Creating Applications in RCADE

Kim Hazelwood Walter B. Ligon III Greg Monn Natasha Pothen Ron Sass
Dan Stanzione Keith D. Underwood*

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

{khazelw, walt, gmonn, npothen, rsass, dstanzi, keithu}@parl.eng.clemson.edu
http://ece.clemson.edu/parl

Abstract—As Field Programmable Gate Array (FPGA) density increases, so does the potential for reconfigurable computing machines. Unfortunately, applications which take advantage of the higher densities require significant effort and involve prohibitively long design cycles when conventional methods are used. To combat this problem, we propose a design environment to manage this additional complexity. Our environment gives the end-user a mechanism for describing and managing algorithms at a higher level of abstraction than other extant methods such as writing VHDL or using schematic capture. This paper describes an experimental version of the environment.

The core of our design tool is a general Algorithm Description Format (ADF) which represents an algorithm as an attributed graph, and a library of components, which are tailored to a particular FPGA device. In this paper we present a set of tools which operate on the ADF representation to provide a number of functions to aid in the creation of applications for configurable computing platforms. These tools include front-end tools for specifying a design, a suite of analysis tools to aid the user in refining and constraining the design, and a set of back-end tools which select components from the library based on the design constraints and place and route these components on the configurable computing platform. In this paper, particular emphasis is placed on the component model and the analysis tools.

*Keith Underwood is supported by an NSF Graduate Research Fellowship

†The Parallel Architecture 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

TABLE OF CONTENTS

1. INTRODUCTION

Configurable Computing Machines (CCMs) based on FPGA technology show tremendous potential for improving the performance of aerospace applications. Unfortunately, the tools to develop applications for CCMs are not maturing as quickly as the technology. This problem manifests itself in two areas: initial application development is complex and time consuming, and migration from one generation to the next requires significant redesign effort. These problems leave many unwilling to migrate their applications from a software implementation to hardware. We have proposed an environment called the Reconfigurable Computing Application Development Environment (RCADE) for the development of CCM applications which will offer higher performance than software solutions, will allow non-experts in FPGA design to develop applications, will simplify the design and maintenance process for those who are experts, and will ease the process of migration from one CCM platform to the next. This paper focuses on the component model used by the back-end of RCADE to generate designs. In particular, we show how our component-based approach leads to a simpler control model, faster and denser designs, shortened lifecycle, and integrates algorithm design and implementation better than conventional approaches.

One of the central themes of RCADE is that the user selects operations from a library without initially specifying an implementation. Once the design is completely specified, these abstract operations are then bound to components. Components in our environment are efficiently implemented blocks of logic (such as adders, multipliers, etc.) that conform to a well-defined interface. This component-based approach provides a number of advantages. The component interface is designed in such a way that components can interface with each other without the need for any controlling state machine. This lack of central control reduces the complexity of timing issues and eases debugging. The interlock mechanism used by the components to achieve this is described in detail in sections 3.1. Separating the selection of the operation from the selection of the component also serves to ease the problem of migration from one FPGA device or CCM platform to the next. The same design, specified by the user as a set of abstract operations, can be re-used by simply running the component selection phase again with a library of components targeted at the new device. Once the component library is ported to a new platform, any RCADE design can be ported quickly and easily.

A final advantage of separating the selection of operations from the selection of components is tighter integration of the algorithm design and implementation. RCADE provides a number of tools that help guide the user in refining their design towards a more efficient implementation. Some of these tools run before component selection to help guide the choice of better components, and some run afterwards to suggest changes to the design. Examples of this kind of tool include the Precision Analysis tool, which attempts to find places in the design where it may be made more compact by using components that use a different data format or reduced precision, or the Throughput Analysis tool that analyzes where space savings can be achieved without performance penalty by choosing smaller, slower implementations. These and other tools are described in detail in section 4..

The RCADE approach to design also shortens the path between algorithm design and design of the implementation. Using conventional hardware design techniques, the design of the algorithm and the implementation are completely separate tasks, and the details of the implementation can take months. In RCADE, the user can interact with both the algorithm design and the implementation, evolving each part and using one to guide the other. The situation is somewhat analogous to programming assembly language on an old batch processing computer vs. using a high-level language compiler on an interactive system. In the former case, once the algorithm is designed the user spent a long period of time constructing a program using instructions at a much lower level of abstraction than the algorithm specification. Further, the penalty for testing the implementation is high, so efforts would be made to complete the whole design correctly the first time. In the latter case, the programmer works at a significantly higher level of abstraction. Operations in the program correspond more closely to the operations specified in the algorithm. In addition, it is relatively simple to compile a program and observe its behavior, then go back and make changes to the source. RCADE attempts to bring the interactive model with a higher level of abstraction to the hardware design world.

In section 2, a brief overview will be provided of the RCADE system, including a description of the infrastructure it is built upon and some of the front-end tools for design entry. Section 3 examines in detail the component model, particularly the interconnection mechanism between components, and the methods of implementing components. Section 4 presents some of the analysis tools used in RCADE, and shows how they can be used to improve some sample designs. Section 5 presents some results of using RCADE to do designs, comparing speed and circuit density of designs using RCADE components to those same designs specified in VHDL. Section 6 compares RCADE to other tools which seek to simplify the design of reconfigurable applications. Section 7 closes with conclusions and future work.

2. System Overview

This section describes the system RCADE was constructed from, and provides a short description of the tools available in the front-end of RCADE.

2.1 CECAAD

RCADE is built upon the Clemson Environment for Computer Aided Application Design (CECAAD) infrastructure for creating Problem Solving Environments [1]. CECAAD provides a common, shared format for representing designs, known as 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 another DU. A number of different tools, known as *agents* have concurrent access to the design. Agents are distinct entities that utilize a well-defined interface to the shared design. CECAAD provides support to ease the creation of new agents, including a model for collaboration between agents and a mechanism for reusing interfaces of existing agents. CECAAD also provides a number of generic agents useful in many environments. The tools provided by CECAAD which are used in RCADE are described below.

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

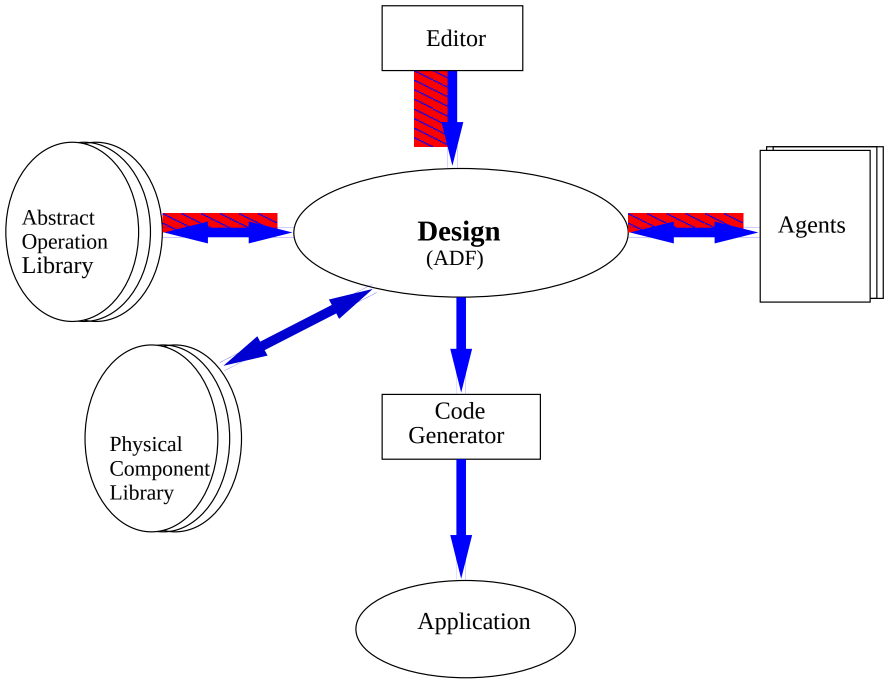


Figure 1: The basic structure of RCADE

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. The *graph editor* provides a direct graphical means for manipulating an ADF graph. The *text translator* translates ADF Designs into and out of a text-based language for describing attributed graph.

2.2 Front-End Tools

The basic structure of RCADE is shown in Figure 1. The basic design flow is as follows: the user specifies the design initially as a graph composed of abstract operations selected from the abstract operation library. The design can be specified either graphically through the graph editor, or textually with a text translation agent. Once the design is specified, the analysis tools discussed in section 4. aid the user in refining and constraining the design. Once the constraints are in place, the component selector binds the abstract operations to implementations from the component library for the appropriate target CCM platform. Additional RCADE tools then aid the user in placing the selected components and partitioning them between devices. Finally the code generator synthesizes an application from the components.

3. Components

Each node which is specified as an operation must eventually be bound to an implementation. Additionally, it is necessary to establish a mechanism for passing data between these components. This section discusses our component interconnection model and the component implementations that we use.

3.1 Component Interconnection

The traditional component interconnection model uses basic components which rely on external control logic “knowing” when data will be produced at each stage. This model introduces a number of complications. For example, each time the properties of a single component are changed, it is necessary to redesign the control logic. Furthermore, the use of iterative components is complicated by the need to provide *start* signals. Finally, it is difficult to stall this type of pipeline to accomodate bursty, buffered IO, especially stalls at the output.

Our design process takes a slightly different approach. We embed control within each component and require that each component correspond to a specific interconnection model. We chose to use FIFOs at each link; thus, components are connected through FIFOs, making the control of a component independent of the components to which it is connected. Each component only produces an element when the destination FIFO is not full and only consumes an element when the source is not empty. In the most economical implementation, these FIFOs have a depth of one, requiring only a single register and a single full/empty bit to implement.

There are many advantages to this type of approach. For example, with a standardized interconnection model and distributed control, we can interchange different implementations of an operation without being concerned about the effects on control. Inherent to this model is the ability to use multi-cycle or variable cycle parts without needing to create special external state machines. This also allows us to always maintain a full pipeline since the components will naturally stall when there is no space available for an output. Another important advantage is the ability to vary the depth of the interconnecting FIFOs to balance pipelines as discussed in Sect. 4.3.

3.2 Component Implementations

Components are initially implemented in RTL behavioral VHDL code. This allows for logical debugging and optimization, and is obviously the most portable and most easily parameterized implementation of the components. Unfortunately, using these components and passing them through the traditional synthesis and place and route path requires a significant amount of time and yields unpredictable results; therefore, RCADE components typically contain relative placements. This produces a number of positive side effects. Obviously, the place and route of a design is a much faster process if pre-placed components are used, making an interactive design cycle more feasible. The use of properly placed components typically results in denser designs than allowing automatic placement of the same logic. Designs using these components are also typically faster than similar designs that don't use preplacement, since the components retain structuring information that is often lost in the traditional path.

4. ANALYSIS TOOLS

Analysis tools are provided to assist a user in improving a design specification. These tools range from the highly interactive Precision Analysis Tool that operates on designs very early in the specification cycle to the almost transparent Pipeline Balancing Tool which operates on a design after the component selection phase. These tools are intended to be very focused and capture some element of a hardware designers expertise so that it may be used by others.

4.1 Precision Analysis Tool

The FPGA devices used in reconfigurable computing platforms are maturing, but the use of standard floating-point formats is still very costly in general applications [2]. Instead, it is often necessary for users to design their applications using a less familiar fixed-point implementation. To compensate for this, we have developed the precision analysis tool to help users specify the appropriate precision for their algorithm. This information can also be used to reduce the number of resources required for some operations.

The precision analysis tool allows a user to specify any *a priori* knowledge about the required precision or adequate precision in the algorithm they are implementing. This information can be entered as either a format or a data range. Formats are specified as the starting and ending bit positions of a number. For example, a number may use the bits from four places to the left of the decimal to four places to the right of the decimal. Precision can also be specified in terms of data ranges. Data ranges consist of a minimum value, a maximum value, and a granularity.

Whether precision is specified as a format or a range, the precision information can be propagated through the graph. The precision analysis tool initializes by checking for precision information at the inputs and outputs. If no precision information has been provided, it prompts the user for enough information to allow continued processing. The precision is then propagated forward to all ports in the graph. The precision analysis tool then attempts to propagate any constraints placed on the outputs back to the inputs. Unfortunately, automatic backwards precision calculation is an incompletely specified problem for many arithmetic operations. For these scenarios, the tool contains an interactive mode for the user to assist in the process of backwards propagation. The tool also allows the user to manually alter the precision at any point in the graph. The effects of this alteration are then propagated through the remainder of the graph.

Operations which the precision analysis tool currently support include addition, multiplication, division, sine, cosine, natural log, and square root. 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 tool 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.

4.2 Throughput Analysis Tool

On most reconfigurable computing platforms, there is a limited amount of communications bandwidth between the host system memory and the reconfigurable computer. Even assuming unlimited bandwidth, there are components which consume more items than they produce or which have a fixed rate of consumption and production. The goal of the throughput analysis tool is to take advantage of these scenarios to select slower, smaller implementations of each operation. Consider the simple examples in Figure 2. In Fig. 2 (a), we see an application which uses four 32 bit inputs that must pass over a single input. Assuming that the inputs and outputs pass over a

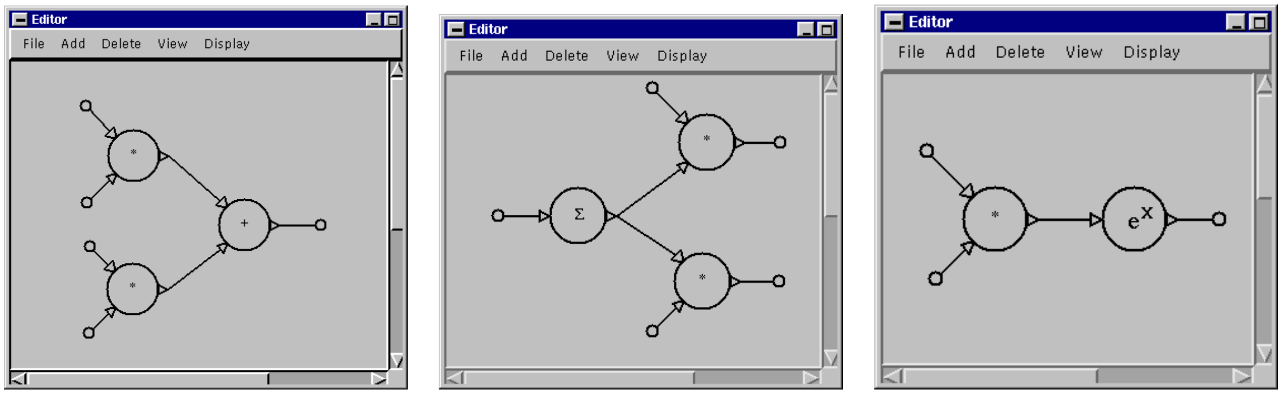


Figure 2: Three illustrations of the usefulness of throughput analysis

single PCI bus to the host and the chip data path can run at 33 MHz, each input can receive at most one data value every four cycles; thus, there is no need for the square parts to be able to consume data more frequently than once every four cycles. Reducing these components from fully pipelined components to components pipelined to take data once every four cycles can gain nearly a factor of four in size reduction. Similarly, in Fig. 2 (b), we see a summation component which sums four elements before producing an output. Again, by using slower multiplier parts, we can achieve a factor of four in space savings. As a final example, consider Fig. 2 (c). Here, we see an iterative exponential component which requires 16 cycles to produce a result and consume the next. In this case, it is unnecessary for the multiplier feeding the exponential to produce a result any more frequently than once every 16 cycles.

The throughput analysis tool begins by considering the inputs and outputs. If no additional information is given, it is assumed that all inputs share a path and all outputs share a path. Currently, we make the further assumption that each input consumes data at the same rate[1]. Similarly, we assume that all outputs are produced at the same rate. Thus, for N inputs, we assume that each input receives an item once every N cycles. The rate of inputs and the rate of outputs are propagated to the attached ports as limits on throughput. All affected nodes are added to a list. Each node has a set of attributes indicating the relative rate of consumption of the inputs, the relative rate of production of the outputs, and the rate of production of each output relative to the rate of consumption of each input. Using these attributes, each node on the list is checked for *consistency*. Consistency is defined for a node as the actual throughputs for the ports satisfying the relative rates specified. Any change to a throughput on a port is propagated to any attached ports, and the affected nodes are added to the list. This is best demonstrated through an example.

Consider the theoretical node[2] in Fig. 3. It has two inputs and two outputs. Two elements are consumed from input A for each element consumed on input B. One element is produced on output C for every element consumend on input B. One element is produced on output D for every two elements consumed on input B. In Fig. 3 (a) we see the status of the graph after a change to the node that is producing data for input B. The numbers indicate the number of cycles per data item. This node is not consistent because if elements of B are only available once every four cycles, elements can only be consumed from A once every two cycles, elements can only be produced on C once every four cycles, and once every eight cycles for D. Fig. 3 (b) shows the node after it has been updated. These updates are then propagated to the attached ports.

[1]Further advances in the throughput analysis tool will allow us to consider the relative consumption rates of the inputs

[2]No computational node is known to have these properties

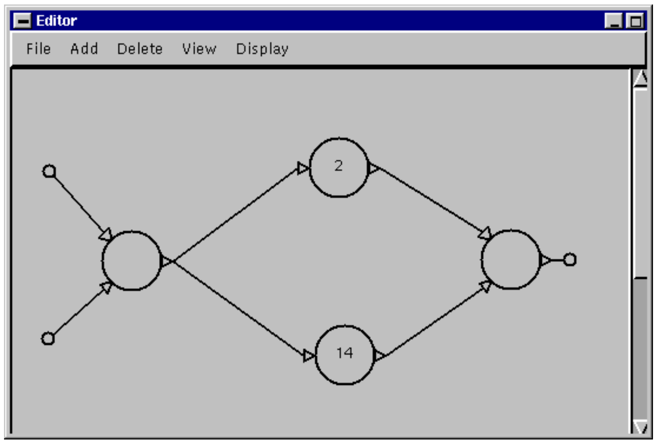
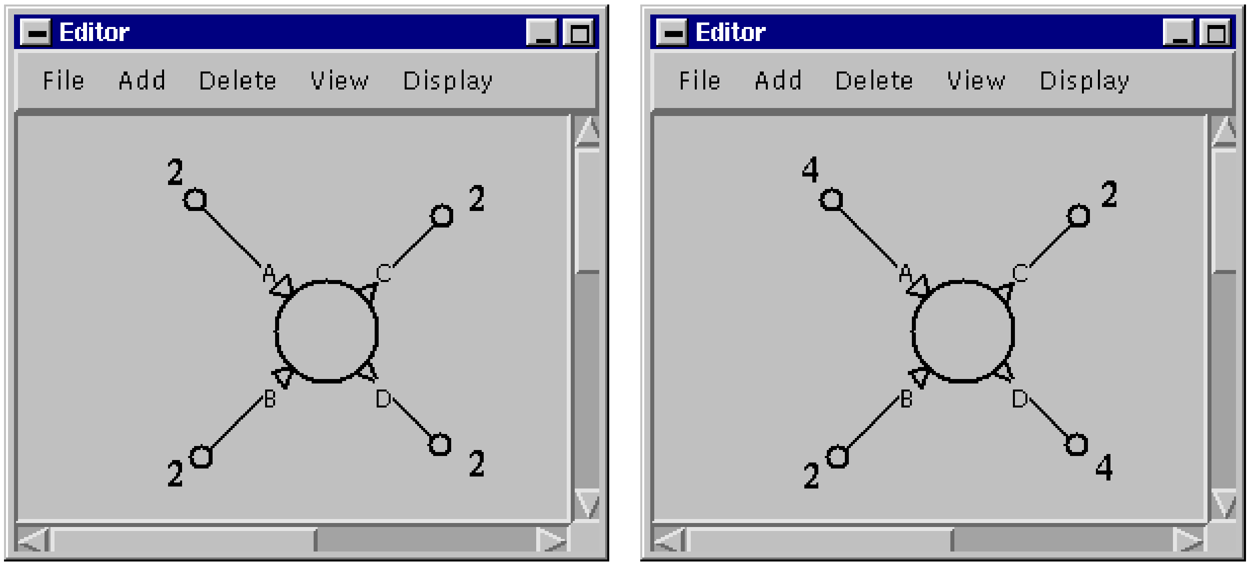


Figure 4: A pipeline balancing example

4.3 Pipeline Balancing Tool

Because many of the component implementations available use pipelining to achieve higher performance, there arises the possibility for imbalance in the pipelines to hinder performance. This problem is illustrated in Figure 4. In this figure, each node is labeled with the number of pipeline stages in that



(a) (b)

Figure 3: (a) An inconsistent node, (b) a consistent node

node. Due to our interconnection mechanism, each link also adds one stage. Note that we are assuming a throughput of one item per clock cycle for each of the nodes. Assuming a path can hold only as many elements as it has pipeline stages, we can see that only four elements are consumed by the upper path before it is "full". Once full, it can no longer consume elements until node C consumes nodes from it; therefore, the lower path can no longer consume elements. At this point, the lower path has only consumed four elements and must propagate these elements through 12 additional stages before producing output. The result is that node C can only process four items every 16 cycles, though it is capable of processing an item every cycle.

To address this problem, we developed the pipeline balancing tool. The pipeline balancing tool is nearly transparent to the user, requiring only that the user choose to invoke it. In short, it analyzes the graph and inserts additional buffering where necessary to maintain performance. The buffers are inserted by replacing the standard interconnects with FIFOs.

The first step in the algorithm is to find all possible paths from each input to each node. The nodes are then sorted by the maximum distance in terms of hops from an input. Starting with the node with the minimum distance, each node is considered and all of the input branches balanced. For a given branch, m, the number of elements, E_m, stored along the path can be calculated as shown in Eq. 1, 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. 2.

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

$$N_i = \max_k \{E_k - E_i\} \quad (2)$$

The placement of the FIFOs is determined by port attributes and interpreted by the code generator. All FIFOs are placed at the last node. If the number of extra buffers needed exceeds sixteen (a sixteen element FIFO can be implemented in a single block of a Xilinx 4000 series FPGA), additional pass-through nodes are added to the graph. The result is that the total number of data buffers along any possible path is identical. Therefore, the pipeline balancing tool keeps latency from having an undue effect on throughput.

4.4 Placement Tool

Proper placement of components within a chip can be very important. As our results in section 5.indicate, poorly constraining the location of components in a design can result in poorer results than not constraining it at all. Unfortunately, the problem of properly placing components within a chip is a difficult one. It relies on the experience and knowledge of the designer — experience and knowledge the average user does not have. To address this issue, we are developing a placement tool which will be responsible for determining the placement of the components.

The first incarnation of the placement tool will present the components, with appropriate size and shape, on a grid representing the device being used. Connections between components will be illustrated as lines. Further extensions will

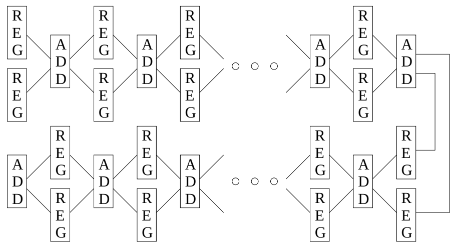


Figure 5: Design layout #1 for the add_tree application

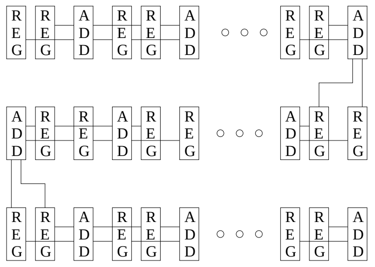


Figure 6: Design layout #2 for the add_tree application

add a suite of algorithms to the tool which attempt to find an optimal layout for the specified algorithm. The simplest of the algorithms will be based on minimizing the maximum distance between connected components while attempting to properly align data paths. Further revisions will leverage the large body of work available in the field concerning placement schemes and module compaction.

5. Applications

In this section we present three applications to show the advantages of the component model vs. traditional hardware design techniques, and to show the importance of proper placement. The applications presented are Adder_Trees, Modis and Snow, Ice and Vegetation Indices. These applications are compared using three different methods. The first method uses RTL behavioral synthesized VHDL code. The second method uses hard macro components and allows the Automatic Place and Route (APR) tools to place the components. The third method uses hard macro components and does not allow APR to place the logic, instead the designer creates placement constraint for the application. We also present results from tests run for two evaluation metrics, design performance and total design time to implementation, using the three methods described.

All application tests were run on an unloaded Sparc Ultra1 with 192 MB of RAM. Synthesis and place and route were run for the target speeds of 33 and 50 MHz. If the design could not make the target speed then N/A was inserted into the results table. The results tables show the target speeds for each test, the actual speed achieved and the design times for synthesis and place and route for each design. Some designs were tested with two different manual placements. The headings Placed #1 and Placed #2 indicate the two different layouts for the application, not using APR tools. The Unplaced heading is for the design using the placement of the APR tools. The Synthesized heading shows results for the RTL behavioral code.

5.1 Add_Trees

In the first design, shown in Figure 5, the registers preceeding each adder are placed in a single column. Each adder then feeds two output registers, which in turn serve as the registers preceeding the next adder.

The second design layout, shown in Figure 6, emphasizes the data path. The data paths of the registers are aligned with those of the adders. This placement is replicated until the end of the chip is reached. The design then folds down and the pattern is continued. The folding is done once more due to finish filling the chip.

The results for the add tests are shown in Table 1. The table shows that the RTL behavioral design could only reach a speed of 34.9 MHz in a little over 33 hours. Using the hard macro components with the APR tools the speed increases to 37.5 MHz while reducing the run time to under 4 hours. When using hand placement of the hard macro components the speed increases even further to 39.2 MHz requiring only 1 hour and 25 minutes. When the target speed was increased for the second placed design the actual speed only improved by 1.6 MHz. This is a return of only 4% performance improvement for an investment of an additional 3 hours of run time. It can be observed that the first placement is not an optimal placement since it does not address the data path through the design as the second layout does. This is shown by the fact that the first target speed was not achieved.

5.2 Modis

This application, shown in Figure 7, is a pared down version of an algorithm used for the modis instrument. The application takes in a data element and splits it to the initial eight registers, performs the operations and then produces a 12 bit result. The initial registers feed a column of multipliers. The multipliers output to registers which are the inputs for two stages of adders.

This application was tested using three variations. The first design uses an iterative multiplier, the second design uses a pipelined multiplier and the final design uses the iterative multiplier with four instances of the algorithm on the chip.

The results for the iterative modis application are shown in Table 2. Note that this implementation required a relatively small fraction of the device to implement; thus, there is little

Table 1: Adder_Tree, Design Performance and Total Design Time

	Placed #1		Placed #2		Un-placed		Synthesized	
Target Speed (MHz)	33	50	33	50	33	50	33	50
Actual Speed (MHz)	32.5	N/A	39.2	40.8	37.5	36.7	34.9	28.5
Synthesis Time	0:02:38	N/A	0:02:38	0:02:38	0:02:10	0:02:35	20:03:52	10:06:51
Place and Route Time	5:25:50	N/A	1:22:28	3:57:37	3:32:45	3:45:28	13:32:24	74:06:34
Total Design Time	5:28:28	N/A	1:25:06	4:00:15	3:34:55	3:48:03	33:36:18	93:46:55

Table 2: Modis with Iterative Multiplier, Design Performance and Total Design Time

	Placed #1		Placed #2		Un-Placed		Synthesized	
Target Speed (MHz)	33	50	33	50	33	50	33	50
Actual Speed (MHz)	41.6	42.5	40.9	42.0	40.9	42.4	37.6	40.6
Synthesis Time	0:02:04	0:02:05	0:02:05	0:02:05	0:02:05	0:02:05	0:46:44	0:44:40
Place and Route Time	0:11:31	0:31:59	0:12:38	0:44:20	0:16:13	0:32:20	0:59:42	2:02:48
Total Design Time	0:13:35	0:34:04	0:14:43	0:46:25	0:18:18	0:34:25	1:46:26	2:47:28

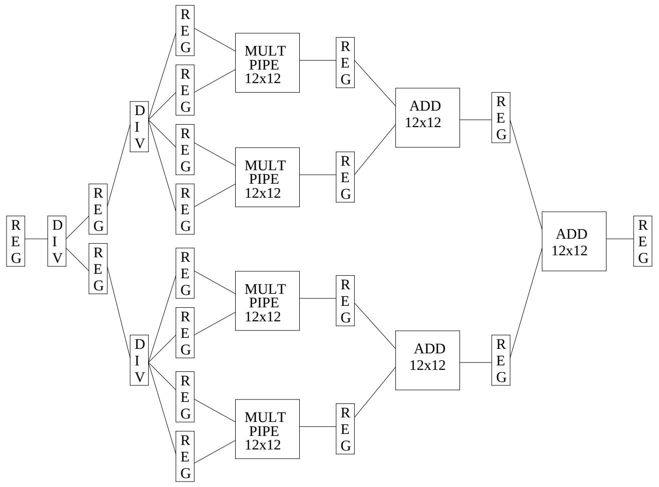


Figure 7: The Modis Application

variation in the performance and implementation times for the three versions using macros. The synthesized version, however, takes significantly longer to achieve poorer results, even on this low density design.

The results for the pipelined multiplier variation are shown in Table 3. Bigger performance differences can be seen in these designs, because the density of the application almost doubles when inserting the pipelined multipliers. The RTL behavioral design could not reach the first target speed of 33 MHz after requiring 18 hours to complete. With this variation of the application the APR placed design actually runs faster than the hand placed design, 44.7 MHz versus 39.5 MHz. This is a case where the placement of the components is difficult and it is left to the designer's ability to achieve a good placement of the components in the device. Obviously, the automated tools can consider more options much faster than the designer. It is for this reason that we are currently adding a placement agent to assist the user in the placement of components.

The results for the iterative multiplier design with mulitple application instances is shown in Table 4. The Total design time in these tests is a major improvement over the RTL behavioral code. The results show the RTL behavioral attaining an actual speed of 37 MHz in 23 hours. The APR placed version shows significant speed up with an actual speed of 42.4 MHz, taking just under 3 hours. The hand placed improves further in the area of total design time reaching nearly the same speed in under an hour. This table illustrates the importance of using hard macro components as the density of the application increases.

Table 3: Modis with Pipelined Multiplier, Design Performance and Total Design Time

	Placed		Un-Placed		Synthesized	
Target Speed (MHz)	33	50	33	50	33	50
Actual Speed (MHz)	39.3	39.5	39.2	44.7	26.2	N/A
Synthesis Time	0:02:04	0:02:04	0:02:06	0:02:05	7:27:34	N/A
Place and Route Time	0:29:15	1:38:11	0:44:21	1:12:27	11:20:48	N/A
Total Design Time	0:29:19	1:40:15	0:46:27	1:14:32	18:48:22	N/A

Table 4: Modis (4 Instances), Design Performance and Total Design Time

	Placed		Un-Placed		Synthesized	
Target Speed (MHz)	33	50	33	50	33	50
Actual Speed (MHz)	41.8	42.3	40.9	42.4	37.4	36.9
Synthesis Time	0:02:30	0:02:25	0:02:25	0:02:25	15:55:00	15:55:39
Place and Route Time	0:49:34	2:10:13	1:32:47	2:40:46	7:32:41	20:34:20
Total Design Time	0:52:04	2:12:38	1:35:12	2:43:11	23:27:41	36:29:59

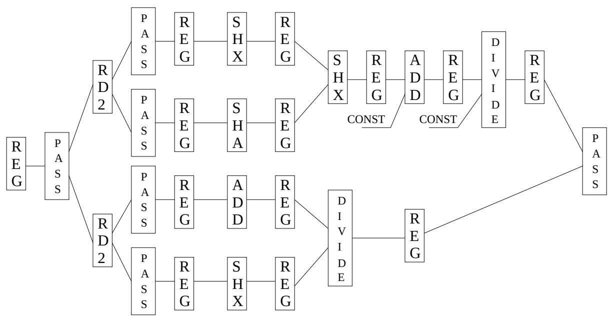


Figure 8: Snow & Ice Index with Vegetation Index

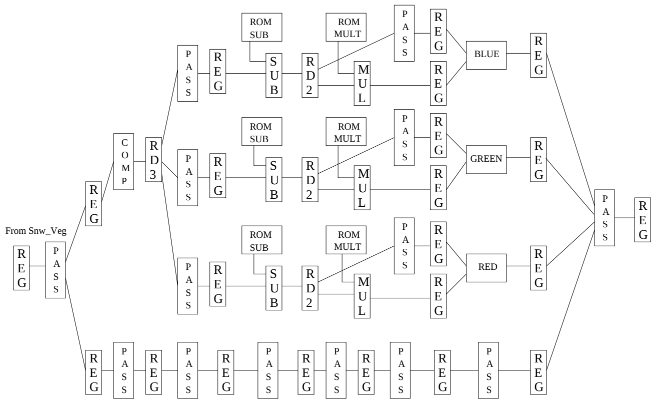


Figure 9: Color Map and Bypass for Snow & Ice Index with Vegetation Index

5.3 *Snow, Ice and Vegetation Indicies*

This application uses bands of satellite telemetry to illustrate the amount of vegetation, snow and ice in a particular region of satellite data. For each index the first two bands of each pixel are used (I, II). The basic algorithm for the vegetation index is (2*II)/(I+II). Each arithmetic component for the algorithm has been translated from vhdl into the hard macro components used. The vegetation index requires a color mapping for the output to be visible in a bitmap form. These components were also translated from the C code for the mapping into hard macro components.

The snow and ice index algorithm does not require a color mapping as it computes each color band in the transformation. The equations for each band are as follows: Red band = 4*II - 3*I, Green band = II, Blue band = I. Figure 8 shows the transformations of the pixels. Figure 9 shows the color mapping for the vegetation index and the pass through segment for the snow and ice index.

The results for the Snow, Ice and Vegetation Indicies are shown in Table 5. The RTL behavioral code can achieve a top speed of 37.5 MHz in a fraction under 6 hours. The APR results show some improvement in one third of the time with 39.7 MHz. The hand placed design does a little better than this with 40.0 MHz in one hour. The results in this test do not fair extremely well for the hand placed design since the density of the entire application is under 40% of the chip.

5.4 *Analysis*

The results indicate several interesting characteristics of the hard macros. First, they improve the performance results for applications across the board over their synthesized coun-

Table 5: Snow, Ice and Vegetation Indicies, Design Performance and Total Design Time

	Placed		Un-Placed		Synthesized	
Target Speed (MHz)	33	50	33	50	33	50
Actual Speed (MHz)	38.7	40.0	39.5	39.7	37.2	37.5
Synthesis Time	0:03:48	0:03:40	0:03:40	0:03:40	2:23:45	2:18:46
Place and Route Time	0:22:03	0:57:45	0:39:06	1:18:44	1:30:59	3:36:43
Total Design Time	0:25:51	1:01:25	0:42:46	1:22:24	3:54:44	5:55:29

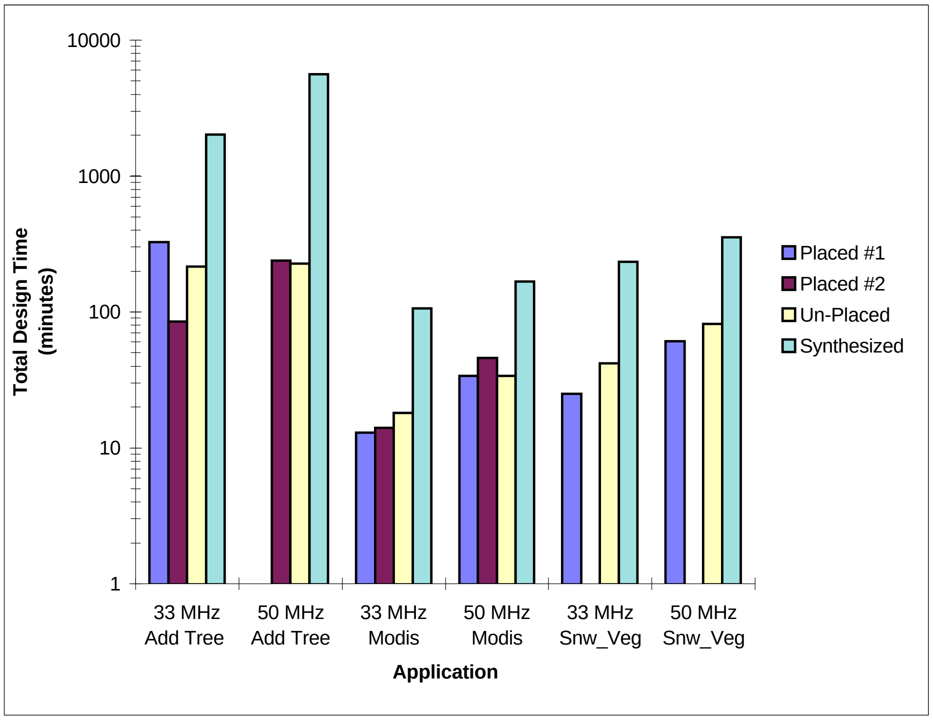


Figure 10: Application design time

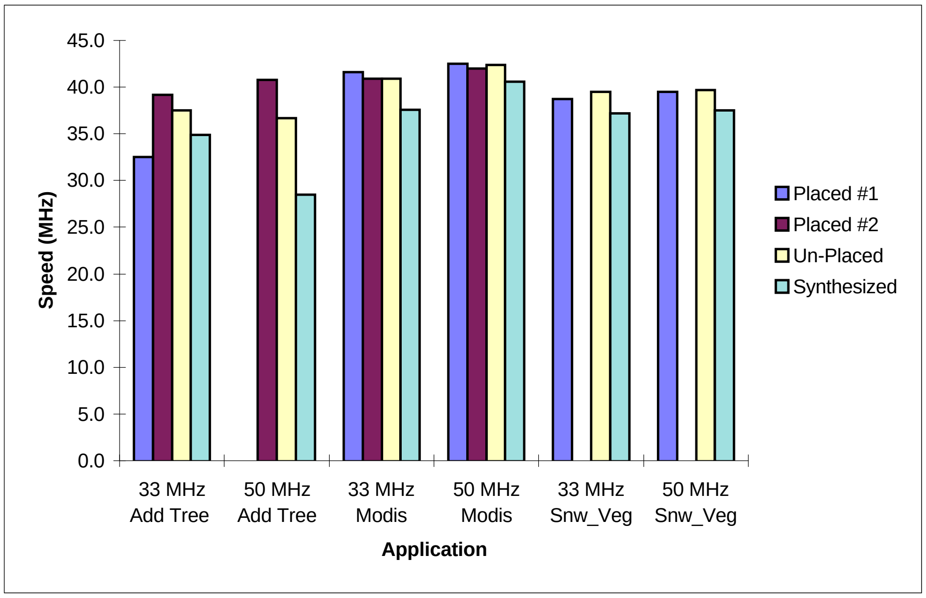


Figure 11: Application performance

terparts. In some cases, these improvements are drastic, though in low density designs it is less noticeable. These results are summarized in figure 11. Second, the use of hard macros drastically reduces the total time to implement a design. Much of this savings comes from reducing the time required for synthesis, though in most case place and route times are also drastically reduced. The impact on design time across all the applications is summarized in figure 10 Finally, it is very important to carefully consider the manual placement of the macros. A poor placement can produce worse results than no placement at all. Unfortunately, when it is left to the designer to choose the placement, there is little to go on other than experience and intuition. Automated placement tools have the advantage of being able to consider all of the routing delays. For this reason, we intend to implement a placing agent which will create automated placements of the components.

6. Related Work

Many research projects have attempted to create fast, efficient, easy to use development environments for reconfigurable computing. Most of these projects have opted for a source language similar to C.[3, 4, 5, 6, 7] While the syntax of C is likely to be familiar to new users, the language lacks a direct means for specifying timing, 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. Our focus, however, is on the additional information that the user can provide.

For all design environments, it is necessary to have a back-end process to map the algorithm to a given technology. Some choose VHDL as an output format, and then allow synthesis and APR to produce results. This is indeed the most portable of the formats. We choose to map to a set of components in order to improve space efficiency, performance, and the time to implementation; however, our use of components is not unique. Work in [8] and [9] reports similar advantages to working with modules, though they typically work on smaller scale designs and do not address the need for distributed control. Koch [10] addresses some of the problems using modules such as internal fragmentation resulting in lowered device utilization. Because our components are completely self-contained and interact with other componets through registers, we see little of this type of fragmentation; however, we are working to address the internal fragmentation inherent in our register implementations.

Others are taking the idea of modules one step further and introducing module generators [11, 12, 6]. These module generators produce parameterized modules based on known descriptions. Module generators are one way we plan to extend our environment and take advantage of the improved implementation times afforded by using modules.

7. Conclusions and Future Work

RCADE combined with the component-based design approach offers a powerful, flexible new tool for the design of reconfigurable computing applications. It helps a user to specify all of the salient properties of a design, and then uses those properties to build a bettter design. Most importantly, it is easily extended with the addition of new agents.

It has also been shown that the use of relationally placed components can greatly reduce run times and improve performance, even if we allow the automated tools to place the macros on the chip. These improvements lead to a design cycle of a reasonable, predictable length. This makes it much easier for others to accept the technology by making iterative design cycles more feasible. Further advances, such as routing the individual components, could reduce this design cycle even further.

Future work will include new agents, and improved components. Specifically, we are working on a placement agent, and a partitioning agent which will aid the user in partitioning a design between multiple devices and placing the components within a device. We are also considering better ways to build components, including the possibility of module generators. Another improvement to components being considered is routing the individual components in addition to the placement that is already being done. This should reduce the place and route times by as much as an order of magnitude.

References

[1] "`http://ece.clemson.edu/parl/pse/`," Aug. 1998. This is the CECAAD web site.

[2] 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.

[3] 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.), (Napa, CA), pp. 136–144, Apr. 1995.

[4] 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.), (Napa, CA), pp. 178–187, Apr. 1996.

[5] 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.), (Napa, CA), pp. 148–156, Apr. 1996.

[6] 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.), (Napa, CA), pp. 165–173, Apr. 1997.

[7] 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.), (Napa, CA), pp. 195–203, Apr. 1996.

[8] J. Shi and D. Bhatia, "Performance driven floorplanning for FPGA based designs," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, (Monterey, CA), pp. 112–117, The Association for Computing Machinery, New York, NY, February 1997.

[9] T. J. Callahan, P. Chong, A. DeHon, and J. Wawryznek, "Fast module mapping and placement for datapaths in FPGAs," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, (Monterey, CA), pp. 123–132, Feb. 1998.

[10] A. Koch, *Regular Datapaths on Field-Programmable Gate Arrays*. PhD thesis, Vom Fachbereich fur Mathematik und Informatik der Technischen Universtat Braunschweig, July 1997.

[11] H. Krupnova, C. Rabedaoro, and G. Saucier, "Synthesis and floorplanning for large hierarchial FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, (Monterey, CA), pp. 105–111, The Association for Computing Machinery, New York, NY, February 1997.

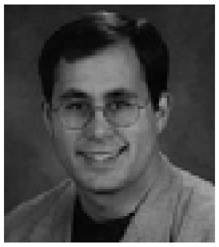
[12] W.-J. Fang, A. C.-H. Wu, and D.-P. Chen, "Module generation of complex macros fo logic-emulation applications," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, (Monterey, CA), pp. 69–75, The Association for Computing Machinery, New York, NY, February 1997.

Kim Hazelwood *received her B.S. in Computer Engineering from Clemson University in 1998. She is currently working towards her masters in Computer Engineering at North Carolina State University. Her interests include computer architecture and hardware design.*

Walter Ligon *received his Ph. D. in Computer Science from the Georgia Institute of Technology in 1992. Since then he has been at Clemson University where he is an Assistant Professor in the Department of Electrical and Computer Engineering. His current research interests are in parallel and distributed systems, I/O for parallel systems, reconfigurable computing,and problem solving environments.*

Greg Monn *received his M.S. in Computer Engineering from Clemson University in 1998. His interests include reconfigurable computing and FPGA design.*

Ron Sass *received his B.S. in Computer Science and Engineering from the University of Toledo. He earned an M.S. in Computer Science from Michigan State University in 1992 and is finishing a Ph.D. in the same field at that institution (December 1997). He was a Visiting Instructor at Michigan State from August 1996 to August 1997. Ron joined Clemson University first as a Visiting Assistant Professor/Research Associate in August 1997 and then as an Assistant Professor in August 1998. Ron's research interests include systems and high-performance computing (reconfigurable computing and instruction-level parallelism), compiling for cluster computers, and modeling multimedia traffic for computer networking. Ron is a member of the ACM.*



Dan Stanzione, Jr. *is a Ph. D. candidate in Computer Engineering at Clemson University. He received his B.S. in Electrical Engineering in 1991 and his M.S. in Computer Engineering in 1993, both from Clemson. Dan's research interests include reconfigurable computing, remote sensing, and problem solving environments for high-performance computing systems.*

Keith D. Underwood *received his B.S. in Computer Engineering from Clemson University in 1995. Currently, he is working on his Ph. D. in Computer Engineering from Clemson University. His interests include reconfigurable computing, parallel and distributed systems, and problem solving environments.*