Accurate Area and Delay Estimators for FPGAs

Anshuman Nayak and Malay Haldar
AccelChip, Inc.
www.AccelChip.com

Alok Choudhary and Prith Banerjee
Northwestern University
Evanston, IL 60208.

Abstract

We present an area and delay estimator in the context of a compiler that takes in high level signal and image processing applications described in MATLAB and performs automatic design space exploration to synthesize hardware for a Field Programmable Gate Array (FPGA) which meets the user area and frequency specifications. We present an area estimator which is used to estimate the maximum number of Configurable Logic Blocks (CLBs) consumed by the hardware synthesized for the Xilinx XC4010 from the input MATLAB algorithm. We also present a delay estimator which finds out the delay in the logic elements in the critical path and the delay in the interconnects. The total number of CLBs predicted by us is within 16% of the actual CLB consumption and the synthesized frequency estimated by us is within an error of 13% of the actual frequency after synthesis through *Synplify* logic synthesis tools and after placement and routing through the *XACT* tools from Xilinx. Since the estimators proposed by us are fast and accurate enough, they can be used in a high level synthesis framework like ours to perform rapid design space exploration.

1 Introduction

Field-Programmable Gate Arrays (FPGAs) have become increasingly popular because recent trends indicate a faster growth of transistor density than even general purpose processors. This high logic density plus the field-programmability offers an inexpensive customized VLSI implementation of circuits with fast turnaround time, making FPGAs very lucrative as a design platform. The problem with designing for FPGAs is that they have rigid routing and logic resources. Hence, it is possible that designs do not fit inside FPGAs or do not meet the performance figures required by the designer, so that the hardware designer has to iterate over his designs to meet area and timing closure. The problem is aggravated since most hardware designers today use hardware description languages like VHDL/Verilog as a design entry point to implement designs on FPGAs. This involves directly dealing with the complexities of the FPGA hardware and debugging involves understanding the cycle-by-cycle behavior of millions of gates, which can be very tedious and time consuming. Since FPGAs are normally used for applications with tremendous time to market constraints, there is a need for more mature design tools. Although design tools exist which take the VHDL description of the hardware and generate bitstreams, a current challenge in this field is to design a High Level Synthesis (HLS) tool which would allow designers to enter designs at a much higher level of abstraction. These HLS tools would take the algorithmic description of the required hardware together with certain area and performance requirements of the designer, and perform an exploration of the design space to output the hardware which meets the designers specifications.

Many researchers have focused on the use of general purpose languages as a target for hardware synthesis. C/C++ is the most popular target language [1, 2, 3, 4, 5, 6]. Some other researchers have attempted to use Java as the target language too [7, 8, 9]. We propose MATLAB^{TM} to be a suitable design entry point because : **(1)** FPGA accelerators are very popular with the signal/image processing community and MATLAB is very popular with this community as it is easier and more intuitive to use than C/C++ **(2)** MATLAB has a rich set of libraries for signal/image processing functions which can be directly mapped to efficient IP cores, thus making MATLAB very conducive to design reuse **(3)** Large amounts of parallelism can be extracted from MATLAB programs with little or no dependency analysis as required by languages like C/C++ **(4)** Mathworks Inc., which has developed the MATLAB language has provided a feature rich simulation environment which can be used to simulate high level algorithms in a bit-true manner. Unlike the other tool flows, we have implemented an automated design space exploration pass which finds the effects of various compiler optimizations on the area and frequency of the synthesized hardware, so that the synthesized hardware meets the users specifications. Such a pass needs early estimators which can be used to narrow down the design space. The most critical estimators are those for predicting the area in terms of number of Configurable Logic Blocks (CLBs) used by the circuit and the delay in the critical path which effects the frequency of the synthesized hardware.

Vootukuru et. al. [10] have presented a method for calculating FPGA CLB resources for all possible functional components and for all possible bitwidths, and maintaining a database. Such an approach would result in a huge database and is hence not useful for use with a HLS framework. Some other researches [11] have proposed fast mapping heuristics. Our method has a single estimation function per functional component and is hence very useful as a fast area predictor. A lot of work has also been done in estimating the frequency from a high level description. Dutt et. al. [12] have proposed computing the area-delay attributes for all the component instantiations and storing in a database to be used for component selection and binding. Since this is very storage inefficient, they again propose [13] a constructive approach in which the component instantiations are generated on demand and the area-delay metrics calculated. The advantage of such a system is that unlike most HLS solutions which are technology-independent, and hence inaccurate, a solution tweaked to a particular technology can be estimated. Timmer et. al. [14] propose a technique in which they start with a worst case implementation of the design with maximum area cost and progressively try to improve upon the area cost of the design by different scheduling techniques. The output of each scheduling technique is a design implementation with a different area cost. The delay of each implementation is found by maintaining a database of RT components. This gives different design points on the area-delay curve. Nemani et. al. [15] propose a simple technology-independent model for predicting the delay of control logic, given the Boolean equations for the logic. The major disadvantages of all these techniques is that they are not fast estimators so that they cannot be used with a high level synthesis compiler. Jha and Dutt have also proposed a framework [16] in which they develop a set of estimation functions that generate area-delay metrics on-line. The advantage of such a technique is that the size of the database is now manageable, but the disadvantage is that they assume the interconnect delay to be zero. Since routing delay in the FPGAs is very prominent, our approach estimates the interconnection delay in addition to the delay in the datapath.

The contribution of the paper can be summarized as follows:

- We present a technique to estimate the number of CLBs consumed by an algorithm written in MATLAB.
- We present a technique to estimate the frequency of the synthesized hardware for an algorithm written in MATLAB.

This work presents an automated way of improving the hardware generated by the MATCH compiler [20]. The rest of the paper is organized as follows. Section 2 presents an overview of our compiler. Section 3 outlines our area estimation framework while section 4 presents our delay estimator. We present some experimental results in Section 5 and conclude in Section 6.

2 Overview of the MATCH Compiler

The MATCH compiler [24] takes in the description of an application in MATLAB and partitions it into software to be executed on general purpose and embedded processors and hardware to be mapped to FPGAs. The hardware generated are targeted for the Xilinx FPGAs on the $Wildchild^{TM}$ board from Annapolis Micro Systems. In this paper, we address the issues involved in generating an efficient hardware once the frontend of the compiler has partitioned the system into hardware and software. In particular, we focus on developing certain early estimators to be used within our framework to choose the proper optimizations so that the synthesized hardware meets the user specifications. Figure 1 shows an overview of our compiler. The input MATLAB code is parsed in to develop a MATLAB AST based on a grammar developed by us [20]. Since MATLAB is a dynamically typed language, the type and shape of the variables are unknown at compile time. Hence, a compiler phase infers the type of the variables and dimensions of the matrices and uses this information to scalarize the MATLAB AST. The AST is then levelized wherein complex expressions are broken down into simple expressions with at most three operands. A dependency analysis phase infers the control and data dependencies present in the AST. A precision and error analysis phase [21] infers the optimum number of bits required for representing the variables in the MATLAB AST and generates a resource optimized VHDL AST. A memory packing phase [21] packs more than one array element into a single memory location depending on the array precision and optimizes on the number of memory accesses. The parallelization phase [25] extracts fine grain parallelism within a single FPGA based on a prediction of the available FPGA resources. A coarse grain parallelizing phase finds out the optimal alignment and distribution of data and loop computations across multiple FPGAs. The IP core integration pass [27] automatically instantiates optimized Intellectual Property cores for the target architecture followed by the pipelining [22] and scheduling pass [26] to generate pipelined hardware. The area/delay estimation pass sits on top of most of the optimization passes. For example, the parallelization pass uses these estimators to find out the maximum factor by which the MATLAB loops can be unrolled as unrolling would lead to instantiation of more resources. The output VHDL code is then passed through commercial synthesis and place and route tools to generate a netlist and bit-stream for the FPGAs. In this paper, we describe in detail, the area and delay estimators which are used by the other optimization passes.

3 Area Estimation

One of the most important constraints required by a FPGA hardware designer is that a design fits inside the

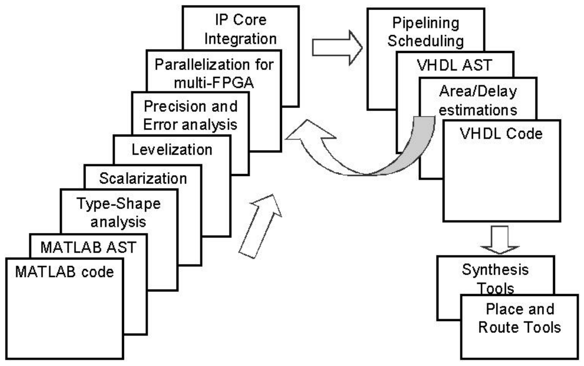


Figure 1. Overview of the Synthesis Framework

specified FPGA. This requires an estimate of the number of CLBs (Configurable Logic Blocks) required by the design. This is because a FPGA has fixed logic as well as fixed routing resources. A simple estimation of the FPGA CLB area from an Register Transfer Level (RTL) description of the hardware based upon the number of lines of VHDL code would be grossly inaccurate, because a single line of RTL code can create a wide complex multiplier while another line might require just a register to hold the result. Since, both the *Control Logic* and the *Datapath* of any design are implemented as lookup tables inside the CLBs, an accurate prediction of the number of CLBs required would need a count of the total number of hardware resources required by the design for the control logic and the datapath.

The total CLBs consumed by the datapath would depend on :

- the total number of different operators
- the number of operators of each type instantiated
- the bitwidth of each operator
- the total number of registers

The operator concurrency or the total number of operators is a measure of the total number of operations that must be simultaneously executed. This information is calculated after the scheduling step. *Paulin* et. al. have proposed a force directed scheduling algorithm where the probability that an operation is executed in a particular time step is calculated [17]. They propose that it is equally likely for an operation to be executed in any of the time slots between its as-soon-as-possible (ASAP) and its as-late-as-possible (ALAP) times. Such a scheduling gives us the probability that a particular resource will be used in a certain scheduling time step. We then use these probability figures to estimate the total number of operators in any execution time step. An initial binding gives us the information on the maximum number of operators of each type that need to be instantiated. Hence, we get the total number of different operators that need to be instantiated. Since the total number of CLBs consumed by each of these operators will depend on the input operand bitwidths, there is a need for a bitwidth analysis pass. We have developed a *Precision and Error Analysis* algorithm to accurately determine the minimum number of bits required to represent both integer and floating point variables [21]. This bitwidth information of the variables input to these operators is used to determine the size of these operators, in terms of the number of CLBs. Figure 2 shows the number of CLBs consumed by the different operators instantiated by the *Synplify* tool from *Synplicity* for the Xilinx *XC4010* FPGA. Hence, if we have an accurate estimation of the total number of different operators of each type and the bitwidth of the input variables, then we can determine the total number of CLBs consumed by these hardware components. Since all of these operators are instantiated as *IP Cores* from certain libraries, information similar to that in Figure 2 is available from the vendors of these libraries.

The CLBs also have certain Flip-Flops in addition to the lookup tables. Hence, all registers used in the designs use up CLB resources. Since the registers are used up by the VHDL variables declared, we need to determine the total number of variables that are simultaneously mapped to registers. In the VHDL representation of the hardware, all signals are mapped onto registers, while variables that cross clock boundaries or state boundaries in a state machine approach as ours are mapped onto registers. But, most high level synthesis tools might reuse registers across different variables. Hence, an estimate of the total number of variables that are simultaneously live would give us the total number of registers needed. The operator execution time defined by its ASAP and ALAP times defines the lifetime of an operation. The lifetime of a variable is approximated from the expectation of the execution times of its production and consumption nodes. Since the lifetime of the variables is known approximately, we apply the *left edge algorithm* [19] to determine the maximum number of variables that would be simultaneously live, and hence the number of registers required.

The total number of CLBs used by the *Control Logic* depends on:

- the total number of function generators used by the control
- the total number of registers used by the Finite State Machine (FSM)

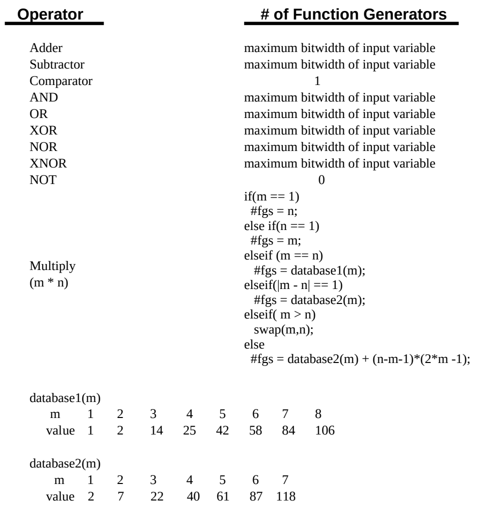
We have determined experimentally that the number of function generators used by each nested *case* statement is three while that for each nested *if-then-else* statement is four. Further, the number of registers used by the FSM depends on the number of states in the FSM which can be easily determined.

We determine the total number of CLBs consumed by both the control and the datapath using the formula:

Number of CLBs after Place and Route = maximum((# of Function Generators)/2,# of registers) * 1.15 (1)

where we divide the number of function generators by two since each CLB has 2 lookup tables. The factor of 1.15 was experimentally determined and is introduced to take into account all global optimizations done by the *XACT* place and route tool from Xilinx and CLBs used up for routing purposes.

Hence, we make a rough estimate of the total number of CLBs consumed by a design based on the resources consumed by the operators and by the registers. This information can then be used to estimate the maximum number of *adders* or other operators that can be instantiated and can hence be used with the parallelization pass to determine the maximum number of loop iterations that can be done in parallel within a single FPGA.



Operator	# of Function Generators
Adder	maximum bitwidth of input variable
Subtractor	maximum bitwidth of input variable
Comparator	1
AND	maximum bitwidth of input variable
OR	maximum bitwidth of input variable
XOR	maximum bitwidth of input variable
NOR	maximum bitwidth of input variable
XNOR	maximum bitwidth of input variable
NOT	0

Multiply (m * n)

```
if(m == 1)
  #fgs = n;
else if(n == 1)
  #fgs = m;
elseif (m == n)
   #fgs = database1(m);
elseif(|m - n| == 1)
   #fgs = database2(m);
elseif( m > n)
   swap(m,n);
else
  #fgs = database2(m) + (n-m-1)*(2*m -1);
```

database1(m)

m	1	2	3	4	5	6	7	8
value	1	2	14	25	42	58	84	106

database2(m)

m	1	2	3	4	5	6	7
value	2	7	22	40	61	87	118

Figure 2. Number of Function Generators consumed by Operators instantiated by the *Synplify* **tool from** *Synplicity* **for the Xilinx** *XC4010* **FPGA**

4 Delay Estimation

Most high-level synthesis tools perform rapid design exploration and output a design which meets the user specified attributes which normally include certain area and performance constraints. To achieve rapid design closure, such tools should not only have rapid area estimators as outlined in the previous section, but also, delay estimators to determine whether the synthesized design would meet the frequency constraints.

Since all functional RT level components are mapped to parameterized Intellectual Property (IP) cores, an estimate of delay features of these cores is essential. One easy way out would be to maintain a database of delay figures for every possible IP core for every possible parameter like *number of inputs* and *input bitwidth*. Since this approach is not practical due to the large number of combinations possible, we propose a modified approach wherein the storage required would be minimal. An important point to note is that most of these IP cores are parameterized and hence, critical path of these IP cores actually consists of a repeatable part which can be easily determined. Hence, the delay of any IP core can be formulated as an equation based on the delay of a repeatable part of the critical path and the number of times it is repeated which depends on the various parameters like input bitwidth and fanout. Figure 3 shows the result of an experiment in which we characterize the delay of a 2-input adder as a function of the precision of the input operands. We can see that two input buffers, a lookup table (LUTs) and a XOR gate are instantiated for all the adders. The varying part of the hardware is a set of repeatable multiplexors, which depends on the precision of the input operand. Hence, if we know the delay of the multiplexors, then we can formulate a delay equation for the 2-input adder based on the input bitwidths.

delay = 5.6 + 0.1 * (bitwidth -3 + floor(bits /4)) (2)
delay = 8.9 + 0.1 * (bitwidth -4 + floor((bits - 1) /4)) (3)
delay = 12.2 + 0.1 * (bitwidth -5 + floor((bits - 2) /4)) (4)

Equation 2, 3 and 4 give the delay as a function of the input operand bitwidth for a two input adder, a three input adder and a four input adder respectively, where *bitwidth* is the maximum bitwidth of the two input operands.

delay = 5.3 + 3.2 * (num_fanin - 2) + 0.1 * (bitwidth + floor(bitwidth - (num_fanin - 2))) (5)

Equation 5 combines the above three equations to have a delay equation for an adder as a function of the input operand bitwidths and the number of input fanins. Hence, the delay of any functional component would be :

delay = a + b * num_fanin + $\sum c_i$ * bit_$width_i$

where the summation is on the different input operands and a, b and c are constants to be experimentally determined. Such delay equations can be experimentally determined for various other basic operators. Complex functions can now be broken down into these basic operations to calculate the delay. These delay equations can now be used to calculate the delay in the critical path of the hardware. Our compiler generates a hardware represented as a state machine. In our approach, we assume a state boundary to be a clock boundary so that all computations within a state are performed concurrently. Hence, the computation which takes the maximum time across all states would determine the critical path of the circuit. Since we know the hardware required for each computation which is implemented as an IP core, the time taken for every computation can be easily obtained from the delay equations. Hence, we can accurately determine the delay in the critical path of the circuit based only on logic components.

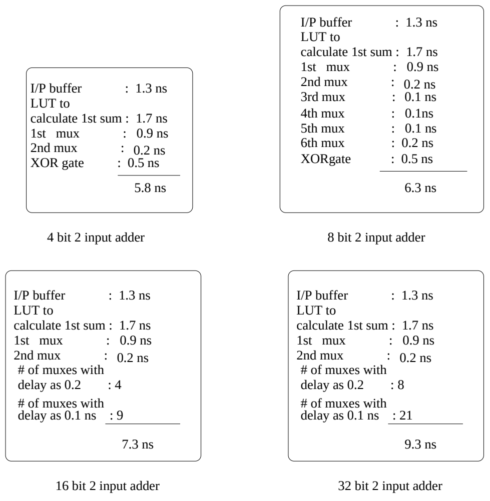


Figure 3. The delay of a 2-input adder is dependent on the number of operand bits

Even though we can estimate the critical path delay in the datapath based on the delay equations, this estimate of the critical path delay would be inaccurate as we neglect the delay in the interconnections. Hence, we propose a technique to find out the lower and the upper bound interconnection delay. This assumes that the placement tool provides a good partitioning of the netlist so that closely connected components are placed together. Since the interconnection delay is directly dependent on the length of the interconnection, a good delay estimator needs to accurately estimate the average interconnection length of a routed circuit. The average interconnection length can be determined by taking a useful result from [18]. There is an empirical relationship known as Rent's rule which is used for predicting the number of external connections from a given number of components in well-partitioned computer logic. If we assume that a good FPGA placement tool provides a good partitioning of the nodes, the average number of external connections will follow Rent's rule. *Feuer* et. al . [18] have derived a formula for the average wirelength distribution in terms of the Rent parameter for random computer logic to be:

$$L = \sqrt{2}\frac{(2-\alpha)(5-\alpha)}{(3-\alpha)(4-\alpha)}\frac{C^{p-0.5}}{1+C^{p-1}} \qquad (6)$$

$$\alpha = 2(1 - p) \qquad (7)$$

where C is the number of CLBs and p is the Rent parameter. The number of CLBs can be accurately determined from the previous section for area estimation and the Rent's parameter is experimentally determined to be 0.72.

Equation 6 and 7 can be used to calculate the average interconnection length of a design while the length of a wire segment between *Programmable Interconnect Points* (PIPs), which mark the beginning of new wire segments in FPGAs, is provided by the FPGA manufacturer. Hence, we can use this to find out the maximum number of PIPs that are used by a two-point connection. Since the routing delay would be the delay of the wire segments between PIPs (which is provided by the FPGA manufacturer) plus the delay inside the PIPs (which is also provided by the manufacturer), we can get an upper bound of the interconnection delay. Further, since most commercial FPGAs like the XC4010 have double lines, we can assume all connections to be routed using these double lines so that the number of PIPs and wire segments would be halved. Since we can also get the delay in a double line from the FPGA manufacturer, we can now get a lower bound for the interconnection delay. The combination of the logic and the interconnection delay would hence give us a lower bound and an upper bound on the synthesized circuit frequency.

5 Experimental Results

We carried out some experimental results to validate our strategy for the estimation of the number of CLBs consumed. We took some image processing benchmarks written in MATLAB, used our compiler to generate the corresponding VHDL and then used *Synplify* from Synplicity to generate the netlist. We input this netlist to the *XACT* place and route tools from Xilinx to get the actual number of CLBs consumed. We then used our area estimator to estimate the number of CLBs consumed. Table 1 shows the estimated CLB consumption and the actual number of CLBs consumed for certain benchmark circuits. We get a worst case error of 16%. This is in fact not so inaccurate because we are dealing with two levels of commercial tools. There is a definite uncertainity on how the logic synthesis tools like *Synplify* share resources across clock cycles, which will affect the total number of resources instantiated. Further, the placement tools might perform

Table 1. Experimental Results showing the percentage error in area estimation

Benchmarks	Estimated CLBs	Actual CLBs	% Error in Estimation
Avg. Filter	120	135	11.1
Homogeneous	42	48	12.5
Sobel	228	271	15.8
Image Thresh.	52	60	13.3
Motion Est.	478	502	4.7
Matrix Mult.	165	160	3
Vector Sum	53	62	14.5

Table 2. Experimental Results showing the accuracy of our area estimator in predicting the maximum loop unrolling factor, * : Results extracted by simulation as design did not fit on the Xilinx 4010

	Single FPGAs		multiple FPGAs			multiple FPGAS plus loop unrolling		
Benchmarks	CLBs	Time	CLBs	Time	Speedup	CLBs	Time	Speedup
Sobel	496	0.410	696	0.06*	6.8	696	0.06*	6.8
Image Thresholding	73	0.28	372	0.04	7.0	395	0.01	28
Homogeneous	93	0.32	378	0.042	7.5	398	0.02	16
Matrix Multiplication	133	12.61	375	2.06	6.1	375	2.06	6.1
Closure	164	12.71	425	2.18*	5.83	425	2.18*	5.83

some global optimizations during technology mapping. Also, the routing tool might use some of the CLBs as feed-through. Equation (1) does account for some of these through an experimentally determined factor. To prove that even this much inaccuracy is sufficient to use this estimator with some of the optimization passes, we use this estimator along with the parallelization pass to find out the maximum unroll factor for a MATLAB loop. We again took some image processing benchmarks and used our compiler to synthesize hardware for the multi-FPGA *WildChild* board. The *WildChild* board has 8 FPGAs so that partitioning loop computations across the FPGAs leads to reduction in the execution time. The third column in Table 2 shows a speedup of around 6-7 on 8 FPGAs [25], where speedup is defined as the execution time on a single FPGA to the execution time on 8 FPGAs. Since most image processing applications have parallel nested loops, a much higher speedup can be achieved by unrolling the loops within a FPGA. Hence, we hand unroll the innermost *for* loop in the benchmarks progressively, until the design would not fit inside the Xilinx 4010 FPGA. Thus, we got the maximum unroll factor for a particular benchmark. Then, we used our estimation strategy to verify that we could predict the maximum unroll factor which is dependent on the number of CLBs available. The last column of Table 2 shows the speedup attained by extracting parallelism within a single FPGA in addition to partitioning loop computations across the multiple FPGAs. For the *Image Thresholding* benchmark, we get a speedup of around 28 on eight FPGAs. The computation inside the *Image Thresholding* code consists of a *if-then-else* statement inside a doubly nested *for* loop. Our parallelization phase predicts a requirement of four CLBs for the *if-then-else* statement and a single CLB for the comparison. Hence, unrolling each loop iteration would result in the usage of five extra CLBs. Using Equation 1, we see that the maximum unroll factor can be predicted to be :

$$(5 * \text{Unroll\_Factor}) * 1.15 + 372 \leq 400$$

where, we require 5 extra CLBs for unrolling the loop once, 1.15 is the factor used to account for the extra CLBs required after the place and route stage, 372 is the number of CLBs already used up by the design and 400 is the maximum number of CLBs in a Xilinx 4010 FPGA. The maximum unroll factor can hence be predicted to be 4. Since the unrolled loop iterations would be done in parallel with the instantiation of extra hardware, the total speedup of the final design would be 4 times the speedup achieved by a multi-FPGA partitioning. It can be seen from Table 2 that the actual speedup attained after loop unrolling is 28 for a design that fits inside the FPGA and uses almost all the available CLBs, which proves that our prediction based mechanism is accurate. For most of the other designs, we could not extract parallelism within the FPGA as almost all of the CLBs were used up by the design. This experiment validates the need of such an estimation technique to generate efficient hardware and also that our area estimator is accurate enough to

Table 3. Experimental Results showing the Routing Delay Estimation. Benchmarks which end with a numeric are different hardware implementations of the same benchmark

Benchmarks	CLBs	Logic Delay (in ns)	Estimated Routing Delay(d) (in ns)	Estimated Critical Path Delay(p) (in ns)	Actual Critical Path Delay (in ns)	% Error in Estimation
Sobel	194	33.9	$2.46 \leq d \leq 9.26$	$36.36 \leq p \leq 43.16$	42.64	1.2
VectorSum1	99	26.1	$1.66 \leq d \leq 7.32$	$27.76 \leq p \leq 33.42$	32.75	2.05
VectorSum2	174	29.1	$2.32 \leq d \leq 8.93$	$31.42 \leq p \leq 38.03$	37.3	1.95
VectorSum3	168	34.5	$2.29 \leq d \leq 8.89$	$36.79 \leq p \leq 43.34$	40.03	8.26
MotionEst.	147	40.3	$2.12 \leq d \leq 8.44$	$42.42 \leq p \leq 48.74$	48.08	1.37
ImageThresh1	227	42.9	$2.68 \leq d \leq 9.79$	$45.58 \leq p \leq 52.69$	48.3	9.09
ImageThresh2	199	34.4	$2.50 \leq d \leq 9.38$	$36.9 \leq p \leq 43.78$	42.05	4.11
Filter	134	38.7	$1.99 \leq d \leq 8.16$	$40.69 \leq p \leq 46.86$	41.372	13.3

be used with some of the optimization passes.

To validate our routing estimations, we performed an experiment on five benchmark circuits. First, we used our compiler to generate the VHDL representation of the hardware. For some of the benchmarks like the *Vector Sum*, we used various optimizations to generate different hardware implementations with the same functionality. Then we used commercial synthesis tools from *Synplicity* and place and route tools from *Xilinx* to extract the final synthesized frequency. Since our compiler outputs a state machine representation of the hardware, we traverse the various states to find out the total CLB utilization based on the estimates outlined in the earlier section. We then use our estimation techniques to predict the delay in the datapath. Again, since the delay equations were derived after several runs of the *Synplicity* synthesis tool, this matches the delay from the *Synplicity* tool exactly. We then used Equation 6 and 7 to estimate the average interconnection length for the routed benchmark. The *Xilinx 4010* databook provides information on the delay in the various routing channels. The delay of a single line in the *Xilinx 4010* is 0.3 nanoseconds, of a double line is 0.18 nanoseconds while that inside a programmable switch matrix is 0.4 nanoseconds. We use this information along with the average interconnection length to calculate the lower bound and the upper bound of the routing delay. Table 3 shows the results of our routing estimation on various benchmarks. It can be seen that for all the benchmarks, the final critical path delay (and hence the frequency) is within the lower and upper bound delay estimated by us. Further, the last column shows a worst case error in estimation to be around 13 %. Since the estimation technique presented is fast and fairly accurate, this can be used with a high level synthesis tool like our compiler for rapid design space exploration. The main advantage will be in pruning off designs, which will never meet the user provided area and frequency constraints, during exploration of hardware implementations.

6 Conclusion

We have presented an estimator for finding out the area of a design in terms of the number of CLBs consumed by the design in a FPGA, and an estimator for finding out the frequency of a design after place and route from a description of the design as an algorithm in MATLAB. We have shown that these estimations can not only be made very fast, but the area estimations are within 16% of the actual number of CLBs consumed, while the frequency estimations are within 13% of the actual synthesized frequency. Hence, our estimators are fast and accurate enough to be used with a high-level synthesis compiler like our compiler for design space explorations.

References

[1] I. Page *Constructing hardware-software systems from a single description*, Journal of VLSI Signal Processing, pp. 87-107, 1996

[2] J. Babb, M. Rinard, C.A. Moritz, W. Lee, M. Frank, R. Barua, S. Amarasinghe *Parallelizing Applications into Silicon*, FCCM 1999

[3] D. Galloway *The Transmogrifier C Hardware Description Language and Compiler for FPGAs*, FCCM'95

[4] M. Gokhale, J. Stone, J. Arnold and M. Kalinowski, *Stream-Oriented FPGA Computing in the Streams-C High Level Language*, Proc. Field-Programmable Custom Computing Machines, April 2000.

[5] G. Doncev, M. Leeser and S. Tarafdar, *High-Level Synthesis for Designing Cu stom Computing Hardware*, Proc. Field-Programmable Custom Computing Machines, April 1998.

[6] A. Duncan, D. Hendry, and P. Gray, *An Overview of the COBRA-ABS High Level Synthesis System for Multi-FPGA Systems*, Proc. Field-Programmable Custom Computing Machines, April 1998.

[7] J.M.P.Cardoso and H.C.Neto *Towards an Automatic Path from Java(tm) Bytecodes to Hardware Through High-Level Synthesis*, Proceedings of the 5th IEEE International Conference on Electronics, Circuits and Systems (ICECS-98), Lisbon, Portugal, September 7-10, 1998, pp. 85-88

[8] B. L. Hutchings and B. E. Nelson,*Using General-Purpose Programming Languages for FPGA Design*, Proc. 37th Design Automation Conference, June 2000.

[9] R. Helaihel and K. Olukotun, *Java as a Specification Language for Hardware-Software Systems*, Proc. International Conference on Computer-Aided Design, pp. 690-697. November 1997.

[10] M. Vootukuru, R. Vemuri and N. Kumar, *Partitioning of Register Level Designs for Multi-FPGA Synthesis*

[11] M. Xu and F. J. Kurdahi, *Area and Timing Estimation for Lookup Table Based FPGAs*, Technical Report # 9530, UCI, Aug.1995.

[12] N.D. Dutt *GENUS: A Generic Component Library for High Level Synthesis*, University of California, Irvine Technical Report # 88-92, 1988

[13] N.D. Dutt and and J.R. Kipps *Bridging High-Level Synthesis to RTL Technology Libraries*, Proc. 28th Design Automation Conference, June 1991

[14] A. H . Timmer, M. J. M. Heijligers and J. A. G. Jess *Fast System-Level Area-Delay Curve Prediction*, Proceedings of the APCHDLSA, pp. 198-207, Brisbane, Australia, Dec. 6-8, 1993

[15] M. Nemani and F. N. Najm *Delay Estimation of VLSI Circuits from a High-Level View ew*, Proc. Design Automation Conference, 1998, pp. 591-594

[16] P.K. Jha and N.D. Dutt *A Fast Area-Delay Estimation Technique for RTL Component Generators*, University of California, Irvine Technical Report # 92-33, April 10, 1992

[17] P. Paulin and J. Knight, *Force-Directed Scheduling for the Behavioral Synthesis of ASICs*, IEEE Transactions on Computer Aided Design, Vol 8, No. 6, pp. 661 - 669, 1989

[18] M. Feuer *Connectivity of Random Logic*, IEEE Trans Computers, vol. C-31, no. 1, pp. 29-33, Jan. 1982

[19] A. Hashimoto and J. Stevens, *Wire routing by optimizing channel assignment within large apertures*, Proc. of the 8th Design Automation Workshop, pp. 155-163, 1971

[20] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak and S. Periyacheri, *A MATLAB Compiler for Distributed, Heterogeneous, reconfigurable Computing Systems*, Proc. IEEE Symposium on FPGA as Custom Computing Machines, FCCM 2000

[21] A. Nayak, M. Haldar, A. Choudhury and P. Banerjee, *Precision And Error Analysis Of MATLAB Applications During Automated Hardware Synthesis for FPGAs*, Proc. Design Automation and Test Conference in Europe, DATE 2001, pp. 722-728

[22] M. Haldar, A. Nayak, A. Choudhury and P. Banerjee, *Automated Synthesis of Pipelined Designs on FPGAs for Signal and Image Processing Applications described in MATLAB*, Proc. Asia South Pacific Design Automation Conference, 2001

[23] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary and P. Banerjee A. Choudhary, *FPGA Hardware Synthesis from MATLAB*, The 14th International Conference on VLSI Design, India, Jan 2001

[24] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee *A System for Synthesizing Optimized FPGA Hardware from MATLAB*, Proc. International Conference on Computer Aided Design, Nov. 2001

[25] A. Nayak, M. Haldar, A. Choudhury and P. Banerjee, *Parallelization of MATLAB Applications for a Multi-FPGA System*, Proc. Int. Symp. on FPGA Custom Computing Machines, FCCM, April 2001

[26] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, *Scheduling Algorithms for Automated Synthesis of Pipelined Designs on FPGAs for Applications described in MATLAB*, Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems, CASES'2000, November 2000.

[27] M. Haldar, A. Nayak, A. Choudhary and P. Banerjee, *FPGA Hardware Synthesis From MATLAB Utilizing Optimized IP Cores* Proc. Intl. Symposium on Field-Programmable Gate Arrays February 2001