Timing Estimation for Behavioral Descriptions

Doron Mintz and Carlos Dangelo
LSI Logic Corporation

Abstract

Behavioral, or high-level synthesis (HLS) generally produces an RTL code which in turn is synthesized into a netlist. The RTL code that is generated by the HLS program needs to meet user constraints such as clock cycle, available function units, area, etc. This paper shows that most synthesis programs will not meet the user timing constraint in many cases. As a result, a generated design might be functionally incorrect. We present an algorithm for estimating the minimum clock cycle for a synthesized design. The algorithm considers false paths, interconnect, wire and control unit delays to derive the minimal clock cycle time. A method that uses the algorithm to synthesize behavioral descriptions that meet the user timing constraints is also given.

Most HLS system perform scheduling and binding of the behavioral code at separate stages of the synthesis. In the scheduling step, function units are assigned to clock cycle time slots where only the intrinsic delay on the function units is taken into consideration. Only at the binding stage, the operators that represent the behavioral code are bound to actual function units. In this stage, mutually exclusive operations can be assigned to the same function unit (sharing). The binding algorithm tries, among others, to minimize the number of different signals (or variables) that are input to the same function unit and thus save on multiplexing costs (delay and area). However, most designs will end up multiplexing the inputs to some of the function units. This additional delay may result in a larger than the clock cycle time needed for a scheduled register to register path.

Even systems that do perform both scheduling and binding at the same time usually fail to consider the delays that multiplexers, registers, wires, and the control logic add to the datapath elements intrinsic delays[10]. This overhead delay is found to be very significant especially in large control oriented designs where many of the function units are shared. At times, both the delay and area overhead attributed to multiplexers at the inputs of shared function units and registers is as large as the function unit's or register's delay and area.

For designs which the timing, for example, is very critical, a post processing procedure can traverse the bound design and reduce delays by splitting multiplexers and adding functional units. This can cause area overhead and sub-optimal solutions. In order to find out if the design meets or exceeds the timing constraints all paths need to be checked while ignoring possible false paths. This could be very time consuming for large designs.

Next we briefly talk about related research and in Section 2 we present an average case linear time algorithm to estimate the minimal clock cycle for a design. Section 3 demonstrates how this algorithm is used to arrive at a timing correct design and Section 4 shows experiment results. Section 5 presents concluding remarks.

1 Related Work

Operator delays are the minimal information needed for a datapath scheduling algorithm. Early scheduling algorithms assigned unit delays to each one of the operators regardless of their actual delay[6]. More mature algorithms allow for variable operator delays as well as chaining, pipelining, and multicycle operations [7]. Still, most algorithms do not consider the effect of the interconnect on the actual timing. Chippe[1], Adam[4] and other synthesis systems that do use estimates of timing (and area) generally do not take interconnect delay into consideration. Chippe[1] does consider bus delays but for the purpose of estimating the total time i.e. the critical path time. In [5] the "best" clock cycle is computed as the largest function unit delay or the length of the critical path divided by the number of the control steps. Again, no consideration of interconnect delays.

Recently more researchers recognize the fact that interconnect delays become more and more important. In [2] the fact that resource sharing introduces multiplexers and thus increases the delay is discussed. ISIS[2] tries to optimize area and timing by changing the allocation and the sharing. It uses a low level timing analyzer to compute the timing. This timing analyzer does not consider the effect of false paths in the design. It also seems that the optimization is done on a state by state basis.

Hsieh *et al.*[3] present TinkerTool, a behavioral synthesis tool that derives a schedule that meets the user clock cycle constraint. TinkerTool exhaustively searches all the paths in the scheduled and bound control and dataflow graph (CDFG) to find the critical path. The maximum interconnect delays for every operator input is added to it's intrinsic delay and the CDFG is scheduled again. The control unit time seems to be added once to the control step time.

TinkerTool's model of the delays is very simplistic. A critical path may run through a non multiplexed operator input yet the tool adds the maximal input delay to the intrinsic delay. This will result in longer and wrong critical path timing. Moreover, the iterative process of changing the delays does not promise convergence at all.

This paper presents a better model for the path delays that incorporates the interconnect delays and control unit delays. We also present an average time linear algorithm to compute the critical path delay (unlike the exhaustive search used by TinkerTool). Finally we show a method for deriving the desired schedule that *will* converge.

2 Estimating the timing of a scheduled description

A behavioral code is usually synthesized into a synchronous design. At the end of the clock cycle, values are stored in the registers and these values are used in the beginning of the next. The values need to be correct and ready before they are latched in. Therefore, the clock cycle time should be larger than any register to register path in the design. As mentioned before, if delay overhead is not considered while scheduling, the given clock cycle might not be enough for the design to be functionally correct.

The problem is to find the minimal clock cycle time such that for each register to register path in the design there is enough time for the values to propagate from the source to the end register.

First we consider the netlist that is generated by the RTL synthesis tool. It is possible that some paths in the netlist are not relevant to the functionality of the design. Therefore, a regular delay predictor cannot predict the actual clock cycle from the netlist. Consider for example the code in Figure 1. Given only one multiplier, the code needs to be scheduled in two steps and the multiplier will be shared between the two multiplications. The corresponding netlist is seen in Figure 2. Since the multiplier is chained to the

```
y = c1+c2;
z = y * c2;
w = z * c2;
x = w - c1;
```

Figure 1: HDL code.

adder in the first clock cycle and to the subtracter in the second, a register to register path {c1,+,*,-,x} exists in the netlist but it will not be relevant for the functionality of the design. Therefore, state information is also necessary to identify the real paths.

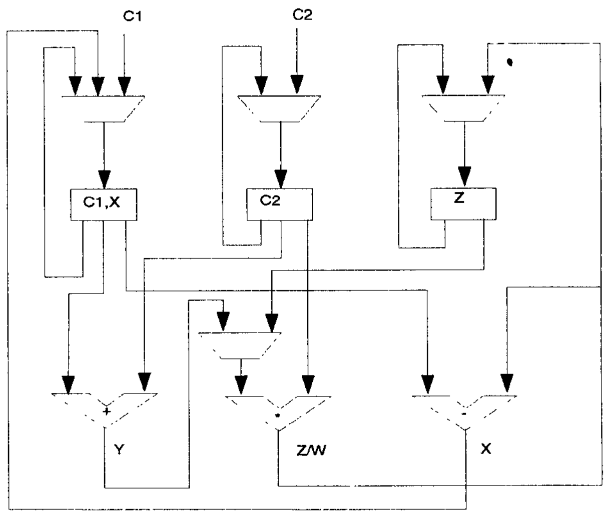

Figure 2: RTL netlist for the HDL code.

The solution then is to extract only those paths that will actually be executed. If this is to be done at the netlist level, the estimator needs to consider the control signals to the multiplexers. Several scans of the netlist will be necessary. Another possibility is to sensitize the paths [8].

We will show that by constructing a new graph from the control and data-flow graph (CDFG) the delay can be estimated in one pass.

We construct a weighted graph in the following way:

1. For each operator in the CDFG creat a node in the graph. Attribute each node with the intrinsic delay of the unit it is bound to.

2. Add arcs between nodes that correspond to chained operators. Assign to the arcs weights

that correspond to the delay on the input multiplexers on the function units which the destination operator is bound to.

3. Add zero weighted arcs between nodes that correspond to operators that conditionally depend on each other (read-write, write-read, and write-write combinations).
4. Creat two dummy nodes marked as *source* and *destination*. Assign zero weights to both.
5. Each data arc that crosses a clock cycle line (in the CDFG) represents a store operation in the design. For each such arc creat a node in the graph. Connect that node to the source of the arc. Attribute this node with the delay associated with the corresponding register. Attribute the arc with the delay of the multiplexer that inputs to that register.
6. Direct the output of this node to the *destination* node. Assign a zero weight to that arc.
7. Direct a zero weight arc from the *source* node to the destination of the original arc in the CDFG.
8. For each arc that flows from a port or a constant in the CDFG, connect the *source* node to the arcs destination (zero weight).

We have constructed a weighted graph that contains the original operators with the addition of dummy operators in place of arcs that cross the clock cycle lines.

This graph represents a simplistic model of the actual flow of the design. One factor that was not taken into account was the need for the control signals to function units and multiplexers to be stable before the computation. This might add a delay on paths and make non critical paths into critical ones.

One option is to compute the control signals at the end of the execution cycle and therefore the delay on the control unit can be added once to the maximal path delay. Under this model, all control values are ready at the next clock cycle and thus the graph that we constructed is a model of that design. This, however, is a restrictive model since it enforces that only one control related evaluation be done at each clock cycle. In a control oriented design (one that contains nested Ifs and Loops) this means that many clock cycles will be needed to evaluate the conditions even though there might be enough resources to do it in much less. Moreover, if the control signals to multiplexers, for example, are not needed to begin a cycle computation, the control can execute in parallel with the execution of the datapath. Our algorithm takes care of the control related issues too.

In our HLS system, the evaluation of the conditions is a part of the datapath. Conditional arcs connect the evaluation operators and the operators that are executed depending on the condition. Also, results of condition evaluation are stored in registers if they are needed in clock cycles other than the one that they are computed in. The database contains information about the membership of operators in the control related hierarchy of the original code. We therefore add the following steps to the graph construction.

9. For each operator: if the condition for it's execution is computed in it's clock cycle then add an arc between the condition evaluation and the operator nodes in the graph, otherwise add an arc from the *source* to the corresponding node.

   Let *cu* be a global variable that stores the maximal delay through the control unit. Assign a reference to this variable as the weight on the arcs.
10. For each operator input that is multiplexed, add a similar arc (from the condition evaluation or *source*). The arc weight is computed as a reference to *cu* with the addition of the multiplexer delay.

All *source* to *destination* paths in this graph are actual paths in the design execution. The problem now is to find the maximal such path.

One obvious solution is to generate all the *source* to *destination* paths and sort based on their time. Obviously this could be non linear in time and space complexity. A linear complexity algorithm would be preferable.

It seems that the solution is a regular depth first search traversal. Each node needs to be visited only once and after all the output arcs are considered the maximal delay to the *destination* is recorded. Unfortunately, this will not result in the correct estimate. Since some paths span more than one clock cycle (multicycle operations), the actual clock cycle is the path time scaled by the number of steps. Given that, we try to perform the same traversal but recording the scaled estimate. For this scheme to work, it is necessary that the following will hold:

> *The maximum scaled path from one node to "destination" can be computed from the maximum path times of its children.*

This, however is not true. Consider the graph in Figure 3. Assume that the delays are 0.5 for op1, 2 for op2, 1.5 for op3 and 5.2 for op4. The maximum length path from (*source* and) op2 to *destination* is {op2, op4} which is 3.6 (scaled for two cycles). However, if the previous assumption holds then the maximal path at op1 will be {op1, op2, op4} which is 3.85 (scaled), while the path {op1, op2, op3} is 4.0 and is the requested value. Therefore, it is not possible to compute only one maximal value at each node. On the other hand it is not necessary to keep all the paths times at each node either, only one time is needed for each scale factor. The maximum number of values that need to be saved is the length of the longest path that contains multicycle operations. This cannot be more than the number of control steps in the graph.

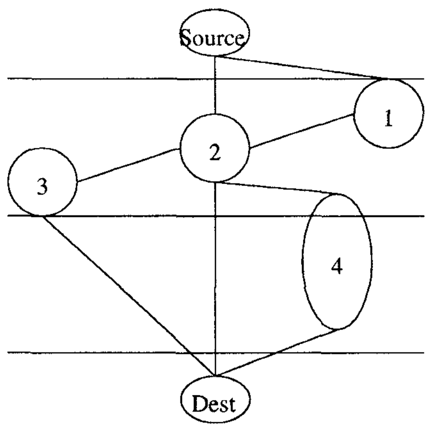

Figure 3: Hypothetical example.

Given that, each node considers all the values from it's successor nodes and keeps the maximal value for each scale factor. The maximum can come from different successors for different scale factors. Each node computes it's maximal value as the maximum of these values (scaled by the scale factor). The correct information is thus propagated back to the *source* node.

The algorithm is given below. Each node has an array *Paths* such that *Paths*[i] is the maximal accumulated delays along a path from this node to *destination* that spans i clock cycles. Initially *Path*[i] $= -\infty$. Each node also remembers its maximal such i. To execute this algorithm the call is: LongestPathTo(*source*, *destination*).

```
Begin
If not visited
    For each output arc
    Let Child be the destination of the arc
        FindLongestPath(Child, destination)
        for each valid $i$
            let $j$ be the path length in clock cycles
            for that path including ThisNode
            $ThisNode.Paths[j] =$
                $max\{Child.Paths[i] + Arc.Weight+$
              $ThisNode.Weight,$
                    $ThisNode.Paths[j]\}$
        $ThisNode.Max = max\{Paths[k]/k\}\ \forall k$
End
```

Let n and v be the number of operators and arcs in the original CDFG and let s be the number of control steps in the design. The new graph contains at most $n+v+2$ nodes (since in the worst case all arcs cross the cycle line) and at most $4v$ arcs (for the same reason, the control arcs, and the connections to *source* and *destination*). Each arc is considered only once and for each arc the father node considers all the possible scale factors (in the worst case – the number of steps). The complexity is therefore $O(vs)$.

Let d be the depth of the graph. d is the length of the path that spans the maximum number of control steps. In the average case, d should be much less than s. The reason being that long multicycle paths result in area overhead since all the inputs along the path need to be preserved and stable throughout it's execution. This means that more registers and sometimes function units will be needed while the saving in time can be insignificant. Minor changes to the schedule, such as splitting such paths so that intermediate values are latched, can reduce the length of these paths and thus reduce d.

The time complexity of the algorithm in the average case is thus linear with the number of arcs in the CDFG.

3 Synthesizing timing correct designs

The algorithm presented in the previous section estimates the minimal clock cycle time needed for the design to be functionality correct. This estimate needs the delay added by the control unit and does not consider the wire delays.

A HLS system needs to synthesize a design that meets the user clock cycle constraint and therefore these factors should be taken into account. Since estimation of these missing values is not available before the actual synthesis we propose that the synthesis will be performed in several iterations.

At first, the behavioral description is synthesized in the classic manner where only the intrinsic delays are considered. We use the Force Directed Scheduling algorithm [7]. After scheduling and binding, the overall area for the datapath and the control unit can be estimated. This area can be used to extract bounds on the wire delays. The delay that is contributed by the control unit can be estimated by a regular delay predictor as the maximum path in the control logic.

To find a schedule that meets the user clock constraints we use a simple search scheme that converges in relatively few steps and does not impose any hardships on the scheduler. We will make the search transparent to the scheduler by trying to find a clock cycle time that will eventually result in a correct timing (according to the user clock cycle). Essentially we do not change the CDFG, only the clock cycle time. Under this scheme, given a user clock cycle C we try to schedule the design with a clock cycle c ($c \leq C$).

After the first iteration and estimation of the timings we derive and fix the control unit time so that all arcs that refer to *cu* have a determined value. This is a good enough approximation since the control unit critical path does not change much between the different schedules. The area for the design will also remain approximately the same for different schedules so the wire delays can be fixed too. We can now find e, the minimum clock time for the synthesized design. Let c and C be as above and w be the average wire delay. We already have one synthesized design that can run correctly at $e + \alpha w$ (where α is related to the average number of chained units in a clock cycle). With the lack of any other information we assume that the ratio $\frac{c}{e}$ will be similar when we synthesize with other clock cycles. We therefore choose $c' = \frac{c}{e}(C - \alpha w)$ as a new clock cycle and schedule again (no dummy operators). In most cases this should result in having two data points bounding the wanted clock cycle. If not, we choose c' again in the same way. Once two bounding values are found, a simple binary search can find the needed value. If the accepted tolerance is $1ns$, for example, this process can takes about 3 iterations.

As a result of this process we get a synthesized design which stands a better chance of meeting the user clock constraints when synthesized into a netlist. The time overhead spent in this stage is significantly less than the time that will be spent in completing a design cycle that goes through several RTL and logic syntheses because of timing problems.

4 Experimental results

We have applied our algorithm to many designs. Once a schedule that meets the user clock cycle constraint is found the RTL code is created. The critical paths identified by the algorithm where verified to be less than the user's clock cycle.

The results were verified after RTL synthesis using a static timing analyzer. For designs that contain chained operators our algorithm was a worst case estimate because of issues discussed above.

As an example we take the well known elliptic wave filter description. We try to synthesize a design that meets a 20ns clock cycle. We omit the details about delays of functional units and interconnect since these are specific to the technology we used. The first attempt to schedule with a 20ns clock cycle resulted in a minimal clock of 30ns. This is the result of a chaining of two adders whose delays are approximately 10ns. The longest path in that clock cycle consists of the two adders and three multiplexers and a register. The delays on these units add up to 30ns. This illustrates the importance of our approach. The next attempt using a 13.25ns clock results in a 17ns estimate. A third attempt with a 16.6ns clock results in 20.15ns estimate. This is within our tolerance and thus the search ends.

5 Conclusion and Further Research

We have shown that some real design issues are frequently not considered in many HLS systems. It is not surprising that there are many more dimensions to the minimization problems that HLS systems try to solve.

We presented an algorithm for better estimating the timing of a scheduled description. This algorithm have been used to synthesize a design that will meet the timing constraint. The algorithm is also integrated into an automated constraint driven partitioning algorithm for behavioral descriptions. Based on the timing estimation partitions are created such that the timing does not violate users constraints.

In a recent publication[9] the authors present algorithms to estimate lower bounds on performance and resources for a given CDFG. The performance (i.e. clock cycle) estimation does not take into consideration the interconnect delay. We believe that such algorithms could be combined to estimate the interconnect area and delay together with a better estimate of the clock cycle time. This might be a direction that we will follow to extend our research.

Further improvements to the methods we presented can include better computation of the control unit delay specific to each unit and better wire delay estimation. When such wire delay estimates are available they could be added to the arcs in the graph as opposed to the heuristic αw factor. Also, better models of delays for chained units can be derived and integrated into the computation. It is also possible that a better scheme of scheduling and binding can be found such that no iterations will be needed.

References

[1] F. Brewer and D. Gajski. Chippe: A system for constraint driven behavioral synthesis. *IEEE Transactions on Computer-Aided Design of Integraged Circuits and Systems*, 9(7):681–695, July 1990.

[2] B. Gregory, D. MacMillen, and D. Fogg. ISIS: A system for performance driven resource sharing. In *Proceedings of the 29th Design Automation conference*, pages 285–290, 1992.

[3] Y-W Hsieh, S. P. Levitan, and B. M. Pangrle. Incorporating interconnection delays in VHDL behavioral synthesis. In *4th ACM/SIGDA Physical Design Workshop*, Lake Arrowhead, California, April 1993.

[4] R. Jain, K. Kucukcakar, M. J. Milnar, and A. Parker. Experience with the ADAM synthesis system. In *Proceedings of the 26th Design Automation conference*, pages 56–61, 1989.

[5] R. Jain, M. J. Milnar, and A. Parker. Area-time model for synthesis of non-pipelined designs. In

Proceedings of the IEEE International Conference on Computer Aided Design, pages 48–51, 1988.

[6] M. C. McFarland. Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions. In *Proceedings of the 23th Design Automation conference*, pages 474–480, July 1986.

[7] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behaviral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design of Integraged Circuits and Systems*, 8(6):661–679, June 1989.

[8] K. Roy and J. A. Abraham. A novel approach to accurate timing verification using RTL descriptions. In *Proceedings of the 26th Design Automation conference*, pages 638–641, 1989.

[9] A. Sharma and R. Jain. Estimating architectural resources and performance for high-level synthesis applications. *IEEE Transactions on Very Large Scale Integration*, 1(2):175–190, June 1993.

[10] J-P Weng and A. C. Parker. 3D scheduling: High-level synthesis with floorplanning. In *Proceedings of the 28th Design Automation conference*, pages 668–673, 1991.