Automatic Mapping of Multiple Applications to Multiple Adaptive Computing Systems

Sze-Wei Ong, Nabil Kerkiz, Bernadeta Srijanto, Chandra Tan, Mike Langston, Danny Newport and Don Bouldin

Microelectronic Systems Research Laboratory

University of Tennessee, Knoxville, TN 37996-2100

ong@microsys0.engr.utk.edu

Abstract

Adaptive computing systems (ACSs) can serve as flexible hardware accelerators for applications in domains such as image and digital signal processing. However, the mapping of applications onto ACSs using the traditional methods can take months for a hardware engineer to develop and debug. To enable application designers to map their applications automatically onto ACSs, a software design environment called CHAMPION was developed at the University of Tennessee. This environment permits high-level design entry using the Cantata graphical programming environment from KRI and hides from the user the low-level details of the hardware architecture. Thus, ACSs can be utilized by a wider audience and application development can be accomplished in less time. Furthermore, CHAMPION provides the means to map onto multiple ACS platforms, thereby exploiting rapid advances being made in hardware.

1 Introduction

Graphical programming environments such as Cantata [1-2] from KRI, LabVIEW from National Instruments and Simulink from MathWorks, allow applications to be graphically represented as a set of functional blocks connected by signal paths. By insulating the application programmer from low-level programming details, these environments allow faster and easier development of complex applications but the execution times on conventional CPUs are often long due to large input data or computationally intensive operators in the applications. For many types of commercial and military applications, which require high throughput, these long execution times are simply unacceptable.

With rapid advances in hardware, these complex applications can now be implemented in an ACS composed of multiple FPGAs serving as general purpose processing elements as well as interfacing devices. Since FPGA-based computing systems can be tailored to the particular computational needs of a given application, they have been shown to have considerable performance advantages over conventional processor-based systems for certain types of applications [4-6].

Traditionally, the task of developing applications for an ACS requires considerable knowledge in digital hardware design and entails a long and tedious process, often requiring months to generate the HDL representation and then to synchronize, partition, and synthesize the digital circuit. Significant effort is also required to resolve the issue of the intricate interactions between the hardware (ACS) and the software (host machine). The lack of supportive design environments results in an unacceptably long turn-around time for leveraging the benefits of this type of hardware. Therefore, it is necessary to develop an end-to-end mapping tool that allows the designers to reduce the time required to move from specification to hardware implementation.

This objective has been achieved by the CHAMPION software design environment which provides automatic mapping of applications in the Cantata graphical programming environment to ACSs. CHAMPION is a complete design environment that provides the tools needed to capture, simulate, and implement software applications on multiple ACSs. In this paper, we present the design flow of this end-to-end design environment. The strength of this ACS-dedicated design flow includes its capability of yielding digital systems with high clock rates in low mapping time. It also allows Cantata applications to be mapped onto multiple ACS hardware architectures such as the Wildforce board and Wildcard developed by AMS [3], and the SLAAC board [6] developed by the University of Southern California. Another advantage is that the design flow allows graphical programming environments other than Cantata to be easily adapted as the design entry for CHAMPION.

The approach taken by CHAMPION is similar to that of several other research programs at Colorado State University and Northwestern University [15-16]. However, in our case, we perform synthesis and place/route on our library cells in advance. Thus, we have accurate information on the size and delay of each cell and only have to re-synthesize small net-lists that represent the collection of cells that fit in each FPGA. The competing approaches merge the VHDL code into a single, large file that must be fully re-synthesized and then partitioned at a finer grain than our approach. Hence, CHAMPION is presented with a net-list that is 10-100x smaller and can be expected to execute in 100-1000x less time while producing performance results within a few percent of the others.

This paper describes the entire compilation path of CHAMPION. The next section gives an overview of the flow. Section 3 describes the process of incorporating a new function or module in CHAMPION. The details of all the components in the design flow are presented in Section 4 to 9. Section 10 presents the results of several applications that have been implemented using CHAMPION. Section 11 concludes and discusses the possible extensions for this research work.

2 System Overview

The design flow of CHAMPION is shown in Figure 1. Cantata is used as a function-oriented programming environment where all the application programs are developed using predefined functions or modules called glyphs. Currently, a set of library glyphs has been developed in the CHAMPION project. A set of tools has been developed to automate the process of developing, verifying and installing the new glyphs in the CHAMPION library.

Once the application is developed using Cantata, the Cantata program is translated into a more graph-oriented database, preserving the original glyphs and their interconnections. Then, each interconnection is checked to verify that the bit-widths of the connecting ports are the same.

Due to the difference in the processing time of each glyph, data traveling over different concurrent paths may arrive at the inputs of a multi-input glyph at different times. To insure that each glyph generates the correct time-sequenced output, data synchronization is then performed. In CHAMPION, data synchronization is achieved by introducing delay buffers into the system. The synchronization software determines the lengths and locations of the delay buffers necessary to balance the various data paths. An optimization algorithm is employed to calculate a set of buffer lengths and insertion points that maximizes the amount of buffer sharing and therefore minimizes the total number of delay buffers.

Partitioning is then performed at the glyph-level, where each glyph element is represented by one node. This yields very low net-list complexity (hundreds vs. tens of thousands). Therefore, the partitioning process has a very short runtime (seconds vs. hours). Another advantage is that the functional flow information is preserved. Thus debugging and simulation of the system are facilitated even after the partitioning.

After partitioning, the internal data structure or format is translated into a structural VHDL representation. The required I/O ports for each of the subnet-lists are then added to the VHDL files. The VHDL files can then be synthesized and merged with the pre-compiled VHDL components corresponding to the Cantata glyphs. Each subnet-list is then placed and routed. The resulting configuration files are downloaded to the corresponding programmable logic component on the ACS board.

In the next few sections, detailed descriptions of each component of the design flow are presented.

3 Library Cell Development and Verification

Application programs can be constructed by interconnecting CHAMPION glyphs using Cantata. If certain glyphs needed for the application cannot be found in the precompiled CHAMPION library, these glyphs can be created and added. First, the designer must develop the fixed-point C or C++ program for the glyph. The reason for using fixed-point arithmetic is to allow the C/C++ program to mimic hardware operations. For complex functions, the C/C++ program can be formed as a macro of lower-level functions.

Next, VHDL code corresponding to each of the C/C++ programs must be generated. The functionality of the VHDL code must be identical to that of the C/C++ program. Identical test vectors are applied to both the C/C++ program and the VHDL code. The simulation results are compared to verify that bit-wise identical behavior is achieved. The steps for developing the new glyph are shown in Figure 2.

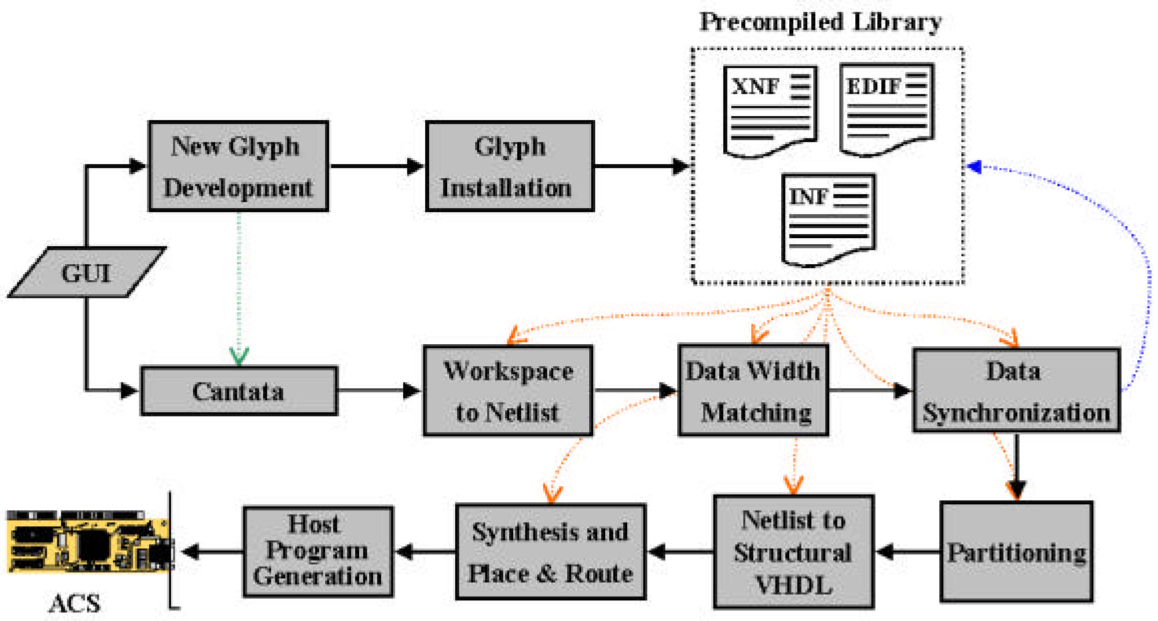

Figure 1: Overview of the design flow in CHAMPION.

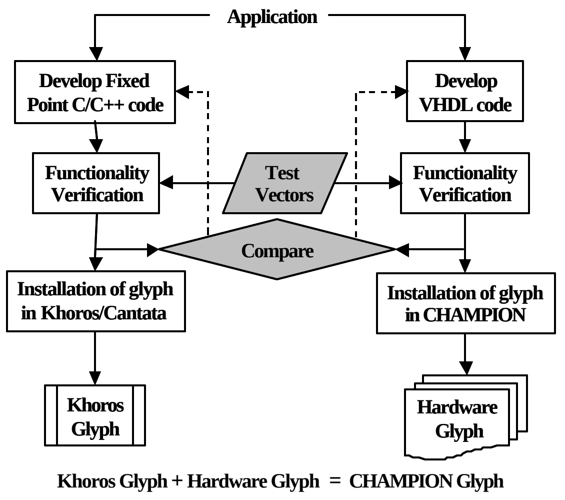
Application
Develop Fixed Point C/C++ code
Develop VHDL code
Functionality Verification
Test Vectors
Functionality Verification
Compare
Installation of glyph in Khoros/Cantata
Installation of glyph in CHAMPION
Khoros Glyph
Hardware Glyph
Khoros Glyph + Hardware Glyph = CHAMPION Glyph

Figure 2: Steps for developing a new glyph.

To accelerate the glyph development process, the commercial software, *A|RT Library* and *Builder* [7] from Frontier Design were integrated into the CHAMPION design flow. The *A|RT Library* and *Builder* provide the ability to generate the VHDL description of the hardware directly from a C-code specification of the application. The user no longer has to do this manually. The new steps for developing the glyph using *A|RT Library and Builder* are shown in Figure 3.

Once the functionalities are verified, the C/C++ program will be converted to a Cantata glyph and installed in Cantata using the tools from KRI. The corresponding VHDL description will be synthesized and converted to multiple technology-dependent net-list files (XNF and EDIF files) for multiple ACS boards. Also generated is a glyph information file (with extension *INF*) stores different sizes, latencies and I/O data bit-widths of the glyph for multiple ACS architectures. The net-list and information files are then installed in the CHAMPION library as the hardware counterpart of the Cantata glyph.

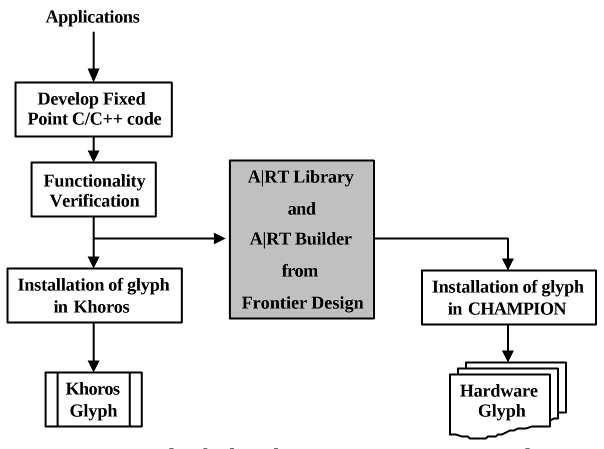
Applications
Develop Fixed Point C/C++ code
Functionality Verification
A|RT Library and A|RT Builder from Frontier Design
Installation of glyph in Khoros
Installation of glyph in CHAMPION
Khoros Glyph
Hardware Glyph

Figure 3: New glyph development using ART tools.

4 Converting Cantata Workspace to Net-list

Using Cantata, the designer can develop the application by interconnecting CHAMPION glyphs to form the Cantata workspace. Simulation, data analysis and visualization can then be performed in Cantata. Once the desired functionality of the application is achieved, the Cantata workspace is translated into a directed hyper-graph where each glyph is represented as a node and the interconnections between glyphs are represented as directed hyper-arcs. Based on the information from the *INF* file, weights are assigned to the nodes and hyper-arcs of the directed graph. This net-list format simplifies the use of graph theory and network optimization theory during the data synchronization and partitioning process.

5 Data Width Matching

In a hardware application, some functions may produce results that require fewer bits for their outputs than for their inputs. Consequently, cascaded glyphs may progressively require different data bit-widths. When one path of operations is connected to a parallel path, a mismatch in the number of bits for these inputs may occur. This mismatch is labeled *positive* since the bit-width of the net carrying the data is larger than the bit-width of the net receiving the data. An example of a positively mismatched data path is shown in Figure 4. A software tool within CHAMPION was developed to analyze each data path and to truncate the additional bits when appropriate. The truncating process (shown in Figure 5) is performed by inserting a "truncating" glyph at the mismatch data path. The "truncating" glyph will remove the additional data bits from the signal.

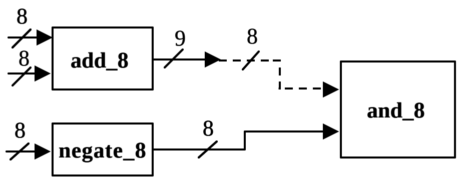

Figure 4: A positively mismatched data path.

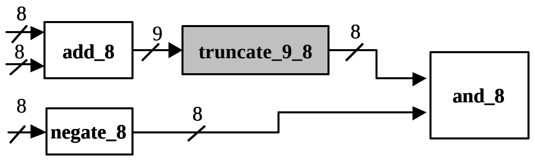

Figure 5: Insertion of a "truncating" glyph.

Similarly, some hardware functions may produce results that require more bits for their outputs than for their inputs, especially to avoid round-off errors. Consequently, a negatively mismatched data path such as the one shown in Figure 6 may occur. In this case, a "padding" glyph (shown in Figure 7) has to be inserted at the mismatched data path. The "padding" glyph will append 0's to the incoming signal.

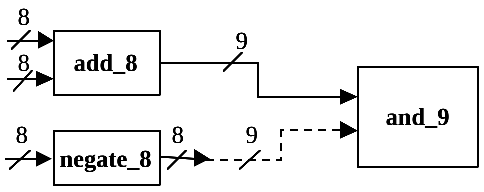

Figure 6: A negatively mismatched data path.

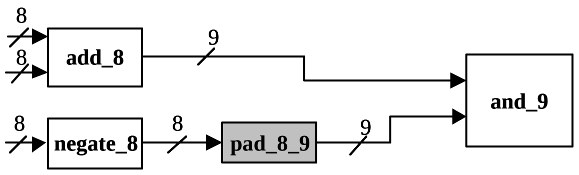

Figure 7: Insertion of a "padding" glyph.

6 Data Synchronization

In a graphical programming environment such as Cantata, executions of the programs are data driven. That is, each Cantata glyph will begin execution only when all its input data is available. However, hardware systems are clock driven. At each clock cycle, each hardware glyph will process whatever data is presented at its inputs.

Due to the difference in the processing time of each hardware glyph, data traveling over different concurrent paths may arrive at the inputs of a multi-input glyph at different times. To insure that each glyph generates the correct time-sequenced output, it is necessary that each glyph receive all its input data precisely at the same time.

Figure 8 illustrates this data synchronization problem. In this system, there are two primary input modules, two primary output modules and five processing modules labeled P_1 through P_5. The processing time for each processing module is labeled T_u. Note from the figure that the primary input and output modules have zero processing time. This is due to the fact that these modules are storage devices for the input and output data. No data processing will be performed in these modules. We assume that all the input data are readily available in the input modules. Therefore all input modules are time synchronized. We also require that all the outputs are made available at the same time. This is because if another digital system is connected to this system, these output modules will become the input modules for the new digital system. In addition, this requirement also allows us to break a large digital system into smaller subsystems for synchronization.

Examining this system, it is clear that the two input signals entering P_4 are not synchronized; the lower signal arrives one time unit sooner than the upper signal. Also, the primary output, O_2, becomes available two unit times earlier than O_1.

To synchronize this system, delay buffers can be introduced into the digital system. These delay buffers are inserted at various locations to delay the signal on the data path which has lower processing time, and therefore match it with the data path with higher processing time. For this synchronization approach, the pipelined time is zero. That is, the input signal can be presented to the system continuously without having to hold each new input signal, as in the case of the traditional synchronization approach using edge-triggered registers. This feature is essential in many designs which require high throughput rates.

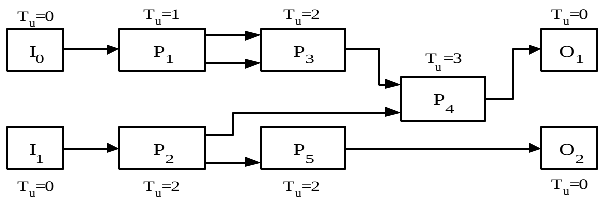

Figure 8: An unsynchronized digital system.

Synchronizing the system requires one to determine the lengths and locations of the delay buffers necessary to balance the various data paths. Straightforward methods for performing this task are not difficult to develop. For example, each multi-input processing module might be examined to check if all the inputs have the same delay from the primary input modules. If all the inputs do not have the same delay, the input with the largest delay is identified. For those inputs with delays which are less than this maximum value, delay buffers must be inserted into each of these "early" input lines in order for all the inputs to the module to have equal processing delay from the primary input modules.

Applying this method to Figure 8, we find that two delay buffers with a total of three unit time delays are required to synchronize the system (as shown in Figure 9). One buffer with a unit time delay has to be inserted in the path between P_2 and P_4. Another buffer with a two-unit time delay has to be placed between P_5 and O_1.

This straightforward method, however, does not necessarily provide the optimum solution in the sense that the total number of delay buffer units used is not necessarily a minimum. The delay buffers can always be moved forward or backward along the data path in order to achieve a maximal amount of delay buffer sharing. This can be seen in Figure 10 where inserting the delay buffer between I_1 and P_2 allows both the I_1-P_4 and the I_1-O_2 paths to share the delay. Therefore, a total of two unit time delays are required, compared to the three unit time delays in Figure 9.

An optimization algorithm can be used to calculate a set of buffer lengths and insertion points that maximizes the amount of buffer sharing and therefore minimizing total length of the delay buffers required. In the CHAMPION environment, the algorithm developed by Hu [8] was adopted and is described next.

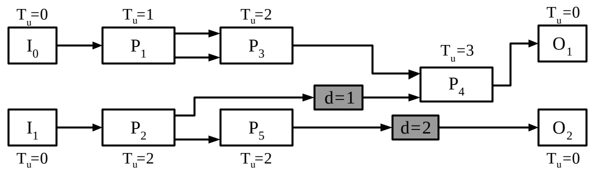

Figure 9: Synchronization using the straightforward approach.

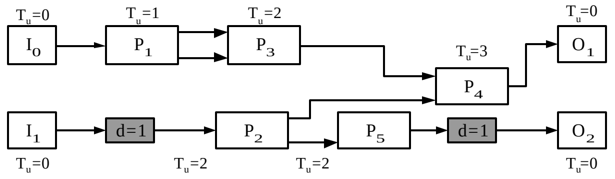

Figure 10: An optimum synchronization of Figure 8.

6.1 Problem Formulation

6.1.1 Generating the Signal Flow Graph

To solve the buffer minimization problem using the algorithm proposed in [8], the system has to be represented using a signal flow graph (SFG) in which each processing module is represented as a node and each data path between the modules is represented as a directed edge. The weight of the edge directed from node u (corresponding to processing module P_u) to node v (corresponding to processing module P_v) is an unknown delay variable. This delay variable d_{uv} corresponds to the delay buffer which has to be inserted between P_u and P_v for synchronization. It will be determined during the synchronization process (solving of the delay buffer minimization problem). The value of d_{uv} computed during the synchronization process equals the size of the delay buffer required to be inserted between node u and node v.

Besides the nodes and edges that represent the modules of a system, two virtual nodes are introduced. One is termed the primary input node, while the other is called the primary output node. All of the input modules of the digital system will be combined and represented using the primary input node. Similarly, all the output modules of the digital system will be combined and represented using the primary output node. Therefore, in the SFG, all input signals to the system originate at the primary input node, and all outputs from the system terminate at the primary output node. Both of these special nodes are assumed to consume zero processing time.

Based on the rules stated above, a SFG representation can be constructed for any given digital system. For instance, for the digital system shown Figure 8, the corresponding SFG is shown in Figure 11. The synchronizing problem is then reduced to assigning optimal values to the delay variables in the SFG.

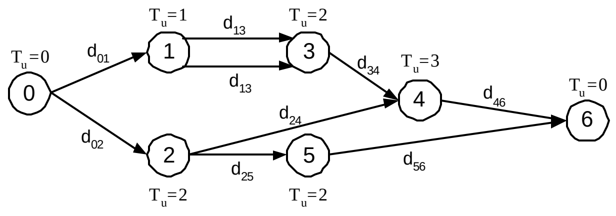

Figure 11: The SFG representation of the digital system shown in Figure 8.

6.1.2 System Containing Hyper-arcs

An output connected to more than one input is called a hyper-arc (shown in Figure 12a). If hyper-arcs exist in the digital system, special representations of these nets need to be considered. One easy solution is to treat the hyper-arc as having multiple ordinary outputs and construct the SFG as shown in Figure 12b. The second representation can be obtained by inserting a virtual node, V1, with zero processing time as in Figure 12c. Another representation, which is proposed in [8], is shown in Figure 12c. This representation uses a binary tree structure which systematically introduces virtual node to the SFG.

The example in Figure 13 illustrates the effects of using the three different types of SFG representations in Figure 12. In the example, a total of 12 units of delay buffers are required to synchronize the hyper-arc if the SFG representation shown in Figure 12b is used. With the insertion of a virtual node, the total number of delay buffers can be reduced to 6 as shown in Figure 13b. If the binary structure is used, the total number of delay buffers can be reduced to 5 as shown in Figure 13c. This example demonstrates that some of the delay buffers can be shared along the hyper-arc through the insertion of the virtual node. The use of a binary tree structure in representing the hyper-arc can greatly exploit the buffer sharing property. To exploit the buffer sharing property fully, an algorithm must be used to arrange the processing nodes in the hyperarc [8]. Different arrangements of the processing nodes driven by a hyper-arc may vary the total number of buffers along the hyper-arc because each arrangement may cause a different length delay buffer to be shared among the nodes driven by the hyper-arc.

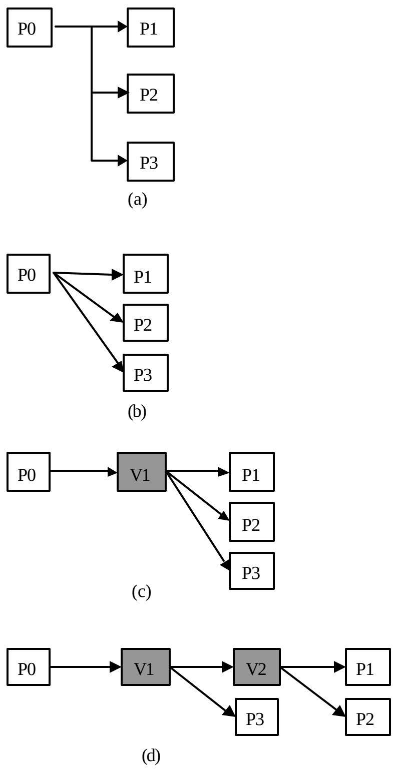

Figure 12: Different representations of an hyper-arc.

6.1.3 Forming the Buffer Minimization Problem

To synchronize a system, all input signals to any given module must arrive at the same time. In other words, the accumulated delays along all the distinct paths from the primary input node, *i*, to a particular SFG node should be equal. The accumulated delay along a particular data path is simply the sum of all the weights of the edges and the processing time, *T*, of all the processing modules on the path from *i* to *u* in the SFG.

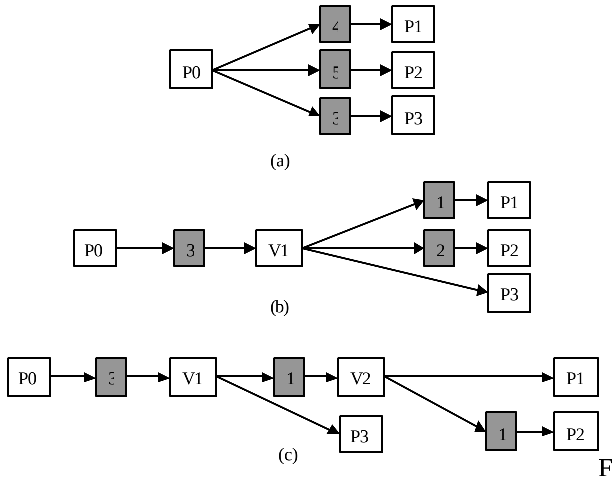

Figure 13: Delays of the hyper-arc.

Denoting the j-th path between node i and u as $P_j(u)$ and the total delay along $P_j(u)$ as $D_j(u)$, it follows that $D_j(u)$ equals the sum of the delays associated with all the edges along $P_j(u)$, i.e., it can be expressed as:

$$D_j(u) = \sum_{(x,y)\in P_j(u)} T_x + d_{xy} \qquad (1)$$

where (x,y) represents an edge from node x to node y.

The synchronization problem can now be defined as assigning a value to each delay variable, d_{xy}, such that the values of $D_j(u)$ for $j = 1,2,\ldots k_u$ are identical for each and every u. Here, k_u denotes the number of distinct paths from the primary input to processing modules P_u.

Thus, the task of synchronizing a SFG may be seen to be equivalent to assigning values of D_u to all graph nodes and values d_{uv} to all graph edges such that

$$D_v - D_u = (T_u + d_{uv}) \qquad (2)$$

holds for each and every pair of nodes, u and v, that are connected by an edge.

The synchronization-minimization problem can then be expressed as an ILP (integer linear programming) problem:

$$Mininmize \sum_{(u,v)} d_{uv} \qquad (3)$$

$$Subject\ to: -D_u + D_v - d_{uv} = T_u \qquad (4)$$

$$\begin{gathered} u \in V, v \in V, (u,v) \in E \\ D_u \text{ integer; } d_{uv} \geq 0, \text{integer;} \end{gathered} \qquad (5)$$

where E and V represent the sets of all edges and nodes in the SFG, respectively, (u,v) denotes a directed edge from node u to node v. The variables to be determined here are the D_u and d_{uv} values. The T_u's are known processing time delays of each node. Equations (3)-(5) form a standard description of an ILP:

$$\begin{aligned} &\min: d_{01} + d_{02} + d_{13} + d_{13} + d_{24} + d_{25} + d_{34} + d_{46} + d_{56} \\ &\text{subject to} \\ &\qquad -D_0 + D_1 - d_{01} = 0 \\ &\qquad -D_0 + D_2 - d_{02} = 0 \\ &\qquad -D_1 + D_3 - d_{13} = 1 \\ &\qquad -D_2 + D_4 - d_{24} = 2 \\ &\qquad -D_2 + D_5 - d_{25} = 2 \\ &\qquad -D_3 + D_4 - d_{34} = 2 \\ &\qquad -D_4 + D_6 - d_{46} = 3 \\ &\qquad -D_5 + D_6 - d_{56} = 2 \end{aligned} \qquad (6)$$

6.2 Solving the Minimization Problem

In [8], it is shown that the constraint matrix of the ILP problem in (3)-(5) satisfies the definition of *unimodularity*. Therefore the ILP problem can be reduced to a LP problem [9]. The buffer synchronization-minimization problems can then be solved using a linear programming algorithm such as the Simplex method. Integer solutions are always guaranteed.

7 Partitioning

One of the main problems in partitioning is complexity. The research in partitioning theory has seen many algorithms with good results even with today's design complexity. The main disadvantage of these approaches is that they are based on gate-level net-lists, thus requiring hours to execute.

In CHAMPION, we drastically reduced the complexity by keeping the structural information and configuring the programmable logic components and their interconnects in the ACS board into a linear array. With this topology, the partitioning operates on a module-level net-list and its order proceeds in a forward-only direction. Thus, partitioning is performed with net-lists containing hundreds of nodes instead of tens of thousands.

For our design flow, the partitioning problem is based on the following constraints: capacity per partition, number of I/O pins per partition, RAM access, and temporal partitioning. The first two constraints are used to meet the limitations of the programmable logic components. The third constraint deals with the memory access for each the programmable logic component. The architectures of some of the ACSs require that a fixed number of local RAMs are available to each FPGA for data writing and data reading. Therefore, a partition can contain only a certain number of RAM access modules.

The fourth constraint deals with temporal partitioning of the ACS board. If the entire application cannot fit in one board configuration, then multiple configurations of the board are necessary and storage of intermediate results between board configurations is needed. In this case, one pair of RAM-access modules must be added to each configuration.

To solve the partitioning problem, three different approaches were investigated in our research. In the first and second approaches, we implemented two existing algorithms: a hierarchical partitioning method based on topological ordering (HP) [10] and a recursive algorithm based on the Fiduccia and Mattheyses bipartitioning heuristic (RP) [11]. Some modifications have been made on these algorithms to take advantage of the acyclic nature of our net-list, and to handle the RAM access constraint and the temporal partitioning constraint. A new recursive partitioning method based on topological ordering and levelization (RPL) [12] was also introduced. In addition to handling the partitioning constraints, the new approach efficiently addresses the problem of minimizing the number of FPGAs used and the amount of computation, thereby overcoming the weaknesses of the HP and RP algorithms.

8 Net-list To Structural VHDL, Synthesis, Place/Route And Host Program

After partitioning, the graph-based net-list for each partition is translated into structural VHDL. The VHDL files describing the ACS I/O ports and the pre-compiled VHDL components corresponding to the Cantata glyphs are then merged with the structural VHDL. The resulting files are then synthesized, and then placed and routed separately.

The final step in the CHAMPION design flow is the generation of the host program which downloads each configuration file to the corresponding programmable logic component on the ACS. The host program also initializes the ACS board and reads the input data from the host workstation, sends the data to the ACS and writes output back to the host workstation.

9 Mapping onto Multiple ACSs

The CHAMPION design flow allows Cantata applications to be mapped onto multiple ACS hardware. In order to support different hardware architectures, multiple technology-dependent net-list files (XNF and EDIF files) were generated for each glyph in the CHAMPION library. Each glyph information file contains three sizes and latencies for the three ACS boards shown in Figure 14. Based on the ACS board selected by the designer, CHAMPION will select the corresponding technology-dependent net-list file and glyph information (size and latency).

The three ACS boards contain different programmable logic components. Therefore, for the partitioning problem, the capacity per partition, number of I/O pins per partition, RAM access, and temporal constraints are different for each board. Three constraint files were developed to store the constraints corresponding to the three ACSs.

The three ACS platforms also use different communication and control circuits, which are described using VHDL files specific for each ACS board. The ACS-specific file is integrated by CHAMPION into the structural VHDL files produced by the partitioning step. The resulting files are synthesized and then placed and routed.

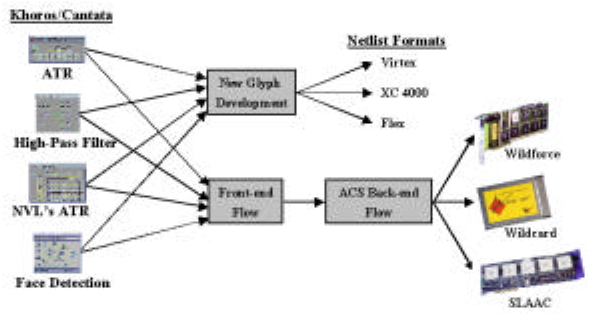

Figure 14: Mapping onto multiple ACSs.

10 Results

Initial verification of the CHAMPION design flow has been completed with additional testing underway. An automatic target recognition (ATR) algorithm [13-14] was implemented to serve as a benchmark for determining the improvement in mapping time for the design flow since it had been manually mapped to the Wildforce-XL board. The 250-hour manual process was performed automatically by CHAMPION in 5 minutes and 23 seconds, demonstrating a productivity improvement of over 2,000x.

Table 1 compares the execution time of the ATR algorithm in Cantata and on the Wildforce board. The manual Wildforce implementation has a faster execution time than the Wildforce implementation achieved using CHAMPION. The main difference between the two implementations is in the time required to reconfigure the board. During the manual mapping performed in [14], the START algorithm is partitioned in a way that some of the partitions can be reused in the next configuration. Therefore, the reconfiguration time is reduced. However, in the automatic mapping, the partitioning tool did not perform partitioning in such a manner. As a result, the reconfiguration time for the CHAMPION implementation is higher. However, compared to the Cantata implementation, the execution time of the CHAMPION implementation is 105 times faster.

	Cantata (seconds)	Wildforce (Manual) (seconds)	Wildforce (CHAMPION) (seconds)
Board Configuration	—	5.125	9.147
Data Transfer & Management	—	0.583	0.735
Data Processing	1054	0.010	0.010
Total Execution Time	1054	5.718	9.892

Table 1: Comparison of execution time [12].

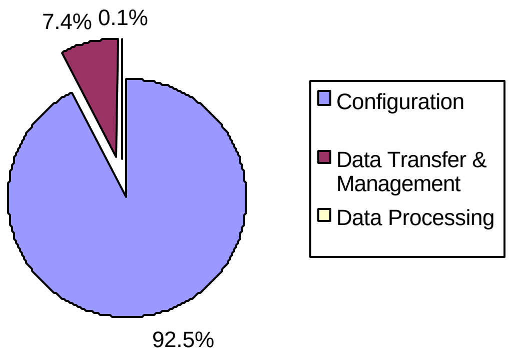

Figure 15: Breakdown of execution time for Wildforce.

Applications	Number of Glyphs	Number of Nets	Mapping Time (sec)	
			Champion Mapping	Synthesis P & R
High-Pass	18	46	75	2358
NVL's ATR	45	71	24	3404

Table 2: Mapping results using CHAMPION.

Applications	Cantata Execution Time (sec)	Wildforce Execution Time (sec)		
		Configure Board	Data Transfer	Data Processing
High-Pass	14.2	1.049	2.669	0.003
NVL's ATR	15676	0.807	153.128	0.2694

Table 3: Execution time on the Wildforce ACS.

	Wildforce	Wildcard[1]
Board Configuration	1.049	0.110
Data Transfer	2.669	3.037
Data Processing	0.003	0.0007

Table 4: Results for high-pass filter.

The execution time in the Wildforce can be broken down into board reconfiguration time, data transfer and management time, and the actual hardware execution time, as shown in Figure 15. The time to process an image is greatly dominated by the time needed to configure the board. For the ATR algorithm, the actual time to process one image is only 10 milliseconds, which is 0.1% of the total execution time. If the reconfiguration time could be eliminated, the hardware implementation would be over 1400 times faster than the Cantata implementation. This can be achieved by using an ACS board with larger FPGAs.

Two other applications, a high-pass filter for image processing and the ATR algorithm from the Army Night Vision Lab (NVL), have also been successfully mapped onto the Wildforce using CHAMPION. Results for these two applications are shown in Table 2, 3 and 4.

11 Conclusions and Future Work

In this paper, a design flow for automatic mapping of Cantata graphical applications onto multiple ACSs has been presented. Relative to contemporary technology, this design flow:

- Demonstrated a productivity improvement of 2,000x over manual methods (5 minutes 23 seconds vs. 250 hours),
- Utilized an optimization algorithm to synchronize the design using a minimum of well-placed delay buffers, and
- Partitioned the applications for multiple ACSs containing multiple FPGAs.

It is particularly significant that CHAMPION can retarget a new ACS board so quickly and easily. Given a high-level description of a new board that one may be considering designing or acquiring, a single-page data file is composed that lists the number of FPGAs, their size and I/O, the RAMs available and their interconnections on the board. The previously captured Cantata application is then resubmitted to CHAMPION which performs the mapping within an hour. Thus, application designers can determine which ACS architecture is the best match to the application and then either build or purchase the appropriate one.

This retargeting capability also permits an application designer to exploit the rapid advances in FPGA offerings. For example, as soon as a new ACS

[1] Calculated results for Wildcard running at 100 MHz

board is announced, the board-specific data file could be generated and the mapping/partitioning performed by CHAMPION. Upon arrival of the new board, the design could be downloaded and executed. The common situation of waiting months from the arrival of a new board until the application can be manually retargeted would be avoided.

In terms of extending CHAMPION, one possibility is to permit it to accept inputs from other graphical programming environments such as LabVIEW from National Instruments and/or Simulink from MathWorks. To include these programming environments, the only component in the design flow that needs to be modified is the front-end translator that converts the Cantata workspace into a CHAMPION net-list. The rest of the steps in the design flow will remain the same.

CHAMPION can also be extended to include a flow for automatic mapping into a single-chip ASIC. Since no partitioning is required, several of the back-end steps are skipped and the composite VHDL file is synthesized as a single unit using Synopsys. We have already begun this effort.

REFERENCES

[1] J. Rasure and S. Kubica, *The KHOROS Application Development Environment*, Khoros Research Inc., Albuquerque, NM, http://www.khoral.com.

[2] D. Argiro, S. Kubica and M. Young, "Cantata: The Visual Programming Environment for the Khoros System" Khoral Research, Inc., Albuquerque, NM, http://www.khoral.com.

[3] Annapolis Micro Systems, Annapolis, MD, http://www.annapmicro.com.

[4] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal, "The RAW Benchmark Suite: Computation Structures for General Purpose Computing." *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April, 1997.

[5] N. Ratha, A. Jain, D. Rover, "Convolution on Splash 2," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April, 1995.

[6] Systems Level Applications of Adaptive Computing (SLAAC), http://www.east.isi.edu/projects/SLAAC.

[7] D. Johnson, "Architectural Synthesis from Behavioral C Code to Implementation in a Xilinx FPGA," *Business Development Manager, Frontier Design Inc.*, http://www.frontierd.com.

[8] X. Hu, S. C. Bass and R. G. Harber, "Minimizing the number of delay buffers in the synchronization of pipelined systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, No. 12, pp.1441-1449, Dec 1994.

[9] A. J. Hoffman and J. B. Kruskal, "Integral boundary points of convex polyhedra," *Linear Inequalities and Related Systems*, ed. H. W. Kuhn and A. W. Tucker. Princeton, N.J.: Princeton University Press, 1956, pp. 223-46.

[10] B. Stanley, "Hierarchical Multiway Partitioning Strategy with Hardware Emulator Architecture Intelligence," Georgia Institute of Technology, Ph.D. Dissertation, 1997.

[11] R. Kuznar and F. Brglez, "PROP: A Recursive Paradigm for Area-Efficient and Performance Oriented Partitioning of Large FPGA Net lists," *International Conference on Computer-Aided Design*, pp. 644-649, November 1995.

[12] N. Kerkiz, "Development and Experimental Evaluation of Partitioning Algorithms for Adaptive Computing Systems," University of Tennessee, Ph.D. Dissertation, 2000.

[13] Levine, B., Natarajan, S., Tan, C., Newport, D. and D. Bouldin, "Mapping of an Automated Target Recognition Application from a Graphical Software Environment to FPGA-based Reconfigurable Hardware," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April, 1999.

[14] B. Levine, A System for the Implementation of Image Processing Algorithms on Configurable Computing Hardware, University of Tennessee, Master Thesis, 1999.

[15] P. Banerjee, et al., "A MATLAB Compiler for Distributed Heterogeneous Reconfigurable Computing Systems *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April, 2000.

[16] J. Hammes, R. Rinker, W. Böhm, W. Najjar, B. Draper, and R. Beveridge, "Cameron: High Level Language Compilation for Reconfigurable Systems," *Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, CA, Oct. 12-16, 1999.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the support of DARPA grant F33615-97-C-1124.