Spatio-Temporal Partitioning of Computational Structures onto Configurable Computing Machines

Rhett D. Hudson, David Lehn, Jason Hess, James Atwell, David Moye, Ken Shiring and Peter Athanas

Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, Virginia 24061-0111
[rhudson|athanas]@vt.edu

ABSTRACT

A complete computing system supports a design path from problem description to implementation. The term configurable computing refers to complete computing systems that support the development of applications for configurable computing machines (CCMs). Configurable computing systems generally include a microprocessor-based host, a configurable processing array and the tools necessary for capturing the problem and mapping it into software for the host and configurations for the hardware This work proposes a framework for a set of platform independent configurable computing tools. The proposed tools temporally partition large designs, described in a textual language, into stages that can be mapped onto the computing array. The temporal partitions are spatially partitioned to support multiple FPGA arrays. These results are then given to platform specific backends that convert the tool's description of the design into functional FPGA configurations, hardware controllers and host-based control code.

Keywords: Run-time reconfiguration, FPGA, CCM, temporal partitioning, spatial partitioning

1. INTRODUCTION

Run-time reconfigurable (RTR) custom computing machines (CCMs) are systems that take advantage of the programmable nature of the FPGA by utilizing multiple configurations. RTR CCMs can solve problems that require more resources than are available on all the FPGAs in the system [EldH96]. They accomplish this by partitioning the problem into stages that fit in the available FPGA logic. The RTR CCM calculates a portion of the problem and then stores the intermediate results. Then the CCM reconfigures the FPGAs for another part of the computation and continues where the last configuration left off [HudL98][EldH94].

Most work in the FPGA-based computing field has dealt with platforms that provide the ability to rapidly prototype application specific hardware. This form of computation follows the traditional design techniques adopted by the ASIC community. Most commercial FPGA design tools are ASIC design tools that were simply retargeted from VLSI cell libraries to support similar libraries of primitives for FPGAs. These rapid prototyping platforms configure their FPGAs once. That configuration is used throughout the execution of an application. The obvious limitation of this approach is that one can always conceive of a problem that requires more resources than are available on a given computing array.

While ASIC solutions must adopt this kind of static design, applications based on the inherently flexible FPGA do not. RTR applications solve the scalability problem of traditional rapid prototyping techniques by adopting a divide and conquer approach. Large problems are broken down and partitioned temporally into stages, each of which fits onto the array. The first stage receives input data, performs computations and stores the results into a memory. The array is then reconfigured for the next stage, which computes results based on the outputs of previous stages. This process is then repeated until all the required stages have executed and the final results are available. Using these techniques, the size of a problem that an RTR application can solve is limited only by the size of the memory required to store intermediate results and possible latency requirements.

The efficiency of these systems is dependent on the amount of time the platform spends doing useful computations. Time spent reconfiguring the FPGAs and reordering the intermediate results is lost as overhead[1]. Consequently, each

[1] Some FPGAs have the ability to overlap computation with configuration by supporting multiple contexts. These devices can, in theory, be used to significantly reduce configuration overhead [ScaV98].

reconfiguration has a cost in terms of efficiency and performance. Ideally, the time spent reconfiguring should be negligible with respect to the time spent computing. The task of partitioning a given design into temporally ordered stages is a complex and time consuming one. This paper describes a set of computer-aided engineering tools that addresses the needs of an RTR designer.

Another critical stumbling block in the commercial development of configurable computing is the lack of standards for interfacing with CCMs. The development of high-level design tools must be based on some set of convenient assumptions. The same way that the C language can assume that target architectures have a stack (or at least something that appears to be a stack) to work with, high-level design tools for CCMs must be able to make some assumptions about their interface with the underlying hardware. This paper presents an object-oriented interface based on a versatile architectural description language that provides the required assumptions necessary for high-level development tools, while not making undue restrictions on the hardware.

Section 2 of this paper describes the boundaries of the problem being addressed and provides a high-level overview of the process required to compile the design description into functional hardware. Section 3 discusses the architectural description file used to describe target architectures. Section 4 gives a brief overview of the graph description language. Section 5 looks at the resource structure of the standard component library. Section 6 addresses the temporal partitioning process. Section 7 provides some literature background on spatial partitioning and discusses the application of those techniques to the CAE tools being described here. Section 8 mentions the software interface to the underlying CCM that allows the tools to be retargeted to any RTR CCM. Section 9 provides some conclusions and discusses some of the limitations of the current generation of CCMs as target platforms for this technology.

2. APPROACH

The design of the image interpolation engine described in [HudL98] demonstrates the basic design process required for the creation of a RTR CCM application. The process begins with the selection of an application that is suited for acceleration by an RTR CCM. To automate the design process, the computational structure of the application must first be described in a high-level representation that is convenient for the user and descriptive for the automated tools. In the approach described here, this language is GDL. GDL is high-level enough that humans could program in it if necessary, however, the intention is for GDL to serve and an intermediate step between another high-level language and the CAE tools described here.

The first step is to parse GDL and create an internal representation of the computational structure in a convenient data structure. The GDL representation includes information concerning the computational resource requirements of the operations that it invokes. These requirements must be compared to the resources available on the target architecture. The resources available on the target architecture are described in another file, called the architectural description file. If the computational structure of the problem requires resources that exceed the capacity of the target architecture, the application must be temporally partitioned. This process analyzes the computational structure and identifies partitions of the structure whose resource requirements do fit onto the computational array. These stages can then be temporally sequenced onto the platform to perform the overall computation.

If the configurable computing array is composed of multiple FPGAs, then each temporal partition produced in the previous step needs to be spatially partitioned. This process breaks up the computational units in each temporal partition into chunks that will fit on individual FPGAs. This process needs to take into account the size and interconnection of the FPGAs available in the architecture. Again, this information is available from the architectural description file.

The interface between the CAE tools and the actual CCM is a set of C++ classes that represent the creation of hardware description files for processing by the vendor's tools and the host side control of the platform. The public interface to these classes provides an abstracted interface to a virtual CCM. The private implementation of these classes constitutes the platform specific backend. This allows the CAE tools to be retargeted to a new CCM by creating a new implementation of the standard library, a new architectural description file, and a new implementation of the two C++ interface classes.

3. ARCHITECTURAL DESCRIPTION FILE

The purpose of the architectural description file is to describe the resources available for performing computations. These resources include the number and size of the FPGAs available, the availability of interconnect between the FPGAs, and the topology of any switching resources. This architectural representation is an abstraction of the actual architecture. The

hardware specific developers who customize a backend for the tools may choose to portray the architecture in a manner that is very different from the actual architecture. Here is an example of an architectural description file for a typical CCM, the AMS WILDFORCE.

```
resource CLB;
resource MPORT;
resource BW;

fpga pe1 { CLB<=1024, MPORT<=8 }
fpga pe2 { CLB<=1024, MPORT<=8 }
fpga pe3 { CLB<=1024, MPORT<=8 }
fpga pe4 { CLB<=1024, MPORT<=8 }

data SYS1_2 { CLB<=0, MPORT<=0 ,BW<=288 }
data SYS2_3 { CLB<=0, MPORT<=0 ,BW<=288 }
data SYS3_4 { CLB<=0, MPORT<=0 ,BW<=288 }

pe1 <-> SYS1_2;
pe2 <-> SYS1_2;
pe2 <-> SYS2_3;
pe3 <-> SYS2_3;
pe3 <-> SYS3_4;
pe4 <-> SYS3_4;
```

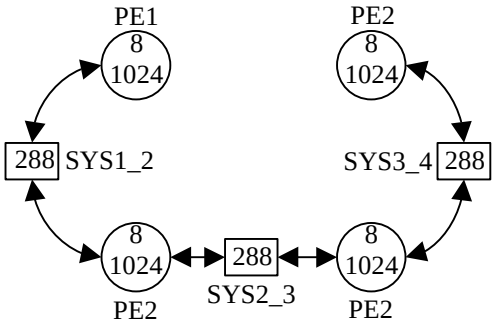

Figure 1: AMS WILDFORCE abstraction.

The first three lines of the file begin with the *resource* keyword. They declare literal identifiers that will be used later in the architectural description file and in attribute specifications in the standard component library. None of the resource literals are reserved words. The choice of name and resource structure is left up to the backend architect. Since the `WILDFORCE` is based on Xilinx FPGAs, the literal CLB was chosen to represent the number of available Xilinx CLBs on a given processing element. The MPORT identifier represents the number of memory ports that can be assigned to a processing element. The BW identifier is used to constrain the maximum bit width of data that can be allocated across an interconnection. The spatial and temporal partitioners handle all resource constraints according to certain rules discussed later. It is the job of the back end architect to structure the resource constraints in the architectural description file so that they accurately represent the limitations of the architecture.

Following the resource identifier declarations come declarations for elements capable of performing processing. These lines begin with the *fpga* keyword. After the keyword is a literal that will be used later on to refer to the specific computational element being described. Following the literal is a list of resource limitations enclosed in braces. These resource limitations are expressions describing the resources available on a given computational element. In the case of *pe1*, the resource limitations indicate that the CLB resources allocated to *pe1* must be less than or equal to 1024.

Next, the sample architectural description file declares data nodes. Data nodes represent interconnections between the computational elements. These lines begin with the data keyword and provide a literal for referring to the data node and a brace-enclosed list of resource constraints. Finally, the sample architectural description file lists a set of expressions that indicate how the *fpga* and *data* nodes are interconnected. The architectural description file shown here describes a structure shown graphically in Figure 1.

The actual `WILDFORCE` architecture is more complex than this representation. The theoretical backend developers in this case have chosen to represent `WILDFORCE` in a manner that is convenient for applications generated by the automated tools. Preliminary results indicate that applications generated by the proposed tools favor architectures with many paths to memory and high-bandwidth interconnect. The `WILDFORCE` architecture can provide one path to memory for each processing element and has 36-bits of static interconnect between the processing elements. To simulate a more powerful architecture, the backend designer in this case has chosen to implement some temporal multiplexing of the memory ports and static interconnect that is hidden from the tools and the applications developer.

There are other features of the `WILDFORCE` that are not represented in this description. There is a fifth processing element that is conveniently wired for providing centralized control of the other processing elements. The interconnect between the

fifth processing element and the ones represented in the sample architectural description file is too narrow for the purposes of the automatic tools, so the theoretical backend designer has chosen to omit it from the description. The backend designer may very well use the fifth processing element for control or some other purpose, but that is an implementation detail that is hidden from the tools. `WILDFORCE` also provides a crossbar interconnecting the processing elements. The crossbar has several features that make it difficult to use in the proposed automated design process. Representing the crossbar interconnect would require a much more advanced architectural description file format and significant changes to the spatial partitioning process. There are also several FIFOs available, but they are not useful to the tool's view of the architecture, so again, they are abstracted away.

4. GRAPH DESCRIPTION LANGUAGE

The CAE tools proposed here begin with a high-level description of the application formulated in GDL. GDL is a language for describing hierarchical graphs. In its current form it is intended to be more of an intermediate language than a human interface. Modifications and extensions to the language could make it a quite capable programming language, but language design is not the focus of this research effort. Translators from more familiar languages are possible [PetA96]. User friendly graphical front-ends are also possible.

GDL supports C++-like declaration and definition of sub-graphs and instantiation of these sub-graphs inside other structures. The CAE tools provide a library of primitive operations that the user can compose to create more powerful operations. Declarations for the primitive operations are #included in a GDL source file. The behavior of the components in the standard library is guaranteed to be uniform across any platform specific backend for the tools. Here are some sample declarations of primitive operators.

```
add(lhs:16,rhs:16)->result:16;
mult(lhs:16,rhs:16)->result:16;
div(lhs:16,rhs:16)->(quot:16,rem:16);
```

The literals *add*, *mult* and *div* are the names of the operations. The C-language analog of an operation is a function, so in GDL the name of the operation is like the name of a C function. The ordered list enclosed in parenthesis that follows the operation name contains the declarations of the operation's formal inputs. The `->` syntax indicates that a list of the operation's formal outputs will follow. If the function has a single formal output, then it can be listed without parenthesis as in the first two examples above. If the operation has multiple formal outputs, they are enclosed in parenthesis and separated by comas.

The formal parameters themselves are made up of a literal representing the name of that parameter and a type. In this early version of GDL, types are specified as counting numbers. The number corresponds to the number of bits used to represent the parameter. So the first declaration above declares an operation that takes two 16-bit inputs and returns a 16-bit result. In the first version of the proposed CAE tools, the user will be constrained to a limited number of bit-widths corresponding to the specific widths of the operations available in the standard library. Future versions of the tools could perform automatic type promotions and generate required components on the fly.

Consider the following GDL code that defines a more complex operation built from primitive components. The following GDL code evaluates the polynomial x^2+3x+2.

```
polynomial(x:16)->result:16
{
    mult(x,x)->sum;
    add(sum,mult(3,x))->sum;
    add(sum,2)->sum;
    sum->result;
}
```

Definitions of operations have the same format as declarations, but are followed by an operation body enclosed in braces. Within the body, users can make references to previously declared operations including both standard library operations and user-defined operations. The GDL compiler creates dataflow graphs based on the textual GDL description. Internally, the GDL compiler represents the dataflow graph as a bipartite directed acyclic graph (DAG). One set of nodes in the bipartite

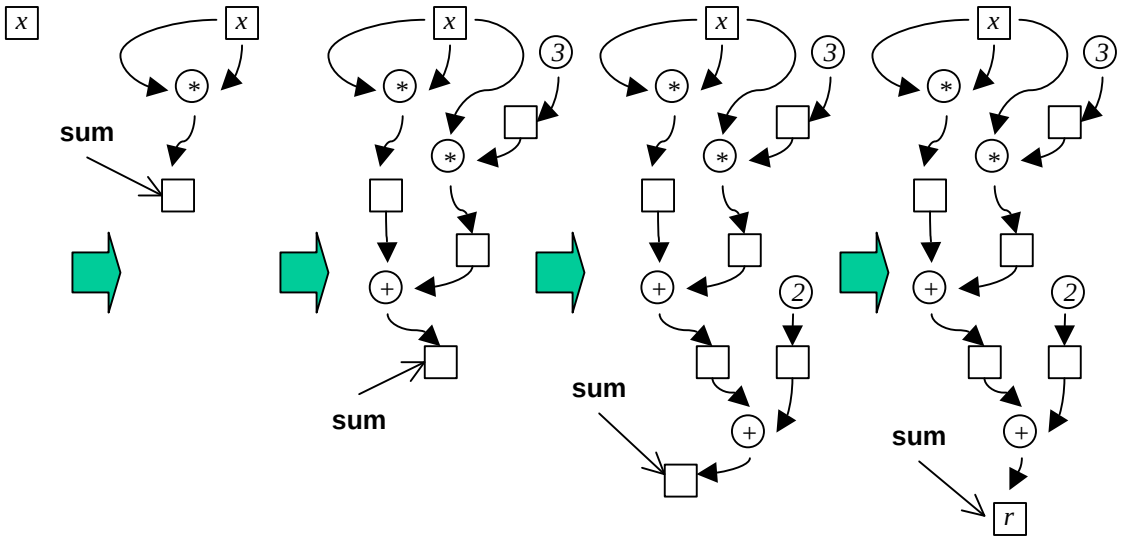

Figure 2: GDL compiler graph creation.

graph represents operations and the other set represents data produced by operations. Figure 2 demonstrates the basic construction process. In the figure, squares represent data nodes and circles represent operations.

When the compiler examines the operation's inputs it creates corresponding formal data node *x*. The operation body contains an ordered set of operation calls separated by semicolons. Each line of the body maps some set of available inputs to some set of outputs. The first line in the example above squares the formal input *x*. The *mult* operation takes *x* as both its inputs and returns its output to something called *sum*. The something is a GDL label called *sum*. GDL labels are used to represent intermediate values within computations. There is no need to declare these labels, the GDL compiler automatically finds them and handles them appropriately. When the compiler instantiates the *mult,* operation it automatically creates a data node to represent the output value. From the textual description, the compiler knows that the user wants to refer to that node as *sum.*

The remaining lines complete the computation. The second line computes the sum x^2+3x using the square of *x* calculated in the previous line. This line demonstrates GDL's ability to nest operation calls. The call to the *mult* operation is nested inside the call to the *add* operation. It also demonstrates the distinction between a label and the data node that the label is pointing at. Notice that the second line of GDL uses the label sum as both part of the input parameters and as the target of its output. Since our data flow graphs are DAGs, the compiler understands that the data node being referred to is a new data node. The data node that *sum* pointed at before stays the same, but the label *sum* is changed to point to the new data node created to represent the output. This language feature is provided to reduce the number of intermediate labels that must be created while describing a computation. Without it, the programmer must tediously create a unique name for each output in the graph.

The third line adds the constant term to the accumulating sum. GDL understands constants passed as parameters, but it handles them in a special way. The constant is translated into the *constant* operator and a data node as shown in Figure 1.

The fourth line performs a label assignment operation. This line does not correspond to an operation call and does not translate to any kind of hardware. It simply changes the result label to point to the data node that the sum label is pointing to. Again, this is a programming convenience for the user.

5. THE STANDARD LIBRARY

The standard library contains the definitions of the operations that are available to the application programmer. The operations are universal to all backend implementations of the tools. Some backends may only implement a subset. The standard library declaration file does vary from backend to backend in terms of the resource requirements of the operations.

There is an additional syntax, similar to the one C++ uses for templates, that assigns attributes to operations. The syntax is made up of key/value pairs. The CAE tools use these attributes to communicate information concerning the operations among the various phases of processing and also with the underlying vendor tools. Attributes are also used to communicate resource requirements and characteristics of the operations to lower level tools. The standard library declarations of the primitive operations provide this information. The names of the resources corresponding to the identifiers declared in the architectural description file. In the example below, the *add* operation requires eight CLB resources, has a latency of one clock, and is mapped to a VHDL implementation model called *add16*.

```
add<CLB=8,LAT=1,NAME=add16>(lhs:16,rhs:16)->result:16;
```

6. TEMPORAL PARTITIONING

Given the architectural description file, the computational structure of the application parsed from GDL, and the resource requirements of the operators, the CAE tools can partition the design. The resource requirements of the computational structure can be computed by flattening the parsed GDL code. Flattening replaces all operation references with the sub-graphs that they represent. Hierarchical information is still preserved for use by the two partitioners through other means. The way the human programmer chose to create functional abstractions within their code may give the temporal and spatial partitioner clues about how to partition the design. Once this structure is computed, the attributes describing resource requirements can be examined and summed to provide a resource requirement statistic for the overall system. If the resulting resource requirement is greater than the resources available on the CCM described by the architectural GDL, temporal partitioning of the design is required. The purpose of temporal partitioning is to break the overall computational structure into sub-structures that have resource requirements less than or equal to those available on the CCM's computing array. The temporal partitioner must also place an ordering on these partitions that correspond to the sequence in which they will execute on the computing array.

There are some rules that govern how the partitioner can go about adding nodes to a partition. If a data node is a primary input, a constant, or has been included in a partition earlier in the ordering it can be added. Data nodes are automatically added to a partition if the operation that they are dependent on is added to the partition. A node *A* is dependent on another node *B* if there is an in-bound edge from *B* to *A*. Operations can be dependent on an arbitrary number of data nodes. Each data node, however, is dependent on at most one operation. The partitioner can add an operation to a partition only if all of the data nodes that it is dependent on are already included in the partition. When the partitioner is finished, it must examine the resulting partition and characterize each of the data nodes included in the partition as an internal or external data node. Internal data nodes are those nodes that have no outbound edges that connect to an operation outside the partition. External data nodes are nodes that have outbound edges incident to operations outside the partition. The values produced at external data nodes must be preserved in memory for use by later partitions. The temporal partitioner will insert additional hardware into the partition to handle storing these values to memory.

One possible strategy for temporal partitioning is to perform a depth-first traversal of the structure attempting to add operations to the current partition. As an example consider the quadratic equation structure diagramed in Figure 2. Assume that the target CCM has a normalized configurable capacity of 16 units. Further, let the normalized resource requirements of the operations be mult=4, square=3, negate=1, sqrt=12, div=8 and add=1.

The process begins by adding one of the primary inputs. Assume the partitioner chooses *a*. The algorithm begins descending from *a* toward the outputs of the graph. There are two operations dependent on *a*, the multiplication by *2* and the multiplication by *c*. Arbitrarily, the algorithm selects the multiply by *c*. The *c* node is a primary input so it can be added to the partition. The multiply unit has a resource requirement of 4 and will easily fit into the current partition since there are 16 unallocated resource units. The partitioner adds *c*, the multiply, and the data node dependent on the multiply. This leaves 12 unallocated resource units in the current partition.

Continuing its descent, the partitioning algorithm selects the only dependent operation of the multiply that was just added, the multiply by *4*. Again, the partitioner can only add the operation if the data nodes it depends on have been added. The data node resulting from previous multiply is already in the partition and the constant *4* can be added. The *mult* operator requires four resource units and twelve are available, so the partitioner adds it and the data node dependent on it to the current stage. The next operation in the descent is the subtraction. The subtraction is dependent on the data node at the output of the squaring operator. To add the subtraction, the partitioner must first add the square operator. The square operator is dependent

on the primary input *b*, which can be added to the partition. The square operation's resource requirement is three. The current partition has eight resource units left, so the partitioner can add the square operation and its dependent data node. The subtraction operation requires a single resource unit, so it and its dependent data node can be added. After adding the two operators the partition has four resource units left. The next operation in the descent is the square root operator. Square root requires twelve resource units and cannot be added to the current partition. Retracing its steps back up the tree the partitioner discovers the other operator dependent on a, the multiply by two. This multiplication is dependent on the constant data node *2* and its resource requirement is four. The partitioner adds the *2*, the multiply, and the multiply's dependent data node to the current partition. That leaves the current partition with zero resources left to allocate, so the partitioner begins a new stage. The partition that was just complete is shown as Partition A in Figure 3.

The partitioner can choose to begin the next partition at one of five data nodes, *a*, *b*, *c*, i_0, or i_1. An attempt to continue working from *a*, quickly reveals that all of the operations dependent on *a* have already been added to a partition. The partitioner chooses to work with *b* next. The *b* operand has only one dependent that has not already been added to a partition, the negate operator. The negate is dependent only on *b* and requires only one resource unit, so the partitioner can add it to the new stage. That leaves fifteen resource units in the new partition. Next, the partitioner tries to add the addition operator that is dependent on the negate operator. That addition is dependent on the result of the square root operator, so it can only be added if the square root can be added. The square root is dependent on a previously computed result and it will fit in the current

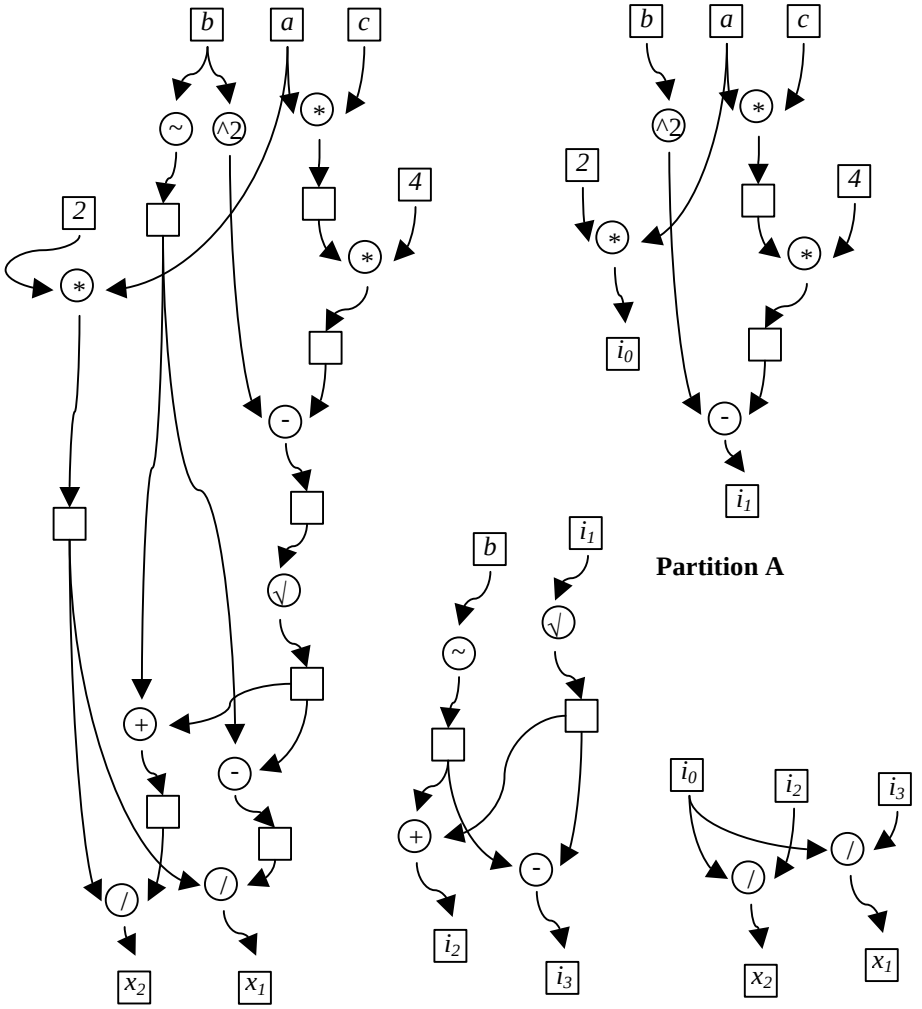

Figure 3: Temporal partitioning of the quadratic function.

partition, so both the addition and the square root can be added to the stage. This leaves two resource units in the current partition. Next, the partitioner tries to add the dependents of the addition, but cannot because the divide operator's resource requirements are too great. Travelling back up the graph it tries to add the subtraction that is dependent on the negate operator. The subtract requires a single resource unit and requires the output from the square root unit, which has already been added to the partition. The partitioner can add it to the stage. That leaves a single resource unit in the current partition. None of the remaining operators will fit in the stage, so the partitioner must begin a new stage. A diagram of the newly completed partition is labeled Partition B in Figure 3.

Examining the primary inputs and the external data nodes, the partitioner discovers that there are only three data nodes with dependent operations that have not yet been added to a partition: i_0, i_2, or i_3. The partitioner chooses to begin i_2. The only dependent operator is the divide that produces x_1. The divide requires eight resource units of the remaining sixteen and its other dependency, i_0, can be added to the partition so the partitioner adds them both. Having reached an output, the partitioner searches back up the structure to find another operation and discovers there is none. So, it chooses to continue from i_3. The required eight resource units are available and i_0 is already in the partition, so the partitioner adds the second divide to the current stage. Since all the operations have been added, the partitioner is done. The graph labeled Partition C in Figure 2 illustrates the last partition.

7. SPATIAL PARTITIONING

Spatial partitioning operates on the computational structures generated by the temporal partitioner. These structures represent partitions of the overall computation that the temporal partitioner generated. The spatial partitioner's job is to produce a mapping of the operators and data nodes onto the graph described by the architectural description file. Based on the example architectural description file, the primary constraints are the size of individual FPGAs, the number of memory ports that each FPGA can support, and the number of I/O pins available to communicate between devices. The spatial partitioner maps operation nodes and data node to nodes in the architecture graph. When it does this it must obey certain rules about the resources that are consumed. The architectural description file specifies how much of each resource can be mapped onto a given architectural node. Each operator in the theoretical backend implementation discussed here consumes some number of CLBs. The spatial partitioner cannot map a new operation node onto a FPGA if doing so would cause the total CLB resource consumption of the FPGA to exceed the limit set in the architectural description file. Notice that the FPGA nodes do not have a BW resource constraint. In the sample backend implementation, the only nodes that consume the BW are data nodes. So, an arbitrary number of data nodes can be mapped to a given FPGA node. Notice that the data nodes all specify that they have resource limitations of zero for the CLB and MPORT resources. This prevents the spatial partitioner from mapping operation nodes and memory ports onto interconnect nodes. Partitioning under these constraints is known to be an NP-complete problem [GarJ79] and consequently various heuristics have been developed to solve the problem.

There is a wealth of information in the literature that discusses spatial partitioning under constraints. The proposed tools will utilize a heuristic that has its origins in [KerL70]. Their heuristic was developed to partition *N* graph nodes into two partitions of equal size, such that the number of edges between them is minimized. It performs this operation in multiple passes. Each pass involves swapping pairs of nodes between the two partitions until each node has been swapped into the other partition. The algorithm chooses the order to swap nodes based on the maximum decrease or minimum increase, if no decrease is possible, in the number of edges between the partitions. When all the nodes have been exchanged the partitions have essentially been exchanged. Before the next pass, the heuristic examines all of the partitions that were generated during the current pass and chooses the one with the smallest cut set as the starting point for the next pass.

In [FidM82], it was suggested that instead of swapping nodes, single nodes be moved between the partitions. This innovation allowed some flexibility in the sizes of the partitions and allowed the introduction of a superior data structure for determining the node whose movement produces the maximum decrease in the cut set. The implementation from [KerL70] required a search of the underlying data structures to determine which pair of nodes should be swapped next. [FidM82] introduced a method that keeps the nodes sorted by their decrease in cut size. To prevent the heuristic from simply placing all the nodes in one of the partitions, which trivially minimizes the cut set, [FidM82] places bounds on the minimum and maximum sizes of the partitions. If the best node to move would cause one of the partitions to exceed these boundaries, the next best node is chosen instead. These improvements resulted in a heuristic whose computational performance was linear with respect to the size of the graph.

The next step in the evolution of the partitioning algorithm is found in [KriB87]. This paper introduced an improvement based on the properties of circuits. Interconnections of computing components are typically better represented by

hypergraphs than by graphs, because nets in circuits tend to have more than two connections. The novel aspect of [KriB87] was to incorporate this knowledge into the portion of the heuristic that determined which node to move next. Consider three nodes C_1, C_2 and C_3 that are all connected by one net. If C_1 and C_2 are both in one partition and C_3 is in the other then there is no advantage in moving C_2 to the other partition. The presence of C_1 in the other partition leaves the net in the cut set between the partitions. However, moving C_2 does make it possible to remove the net from the cut set in a later iteration if C_1 is moved. [KriB87] develops a method of incorporating this knowledge into the selection process for which node should be moved next.

All of the heuristics presented so far are strictly bipartitioning algorithms. They only handle two partitions. Obviously, the proposed CAE tools will need to partition computational structures across more than two FPGAs so k-way partitioning is required. The literature has presented heuristics for k-way partitioning by repetitive use of bipartitioning. Since these methods neglect the special properties inherent in problems involving multiple partitions, their results are not as good as the heuristics that operate on multiple partitions simultaneously. The heuristic presented in [SanL89] handles multiple partitions forms the basis of the partitioning algorithm used by the proposed CAE tools.

The nature of FPGA-based computing arrays provides some additional prospects for optimizing spatial partitions. Unlike VLSI applications, where each additional piece of circuitry has a direct cost associated with it, FPGA applications have fixed hardware resources available. When spatial partitions do not fully utilize the resources available to them there is no cost associated with adding circuitry to those partitions, i.e. the hardware is there whether it is used or not. [KuzB94] and [TogS96] present methods of replicating components across different partitions to remove edges that cross partition boundaries.

When the spatial partitioner has completed its efforts, there will be a set of mappings of computational structures onto architectural graphs for each of the temporal partitions created by the temporal partitioner. Synthesizable VHDL files can be automatically extracted from these mappings.

8. THE STRUCTURAL AND CONTROL INTERFACES

At the time of this writing the structural and control interfaces have not been fully developed. In principle the interface must provide an abstraction that exposes a uniform interface on the CAE tool side and a full-featured implementation on the hardware specific side. In addition, information from the temporal partitioner must be utilized to develop control algorithms for the reconfiguration cycles. The JHDL system described in [BelH98] is one possible candidate for the interface. The JHDL system has the important feature of encapsulating the structural and control specifications into a single description.

9. CONCLUSIONS

Th ideas for the proposed tool set presented here have addressed some of the missing elements that prevent configurable computing from being regarded as a complete computing solution. The most prominent of these missing elements is the lack of an automated way to convert high-level problem descriptions into low-level hardware. Programming configurable computing devices with ASIC-heritage design tools is akin to programming microprocessors in Assembly language. It limits dramatically the number of people to whom the technology is accessible. The tools proposed here have attempted to raise the level of accessibility a little higher. The intention of GDL is to serve as an intermediate language that is the target for even higher-level tools that process more traditional problem descriptions like C or graphical flows.

In addition to the missing software elements, there may be elements of hardware architectures that are also missing. Most CCM architectures in existence today were designed to be the targets of the ASIC-like development process. Such platforms may not be the best targets for tools that follow a different design path. Preliminary results from analyzing the design flows presented here tend to favor architectures that support many memory ports per FPGA and that provide high inter-FPGA connectivity. Existing commercial architectures do not support this level of flexibility. In the same way that microprocessor-based architectures have been adapted to support the behavior of applications compiled from high-level languages, CCM architectures may need to be adapted to provide hardware support for the inefficiencies created by high-level design tools.

REFERENCES

[BelH98] P. Bellows, B. Hutchings, "JHDL-An HDL for Reconfigurable Systems", in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[EldH96] J. G. Eldredge, B. L. Hutchings, "Run-Time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-Based FPGAs," *Journal of VLSI Signal Processing*, vol. 12, pp. 67-86, 1996.

[EldH94] J. G. Eldredge, B. L. Hutchings, "RRANN: The Run-Time Reconfiguration Artificial Neural Network," *Custom Integrated Circuits Conference*, pp. 77-80, San Diego, California, May 1994.

[FidM82] C. M. Fiduccia and R. M. Matheyses, "A linear-time heuristic for improving network partitions," *Proceedings 19th Design Automation Conference*, pp. 175-181, 1982.

[GarJ79] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Company, 1979.

[HudL98] R. D. Hudson, D. I. Lehn, P. M. Athanas, "A Run-Time Reconfigurable Engine for Image Interpolation," submitted for review in *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, April 1998.

[KerL70] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Systems Technical Journal*, vol. 49, pp. 291-307, Feb. 1970.

[KriB84] B. Krishnamurthy, "An improved min-cut algorithm for partitioning VLSI networks," *IEEE Transactions on Computers*, vol. C-33, pp. 438-446, May 1984.

[KuzB94] R. Kuzman, F. Brlez and B. Zajc, "A Unified Cost Method for Min-Cut Partitioning with Replication Applied to Optimization of Large Heterogeneous FPGA Partitions," *Proceedings of EURO-DAC*, pp. 271-276, Sept. 19-23, 1994.

[PetA96] J. Peterson, R. O'Connor, P. Athanas, "Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures" presented at FCCM`96, Napa, CA, April 1996.

[SanL89] L. A. Sanchis, "Multiple-Way Network Partitioning," *IEEE Transactions on Computers*, vol. 38, no. 1, Jan. 1989.

[ScaV98] S. M. Scalera, J. R. Vazquez, "The Design and Implementation of a Context Switching FPGA", *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[TogS96] N. Togawa, M. Sato and T. Ohtsuki, "A Performance-Oriented Circuit Partitioning Algorithm with Logic-Block Replication for Multi-FPGA Systems," *Proceedings of IEEE Asia Pacific Conference on Circuits and Systems*, pp. 294-297, Nov. 18-21, 1996.