microprocessors through retargetable software tools," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., pp. 195–203, (Napa, CA), Apr. 1996.
11. M. Gokhale and E. Gomersall, "High level compilation for fine grained FPGAs," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., pp. 165–173, (Napa, CA), Apr. 1997.
12. M. Gokhale and J. M. Stone, "NAPA C: Compiling for a hybrid RISC/FPGA architecture," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., (Napa, CA), Apr. 1998.
13. M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek, "Object oriented circuit-generators in Java," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., (Napa, CA), Apr. 1998.
14. O. Mencer, M. Morf, and M. J. Flynn, "PAM-Blox: High performance FPGA design for adaptive computing," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., (Napa, CA), Apr. 1998.