A Verifiable High Level Data Path Synthesis Framework

Görker Alp Malazgirt*, Ender Culha†, Alper Sen*, Faik Baskaya†, Arda Yurdakul*
*Department of Computer Engineering
†Department of Electrical Engineering
Bogazici University
Bebek 34042, Istanbul, Turkey
alp.malazgirt, ender.culha, alper.sen, baskaya, yurdakul @ boun.edu.tr

***Abstract*—This work presents a synthesis framework that generates a formally verifiable RTL from a high level language. We develop an estimation model for area, delay and power metrics of arithmetic components for Xilinx Spartan 3 FPGA family. Our estimation model works 300 times faster than Xilinx's toolchain with an average error of 6.57% for delay and 3.76% for area estimations. Our framework extracts CDFGs from ANSI-C, LRH(+) [1] and VHDL. CDFGs are verified using the symbolic model checker NuSMV [2] with temporal logic properties. This method guarantees detection of hardware redundancy and word-length mismatch related bugs by static code checking.**

I. Introduction

Current system designs aim to pack as much capability as possible in order to meet application specific demands. This has brought the importance of design automation and emergence of technologies like reconfigurable computing in order to cope with the increasing complexity both algorithmically and quantitatively. Therefore, FPGAs become widely used for prototyping, designing and testing Digital Signal Processing (DSP) applications. However, the majority of DSP algorithm developers are not familiar with HDL design. They prefer to use high level programming languages (HLL) to design, prototype and test their algorithms. That causes the necessity of synthesis tools which converts HLL into HDL with as little overhead as possible.

Short time to market conditions increase the importance of early estimation of delay, area and power behaviors of the applications in hardware. The commercial tools which benchmark about these behaviors like Xilinx [3] and Altera [4] can estimate these behaviors by going through the steps of synthesis, placement and routing. However, this can take minutes to hours depending on the application.

The role of verification has been increasing among research on efficient reconfigurable architectures in different domains because a good design is not enough to guarantee a working system. For example, in DSP domain, increasing complexity has been dealt with switching from low level programming languages (assembly) to high level programming languages (C/C++, Java). High Level Synthesis (HLS) tools have been providing application description mapping HLL to Register Transfer Level (RTL).

Our framework proposes to address the aforementioned challenges by creating a framework which creates RTL from HLLs (ANSI-C, LRH(+)) and verifies four properties which might be increased. Our estimation models of area, delay and power behavior of arithmetic components are used while generating RTL. The estimation model that we present in this work is for Xilinx Spartan 3 platform. A typical HLS chain includes steps such as scheduling, resource allocation, binding and register optimization. However, this paper presents generation and verification of HLL to Golden-RTL generation, but the implementation of the toolchain supports the steps of the HLS chain as submodules which can be integrated to the system as the further work. The process flow we apply is shown in Figure 1. Input to the framework is an HLL, either ANSI-C or LRH(+) - the language of the RH(+)[1] framework.

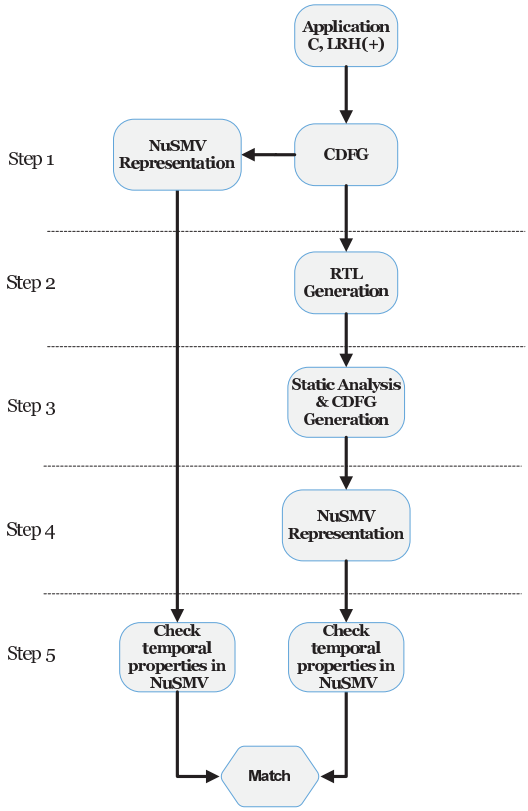

Fig. 1: Framework process flow

LRH(+) enables sequential and concurrent programming in the same environment and includes both traditional programming constructs and the constructs special to reconfigurable computation. LRH(+) introduces bit-length flexible data types, custom operations and explicit parallelism on the application source code. In Step 1, we generate Control Data Flow Graph (CDFG) of the input code by applying static code analysis.

Static code analysis is one of the methods of verification in which the analysis of code is performed without actually executing the code as opposed to dynamic analysis with execution of code. We apply static code analysis to both source code and to the generated RTL files. We generate NuSMV code both from our CDFG and RTL. The properties we apply are similar to the ones in compiler verification techniques [23]. Hence, the uniqueness of our method is to apply static code analysis both to source code and RTL code. In addition, we apply known compiler verification properties to verify our RTL. These properties our explained in Section V-C

The generation process is explained in Section V-B. Step 2 is our proposed RTL generation process. RTL generation lies on our hardware synthesis estimation model which is explained in Section III. The RTL generation is explained thoroughly in Section IV. Generated HDL files are analysed in Step 3 and we generate a new CDFG. In Step 4, we apply the same conversion methods in step one to generate a second NuSMV state machine code. Step 5 is the execution, comparison and logging stage. NuSMV codes are executed and outputs are compared with supplied temporal logic properties. This is explained in Section V-D.

II. Related Work

There are several estimation tools for FPGA based implementations. A method of area estimation model for Look-Up-Table (LUT) based FPGAs is proposed in [5]. This paper presents accurate area estimation results but does not present any delay estimation model. The method in [6] develops delay and area model at the Data flow graph level (DFG). Unfortunately, the number of operations that are supported are limited and estimation error is high for some applications. In [7], area, time, power models are given for Xilinx IP core which is integrated to FANTOM design automation tool. In [8], an estimation technique is proposed dealing with a MATLAB specification.

There exist commercial [9], [10], [11] and academic environments [12], [13] that allows rapid hardware development with limitations. Evidently, the compilation process of such languages is not completely different than regular software compilation processes. The main difference has been generally on the back-end of the compilers.

Systems that can be realized as finite state machines are formally verified by model checking [14]. The importance of formal verification is known to be able to show full functional correctness by proof-based methods. However, it is known to be very complex and generally requires expert knowledge. Code review is the human comprehension of the given finite state machine or the software. Although it can address design flaws at the higher level, lower level details can be missed very easily. Model checking of high level programs have existed for some time [15], [16]. There also exist off the shelf and research-level hardware model checkers at the netlist level [17], [18] and at CDFG level [19] with standard logic assertions.

Verification of high level RTL generation mainly differs based on the methodologies. Authors of [20] favor to divide the complex checking of algorithmic description into smaller problems which are intra cycle equivalence and valid register sharing property. The structural RTL definition and structural behavior might not be completely equal but by defining conflicting register sharing cases, the temporal model checking properties are written. Similarly, intra cycle equivalence checks that at any given time the behavioral structural register mappings should match all the input assignment operators that is performing. Our method inherently checks this because the extraction of variables from HLL allows us to bind each variable to an operator and our properties check if any bugs exist by wrong usage of variables. Therefore, our aim is to verify that the design uses as little memory units as possible. This is achieved by removing redundant variables which are stored either in registers or in memory. The work in [21] suggests a verification methodology which converts data-path and control-path specification to a proof script. The concept of critical states and paths are introduced. Critical path is a path that starts and ends from a critical state (critical state can be input output variables and conditionals etc.). The proof checking equates the given behavioral description to the RTL. Our method can capture the same behavior because it inherently traverses all the paths from input to output path. Therefore, extra pre-processing is neglected in our solution yielding relatively quicker verification.

III. Estimation Model of Arithmetic Components

Shrinking time to market, short product lifetimes increase the necessity of early behavior estimation of DSP algorithms during the development, prototyping cycle. An estimation model for our RH(+) HLS and RTL generation tool is proposed in this section. Firstly, datapath components are selected according to their delay and area behaviors. Then, behavior estimations of these components are measured with Xilinx XST tool by synthesizing them on FPGA with changing parameters of these components. We use Xilinx XST synthesis reports in our analyses. We apply Linear Regression on these measurements to extract model for delay, area and power metrics. The output of the RH(+) HLS tool is the data flow graph that represents the algorithm defined in high level language. These CDFGs are used for behavior estimation and for RTL generation purposes. The estimation methodology is handled in two steps; node estimation and graph estimation.

A. Selection of Arithmetic Components

The estimation is modeled for four basic types of operators; adder, subtractor, multiplier and divider. The subtypes of the adders/subtractors are inspected for design space exploration.

These subtypes are Carry Lookahead Adder (CLA), Carry Skip Adder (CSKA), Carry Select Adder (CSLA), and Ripple Carry Adder (RCA). It has been seen that unlike ASIC, RCAs are the fastest and smallest adder in FPGAs because FPGAs have a dedicated logic for implementing RCA. CLA, CSKA and CSLA are inspected in terms of area and delay behaviors. CLA and CSKA do not have any advantage on RCA for FPGAs neither in area nor in delay behavior. However, CSLA has a slight advantage on delay over RCA for the bit sizes wider than 128. There was not any advantage on area behavior. So, the RCA and CSLA are placed on the adder library for RTL generation and design space exploration purposes. The subtractor has no difference in architecture and performance so subtypes of subtractors are the same as adders. The LogiCore IP Multiplier 11.1 and Divider 3.0 from Xilinx IP Core library [3] are used as multiplier and divider components. All the aforementioned estimation, generation and verification processes are measured for the multiplier and divider cores from Xilinx library with hardware blocks enabled (instead of LUT implementations).

B. Parametric Area and Latency Estimation for Nodes

The arithmetic components are synthesized in Xilinx ISE 13.2 with different port sizes. The delay and area behavior results are gathered from the synthesis reports. Curve fitting is applied to these results by using linear regression analysis so as to embed these models into RH(+) framework and RTL generation software. Polynomial functions and piecewise polynomial functions are used for curve fitting. For Spartan 3 FPGA, we observe that RCA model is very regular in behavior. Hence, polynomial function in 1 fits well.

$$y = p_0 * x + p_1 \tag{1}$$

In equation (1), x corresponds to input bitsize, y corresponds to delay in nanoseconds or area in slices and p_0, p_1 corresponds to model parameters. However, the models of CSLA, multiplier and divider are not so regular. For that reason, piecewise polynomial functions are applied for these components. The following equation is an example for multiplier which is a piecewise polynomial function.

$$For \quad 0 < x < 32$$
$$f = p_0 + p_1 * x + p_2 * y + p_3 * x^2 + p_4 * x * y + p_5 * y^2 + p_6 * x^3 + p_7 * x^2 * y + p_8 * x * y^2 + p_9 * y^3; \tag{2}$$

$$For \quad 32 \leqslant x < 64$$
$$f = p_{10} + p_{11} * x + p_{12} y + p_{13} * x^2 + p_{14} * x * y + p_{15} * y^2 + p_{16} * x^3 + p_{17} * x^2 * y + p_{18} * x * y^2 + p_{19} * y^3; \tag{3}$$

In equations (2) and (3), x and y are the bit sizes of the inputs, f is the delay in nanoseconds, parameters p_0 to p_{17} are model parameters.

Listing 1: HBD File of RCA Adder

```
@synth
*variable = (x)
@hbd
*Property = Delay
*functionType = poly
*(1:0.0559)+(0:1.5316)
@endhbd
@hbd
*Property = Area
*functionType = poly
*(1:0.4996)+(0:0.055)
@endhbd
@hbd
*Property = Power
*functionType = poly
*(4:0.000000003288)+(3:-0.0000014918)+(2:0.00020272)
@endhbd
@endsynth
```

Every node in CDFG represents an arithmetic operator which has delay and area cost. The created model for our arithmetic operator is embedded in RTL generation software with hardware behavior description files. Hardware behavior description (hbd) file is a file with a format that defines the area, delay and power models of arithmetic units. In listing 1, hbd file of RCA adder is shown as an example. Hbd files start with @synth and @endsynth. The fields between every @hbd and @endhbd describe a model function. Property field describes whether this is delay, area or power model. FunctionType field describes the characteristic of the function which can be polynomial, piece-wise linear or piece-wise polynomial. The line after FunctionType describes the function in a specific format. The function in listing 1 corresponds to the equation 1. The value between "(" and ":" corresponds to power of the variable and the value between ":" and ")" corresponds to coefficient.

For example, for the function of delay property in listing 1, 0.0559 corresponds to p_0 and 1.5316 corresponds to p_1 of equation 1. Every arithmetic operator has a hbd file. During behavior estimation, the software reads and parses the hbd file and extracts equations. For every node in the CDFG the hbd file is read and estimation values are calculated.Hbd files are useful for embedding estimation model in the RTL generation software. Moreover, it gives extendibility to the design automation tool. Hbd file has a user-friendly format and the user can easily introduce a module to the HLS tool by specifying its area, delay and power consumption with an hbd file.

C. Area and Latency Estimation of the CDFG

With node estimation, latency and area of every vertex in the CDFG is estimated. However, for the estimation of the overall performance, a complete estimation of the CDFG is necessary. When our estimation model is compared with [6] and [7], Model in [6] make estimations by extracting general information such as how many components are used and what the average input length does the operators have. It simply calculates how much resource used in FPGA and estimates delay based on the resource usage. However, in our

methodology, we calculate the delay and area costs for every component in the datapath and calculate the area estimation by summing these values and calculate the delay estimation by finding critical path with graph processing. Estimation of our model is more accurate especially when the application's critical path is not proportional to graph size because Enzler's [6] model does not use critical path in calculations. The model in [7] is closer to our approach but it estimates only area not delay.

IV. RTL Generation

The Dataflow graphs that are generated by the RH(+) tool are synthesized to VHDL as Golden-RTL. Golden RTL is the datapath circuit without any multiplexer and resource sharing. Golden RTLs are used for verification of design units. In the graph, every vertex represents an arithmetic operation and edges represent the signals between the operations. Generation of Golden RTL program consist of two main functions.

A. HDL Generation

We generate necessary adder, subtractor, multiplier and divider VHDL files as components and connect them at the top level. During generation, firstly, every vertex in the graph is generated as a component. Every arithmetic operator in our library has a template file and parameter file. Template file is a template VHDL file of the operator which has fixed fields and editable fields. Parameter file defines the editable fields in template file and corresponding values of these fields. Using template and parameter file decreases the software run-time by not allowing the fixed fields to be generated over and over again, moreover it provides extendibility on the library. The user can integrate an operator to the RH(+) framework by putting template and parameter files in the library. The parameter files and template files are used for generating the arithmetic components by the RTL generator software. After every necessary component is generated, these components are connected at the top level. Signed or unsigned extension can be done during the port mapping of the signals and operators. The RTL is generated with 4 options that can be selected by the user.

- Neither inputs nor the outputs will be registered
- Only inputs are registered
- Only outputs are registered
- Both the inputs and outputs are registered

V. Formal Verification of Golden RTL

CDFG represents all the paths that might be traversed through a program during its execution. Each node in the CDFG represents operations or control structures. Each edge represents the data dependency between operations. CDFGs are concurrent structures. Since it lies in the middle of RTL level and high-level representation, it is mostly where all the optimizations, reductions and decisions about the system is made. Hence, it is important to capture as many details as possible from higher to lower levels. By using Computational Tree Logic (CTL) and Linear Time Logic (LTL) properties [14], bugs are detected right before converting the CDFG to RTL.

A. CDFG Structure

Vertex and Edge data structures are defined to represent the CDFG. Vertex data structure holds the necessary information of operations and variables. Variables represent storage elements. They can either be registers, internal or external memory based on the architecture. The important properties of the Vertex is given below:

- *Name*: Name of the Vertex
- *Index*: ID of the Vertex
- *IsOperator*: Flag for identifying if the vertex is an operator or a variable
- *OperatorType*: Types can be Adder, Subtractor, Divider and Multiplier
- *PrecedenceLevel*: Precedence level of the operators/variables in CDFG. It helps to identify dependencies between components.
- *Processed*: Flag that helps to identify if any operator/-variable is processed
- *Wordlength*: Wordlength of the operation/variable

Edge structure holds the dependencies between vertices. Apart from pointers to and from Vertices, only a single property is necessary:

- *Name* Name: Name of the Edge

B. Generation of NuSMV Representation

NuSMV [2] is a symbolic model checker developed as the reimplementation and extension of SMV [22]. NuSMV takes a text consisting of program describing a model and some specifications (temporal logic formulas). It outputs *"true"* if the specification holds or prints a trace showing a counterexample. The conversion of CDFG to NuSMV was done automatically by the RH(+) framework. The conversion steps are given below:

- Identify precedence levels of the CDFG in order to differentiate the input variables, output variables and the operators.
- Generate boolean and integer variables for marking and storing usage, declaration and wordlength information.
- Generate a state in NuSMV for each node in CDFG.
- Declare transitions for each states according to the dependencies in CDFG
- Generate properties to check for each variable or operation

The process starts with identifying precedence levels. Identifying precedence levels is important because it produces an initial scheduling for the given CDFG. Each operation needs inputs and those inputs can either be variables or outputs of other operations. In addition, it clears the input nodes and the output nodes of the CDFG. Two new nodes are inserted in the CDFG which are *Sink* and *Source*. After identifying the precedence levels, the process starts to create boolean

and integer variables. Those variables and their intentions are summarized below:

- *defined*: Operands used by operations are stored as variables. Before a variable is used, it must be defined. This boolean flag identifies in which state the variable is defined in CDFG.
- *used*: Both variables and operands are defined but not necessarily used. This flag determines in which state the variables and operations are used.
- *assigned*: After variables are defined, they must be initialized (assigned) in order to be used because their intrinsic values can be random. Hence, this flag is true whenever the variables are initialized or assigned and false otherwise.
- *wordlength*: Wordlength is used to correctly calculate the wordlengths of the operators, their inputs and output sizes as well as variable wordlengths. Each operation has different contribution to wordlengths. Overestimating wordlengths of operations increases area and power consumption. Underestimating wordlengths creates faulty results and wrong connections between components. So, wordlengths are essential for each operation and variable.

After we create the flags, first state to create is *Source*. All the states emerging from the *Source* state are the states that have precedence level equal to one. Based on the precedence levels and transitions in CDFG, we make next states assignments. Similar to *Source* state assignment, the final state is *Sink* state. States that have the highest precedence levels make transitions to the *Sink* state.
The procedure explained above from CDFG to NuSMV follows the routines below:

- GenerateSMV: This routine is separated into subroutines:
  - GenerateVars: This routine makes all the declarations of the variable and operation properties
  - GenerateAssignment: This routine makes all the initial assignments of the variables and the operators
  - GenerateState: This routine creates the states depending on the node information taken from the CDFG
  - GenerateVarState: This routine creates all the state transitions of the variables
  - GenerateCompState: This routine creates all the state transitions of the operations
- UpdateGraphSMV: LRH(+) code is parsed, necessary data properties are filled and precedence levels are calculated. For precedence level calculation, the algorithm traverses the CDFG and checks the successor predecessor relationship.

C. Temporal Logic Properties

The temporal properties are written either in CTL or in LTL. For this project, there are four properties that are checked. We combine atomic formulas with a set of connectives to form well formed formulas. We summarize the semantics of the connectives below:

- AG: For all the computational paths, the formula must hold globally
- AF: For all the computational paths, there will be some future state where the formula holds
- EF: There exists a computational path, where there will be some future state the formula holds
- EX: In some next state, the formula holds
- $E[s \quad U \quad t]$: There exists a computational path such that s is true until t is true

Based upon the semantics above, the properties are explained below:

- Unused Variables Property: This property checks that all variables that have been declared can be used eventually. The CTL formula is straightforward:
  $AG(defined \rightarrow EFused)$
- Assigned value is never used: When a value is assigned to a variable it should be used eventually. A value may not be used on all paths and it is not useful to check for this general case.
  $AG(assigned \rightarrow EX(E[!assigned \quad U \quad used]))$
- Deadcode Elimination: Dead code is code that can never be reached in a program. By adding source and sink nodes, we check that from sink every path eventually leads to the source.
  $AG(sink \rightarrow AF(source))$
- Wordlength Check: Given a CDFG and input variable wordlengths, depending on the operations on the CDFG, output variables have expected wordlengths (similar to worst case). When CDFG is converted to NuSMV representation, these expected wordlengths are assigned as initial values. Then depending on the state transitions in NuSMV, the wordlengths alter. The check of final wordlength with the initial(estimated) wordlength equates that the results match in wordlength, otherwise a counterexample path is printed.
  $AF(out\_wordlength = val)$ is checking the output variable's wordlength as an example. Wrong connections can alter the wordlength.

These properties are not unique to hardware design. For instance, similar properties are used for software verification in [23]. Although these properties are not unique to hardware design the usage such as our work helps identifying important faults in hardware. Uninitialized variables mean that they will occupy extra storage either in expensive register-file or an external memory which increase overall latency. Therefore by reducing unnecessary variables, power consumption, area and latency performances are improved. Dead code elimination finds connection faults and redundant hardware. Unnecessary hardware can also degrade performance and area. Therefore, all the properties have direct effect on area, power consumption and latency of the generated RTL.

D. Backward Process: Generation of CDFG from RTL

The generation of CDFG from RTL and generation of NuSMV from the new CDFG enables the designer to verify the

TABLE I: Delay and area estimation of the benchmarks

Application	Input Wordlength	Delay Estimation Error (%)	Area Estimation Error (%)	Chip Occupation (%)
DE	4-bit	0.23	9.12	0.37
	8-bit	0.40	4.74	1.36
	16-bit	0.34	2.34	5.13
	24-bit	0.27	1.05	11.22
	32-bit	0.02	0.58	19.70
IIR	4-bit	0.38	14.54	0.62
	8-bit	0.36	8.87	1.93
	12-bit	0.03	5.75	3.90
	16-bit	1.76	5.06	6.64
	24-bit	6.52	3.10	14.08
	32-bit	12.07	2.28	24.35
	36-bit	18.92	1.75	30.45
	44-bit	19.78	0.92	44.67
AR	4-bit	0.23	2.00	1.14
	8-bit	0.16	0.38	4.11
	12-bit	0.20	1.28	8.87
	16-bit	2.97	3.05	13.80
EW	4-bit	11.64	6.24	0.42
	8-bit	2.08	4.57	0.57
	12-bit	0.76	3.60	0.73
	16-bit	0.87	2.98	0.89
	24-bit	2.29	2.3	1.20
	32-bit	6.72	1.78	1.51
	36-bit	1.61	1.62	1.67
DCT	4-bit	31.54	10.14	0.25
	8-bit	25.26	6.12	0.42
	12-bit	27.97	4.39	0.59
	16-bit	21.77	3.44	0.76
	24-bit	10.85	2.40	1.09
	32-bit	0.43	1.86	1.43
	36-bit	1.47	1.69	1.60
	40-bit	1.32	1.52	1.77

properties on the generated RTL. Therefore, after this process, there will be two CDFGs generated: one from the HLL and the other from the generated RTL. Hence, same properties are expected to yield same results. We apply static analysis on the generated RTL. By parsing structural mappings, signal/wire connections from RTL and querying components from RH(+) component database we build the second CDFG. Then, this CDFG is converted to NuSMV representation and properties are checked. Finally, we observe the results of the first CDFG and the second CDFG.

VI. Experiments

We use five benchmarks namely Elliptic wave filter (EW), Auto Regression (AR), Discrete Cosine Transform (DCT), Infinite impulse response filter (IIR), Differential Equation (DE) to evaluate our framework. These algorithms are coded in LRH(+) and CDFGs are generated by RH(+) compiler. CDFGs are used as inputs to our estimation, RTL generator and verification tool. We consider different input wordlengths and synthesize our design on the Xilinx Spartan 3 FPGA. We compare our estimation of area and delay results with Xilinx ISE 13.2 synthesis, place and route reports. The design area is measured in FPGA slices and the delay is measured in nanoseconds. Table I shows the error of applications with different wordlengths between our estimations and Xilinx's estimation for delay and area. The average error in delay estimation is 6.57% and 3.76% in area estimation. Benchmark estimations are based on aggregating standalone behaviors of individual nodes (operators like addition, multiplier etc.) which means routing between nodes are not considered and estimated accordingly. However, internal routing of each node is estimated. We design platform specific technology file which estimates routing overhead. We present the CPU run time of our estimation software and Xilinx ISE for five different applications in Table II.

TABLE II: CPU Run Time Comparison of Xilinx Estimation and Our Estimation Tool

Application	Input Wordlength	Our Estimation Run Time (msec)	Our RTL Generation Run Time (sec)	Xilinx Run Time (sec)
DE	16-bit	18.72	235.56	174.62
IIR	16-bit	24.96	616.23	207.21
AR	12-bit	28.08	580.32	416.41
EW	16-bit	23.40	756.60	114.88
DCT	16-bit	20.28	822.12	107.87

Listing 2: LRH(+) source code

```
Function;Func1: T,index:output;
  Expression;exp3;
   .=(error,.-(output,actual_val));
   .=(output,.+(.+(.*(Kp,?error),
   .*(Ki,integral)),
   .*(Kd,derivative)));
  ExpressionEnd;
FunctionEnd;
```

Estimation Run Time (msec) corresponds to software CPU run time for processing graph, calculating the area and delay costs for every vertex in the graph using hbd files and then estimate the area and delay of the whole graph. RTL generation Time (msec) corresponds to software CPU run time for generating VHDL files of the graph. Xilinx Run Time (sec) corresponds to CPU run time of synthesis, mapping, placing and routing of the graph in Xilinx ISE.

Listing 3: Generated NuSMV code

```
MODULE main
VAR
Kp_used: boolean;
Kp_assigned: boolean;
Kp_defined: boolean;
Kp_wordlength: 0..8;
SE79_used: boolean;
SE79_wordlength: 0..9;
state : {sNode_output,sNode_Kp,sNode_Ki,
sNode_SE79, SE80, SE81, SE82}
ASSIGN
init(state) := sNode_Source;
init(Kp_used):= FALSE;
next(state) :=
case
state = sNode_Sink : {sNode_Sink};
next(Kp_defined) :=
case
state = sNode_Source : TRUE;
TRUE :Kp_defined;
esac;
```

As Xilinx makes estimations by passing through these steps, our estimation run time is much faster while the estimation error is acceptable. *Step 3* of Figure 1 represents our estimation as part of the framework process flow. Listing 2 shows a part of the the DE algorithm in LRH(+). The RH(+) compiler creates the corresponding CDFG which is shown in Figure 2. All the CDFG parsing and NuSMV code generation are fully automated. We use Microsoft Visual Studio 2010 for code generation and NuSMV version is 2.4. Listing 3 shows a part of the generated NuSMV code resulting from the CDFG (Figure 2). Each operation is represented as a unique state

TABLE III: Total verification time of our benchmarks in msec

Application	Node Size	8-bit WL	16-bit WL	32-bit WL
DE	11	792	1221	1584
IIR	26	1586	3259	5581
AR	28	1440	1728	6080
EW	28	1645	3508	6240
DCT	28	1838	3355	6376

in NuSMV and state transitions occurs with respect to the original flow in the CDFG. Flow is from source to sink and each variable is altered according to its definition that is explained in Section V-C. Hence, by correctly altering the variables, the necessary logic flow is captured. Table III shows the total verification time of our benchmarks. Total verification time refers to *Steps 1,3,4 and 5* of Figure 1.

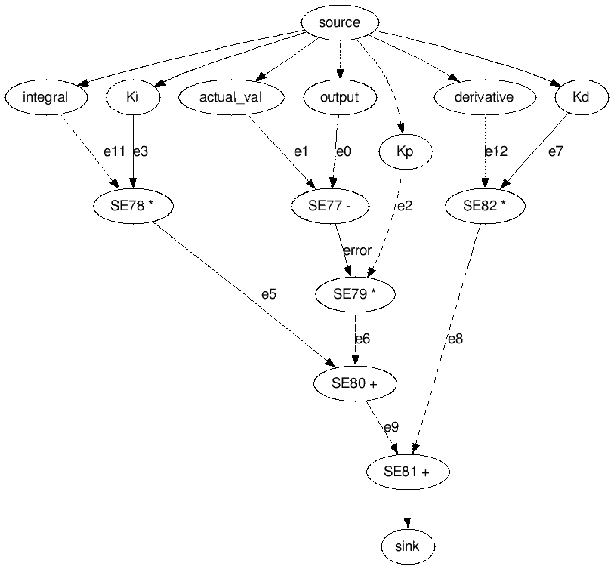

Fig. 2: Partial CDFG representation

Node size refers to the number of operations in the CDFG. All the benchmarks were verified with three different input variable wordlengths (8, 16 and 32 bits). The benchmarks have maximum of twenty eight operations (nodes). We observe that the node size and input wordlengths increase the total verification time. The platform used for tests are Intel Core i5-750 2.67 Ghz 4GB RAM running Windows 7 64-bit OS. NuSMV interactive window displays if given properties are satisfied otherwise shows a counterexample path from *Source* to *Sink*. Listing 4 displays a counterexample. The property *Wordlength check* is checked and the path displays which value combinations are conflicting with the expected result. By following the states in the counterexample output, one can trace the path that does not logically satisfy the properties which is to be verified.

The counterexample can draw an unsatisfying path from first state to the last state. We created this counterexample by checking an unexpected variable wordlength. The first line of Listing 4 checks if the wordlength of the variable *out* is 11.

Listing 4: NuSMV counterexample

```
-- specification AF out_wordlength = 11  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  a_used = FALSE
  a_init = FALSE
  a_wordlength = 8
  a_defined = FALSE
  b_used = FALSE
  b_init = FALSE
  b_wordlength = 8
  b_defined = TRUE
  c_used = FALSE
  c_init = FALSE
  c_wordlength = 8
  c_defined = TRUE
  d_used = FALSE
  d_init = FALSE
  d_wordlength = 8
  d_defined = TRUE
  out_used = FALSE
  out_init = FALSE
  out_wordlength = 11
  out_defined = FALSE
  sink_done = FALSE
  source_done = FALSE
  SE77_used = FALSE
  SE77_wordlength = 12
  SE78_used = FALSE
  SE78_wordlength = 12
  SE79_used = FALSE
  SE79_wordlength = 10
  state = sNode_Source
-> State: 1.2 <-
  a_defined = TRUE
  out_defined = TRUE
  source_done = TRUE
  state = sNode_a
-> State: 1.3 <-
  a_init = TRUE
  state = sNode_SE77
-> State: 1.4 <-
  a_used = TRUE
  b_used = TRUE
  SE77_used = TRUE
  SE77_wordlength = 9
  state = sNode_SE79
-> State: 1.5 <-
  SE79_wordlength = 21
  state = sNode_Sink
  out_wordlength = 10
```

However, the last line of the counterexample shows us that with given input variables, the wordlength of *out* becomes 10.

Listing 5: Automated DE RTL Part Example

```
signal d_5_e : std_logic_vector( 4 downto 0);
signal d_2_e : std_logic_vector( 15 downto 0);
signal SignedMultiplier5_d0_e : std_logic_vector( 16 downto 0);
begin
e18 :   d_5_e <= "0" & d_5 ;
e5 :    d_2_e <= "000000000000" & d_2 ;
e20 :   SignedMultiplier5_d0_e <= "00000" & SignedMultiplier5_d0 ;
SE77 : SignedMultiplier_333_100_i1_4_i2_4 port map
(d_0 => d_0 , d_1 => d_1 ,  clk => clk , out_d_0 => SignedMultiplier0_d0);
SE78 : SignedMultiplier_333_101_i1_4_i2_4 port map
(d_0 => d_2 , d_1 => d_3 ,  clk => clk , out_d_0 => SignedMultiplier1_d0);
SE79 : SignedMultiplier_333_102_i1_4_i2_4 port map
(d_0 => d_0 , d_1 => d_4 ,  clk => clk , out_d_0 => SignedMultiplier2_d0);
SE80 : SignedMultiplier_333_103_i1_4_i2_4 port map
(d_0 => d_2 , d_1 => d_3 ,  clk => clk , out_d_0 => SignedMultiplier3_d0);
SE82 : RCAAdder_212_104_i1_4_i2_4 port map
(d_0 => d_1 , d_1 => d_3 , out_d_0 => RCAAdder0_d0);
```

We verified all benchmarks with our properties and all four properties passed successfully. Listing 5 shows part of the automated RTL generation of the DE algorithm written in LRH(+) (Listing 2). We apply static code analysis to generate the CDFG from RTL and query component information from RH(+) component database in order to regenerate the node and edge information of the original CDFG. The process of generating CDFG from RTL is straightforward for combinational datapaths. For sequential datapaths, the registers are represented with states that might not conditionally jump to any other state so the arithmetic path from the source to sink is not affected.

Therefore, even though the initial CDFG and the one generated from the RTL are not isomorphic, they generate same answers to the properties that are to be verified.

VII. Conclusion

In this paper, we show the process of a data-path synthesis framework which produces formally verifiable Register-Transfer-Level (RTL) logic from high level languages such as ANSI-C and LRH(+). Estimation of delay, area and power of Xilinx Spartan 3 FPGA were realized in order to speed up the design phase. In estimation models, we use polynomial and piecewise polynomial functions. We benchmark five different applications for modelling, estimation and verification performance. The average delay error is found 6.57% and area error is found 3.76%. Our behavior estimation model works in the order of milliseconds whereas Xilinx's process flow generates the expected results in the order of seconds. On average, our estimation tool is 300 times faster than Xilinx with acceptable error margins. The created RTL is verified by four different temporal logic properties for checking redundant hardware creation and wordlength related mismatches. We observe that the verification time differs with node size and input wordlengths. It has been seen that, generation CPU run time is dependent both on the operator vertex and edge sizes of the graphs. As the vertex and edge number increases, generation run time increases. Moreover, the number of signal extension increases the run time because the signals are extended when the operator vertex input wordlengths mismatch the wordlength of the incoming edges. Such situation happens for example when an adder is generated with 8 bit input wordlength but one of the input has wordlength less than 8 bits.

Acknowledgment

This work is supported by Bogazici University Scientific Research Fund (BAP) Project No: 11A01P6/6346

References

[1] B. Kurumahmut, G. Kabukcu, R. Ghamari, and A. Yurdakul, "Design automation model for application-specific processors on reconfigurable fabric," in *Proc. Forum Specification & Design Languages FDL 2009*, 2009, pp. 1–6.

[2] Nusmv v2.5 tutorial. (2012,february). [Online]. Available: http://nusmv.fbk.eu/NuSMV/tutorial/index.html

[3] Xilinx ise design suite12.4.(2011,december). [Online]. Available: http://www.xilinx.com/products/design-tools/ise-design-suite/

[4] Altera quartus ii design suite.(2012,february). [Online]. Available: http://www.altera.com/

[5] D. Kulkarni, W. A. Najjar, R. Rinker, and F. J. Kurdahi, "Compile-time area estimation for lut-based fpgas," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, pp. 104–122, January 2006.

[6] R. Enzler, T. Jeger, D. Cottet, and G. Troster, "High-level area and performance estimation of hardware building blocks on fpgas," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, ser. Lecture Notes in Computer Science. Springer, 2000, vol. 1896, pp. 525–534.

[7] L. Deng, K. Sobti, and C. Chakrabarti, "Accurate models for estimating area and power of fpga implementations," in *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, 31 2008-april 4 2008, pp. 1417–1420.

[8] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate area and delay estimators for fpgas," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '02. IEEE Computer Society, 2002.

[9] Handel-c language reference manual.(2011,december). [Online]. Available: http://www.mentor.com/products/fpga/handel-c/upload/handelc-reference.pdf

[10] Impulsec codeveloper. (2012,february). [Online]. Available: http://www.impulseaccelerated.com/productsxilinx.htm

[11] Open systemc initiative. (2012,february). [Online]. Available: http://quickgraph.codeplex.com/

[12] S.-H. Chou, C.-N. Wen, Y.-L. Liu, and T.-F. Chen, "Veric: A semi-hardware description language to bridge the gap between esl design and rtl models," in *Proc. Quality Electronic Design Quality of Electronic Design ISQED 2009*, 2009, pp. 535–540.

[13] S. D. Sahasrabuddhe, S. Subramanian, K. P. Ghosh, K. Arya, and M. P. Desai, "A c-to-rtl flow as an energy efficient alternative to embedded processors in digital systems," in *Proc. 13th Euromicro Conf. Digital System Design: Architectures, Methods and Tools (DSD)*, 2010, pp. 147–154.

[14] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[15] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch, "Model checking software at compile time," in *Theoretical Aspects of Software Engineering, 2007. TASE '07. First Joint IEEE/IFIP Symposium on*, june 2007, pp. 45 –56.

[16] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds. Springer Berlin / Heidelberg, 2004, vol. 2988, pp. 168–176.

[17] K. McMillan, "Verification of infinite state systems by compositional model checking," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, L. Pierre and T. Kropf, Eds. Springer Berlin / Heidelberg, 1999, vol. 1703, pp. 705–705.

[18] L. Semeria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake, and B. Pangrle, "Rtl c-based methodology for designing and verifying a multi-threaded processor," in *Design Automation Conference, 2002. Proceedings. 39th*, 2002, pp. 123 – 128.

[19] M. Zhu, J. Bian, W. Wu, and H. Xue, "Property-classified hybrid verification based on cdfg," in *ASIC, 2003. Proceedings. 5th International Conference on*, vol. 1, oct. 2003, pp. 233 – 237 Vol.1.

[20] P. Ashar, S. Bhattacharya, A. Raghunathan, and A. Mukaiyama, "Verification of rtl generated from scheduled behavior in a high-level synthesis flow," in *Proc. Digest of Technical Papers. 1998 IEEE/ACM Int. Conf. Computer-Aided Design ICCAD 98*, 1998, pp. 517–524.

[21] N. Mansouri and R. Vemuri, "A methodology for automated verification of synthesized rtl designs and its integration with a high-level synthesis tool," in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and P. Windley, Eds. Springer Berlin, 1998, vol. 1522, pp. 537–537.

[22] K. L. McMillan, "Symbolic model checking: An approach to the state explosion problem," Ph.D. dissertation, Carnegie Mellon Department of Computer Science, 1992.

[23] M. Lussenburg, "Extending and evaluating the goanna static model checker," Master's thesis, ETH Zurich, 2006.