Contents

List of Figures

Clock-Driven Performance Optimization in Interactive Behavioral Synthesis

Hsiao-ping Juan, Daniel D. Gajski and Viraphol Chaiyakul

Department of Information and Computer Science

University of California, Irvine, CA 92717-3425

Abstract

In interactive behavioral synthesis, the designer can control the design process at every stage, including modifying the schedule of the design to improve its performance. In this report, we present a methodology for performance optimization in interactive behavioral synthesis. Also proposed in this report are several quality metrics and hints that can assist the user in utilizing the proposed methodology. When the user is optimizing the performance of the design, one important decision is the selection of a clock period. We have developed an algorithm to estimate the effect of different clock periods on the execution time of the design. This algorithm can be used to facilitate clock period selection by the user in order to optimize the performance of the design. We have tested our methodology on several benchmarks. The experimental results support the proposed methodology by demonstrating an average improvement of 46.2% in design performance.

1 Introduction

Many years of research have been dedicated to the development of automatic behavioral synthesis tools [3][4][6][11]. Recently, several EDA vendors have also introduced commercial products based on behavioral synthesis. In these systems, designs are obtained with minimal user interaction. The only means of controlling the output from such systems is via constraints expressed in terms of area and/or performance.

Automating behavioral synthesis is a very complicated issue. However, it is well accepted that mojority of synthesis tasks are NP-complete problems. In addition to complexity, the order in which these synthesis tasks are performed also has an impact on both the efficiency and results of the overall synthesis process. Hence, the resulting designs sometimes cannot satisfy the performance or area demands of real-world constraints. When the design produced by automatic behavioral synthesis is not a good one, it presents the user with the following dilemma: if the user modifies the input description and constraints and resynthesizes it, he/she may get a completely different and unpredictable design, which still may not satisfy the constraints. In addition, low level tasks such as placement and routing, which usually require tremendous amount of time, need to be done in every iteration. On the other hand, if the user modifies the output design manually, then he/she needs to spend considerable effort to understand the synthesized result and to prove the correctness of the modified design.

To develop a feasible approach for behavioral synthesis, we have substituted the goal of a completely automated, "push-button" synthesis system with one which attempts to maximally utilize the human designer's insights. This approach, as opposed to automatic behavioral synthesis, is called **interactive behavioral synthesis**. Using interactive behavioral synthesis, the users can control the design process, observe the effects of design decisions, and manually override synthesis algorithms at will. For example, after an automatic algorithm is used to schedule a behavior, if the scheduling result cannot satisfy the performance constraint, the user can manually modify the result. This interactivity will allow synthesis systems to generate high-complexity designs of acceptable-quality in the immediate future, instead of the many years of research needed to improve the current automatic synthesis techniques. With this goal in mind, we have implemented an interactive behavioral synthesis system called the *Interactive Synthesis Environment (ISE)*, and developed a design methodology for its use.

The maximum execution time of a design can be defined as the product of the clock period used in the design and the maximum number of clock cycles. Hence, to optimize the performance of a design, it is important to select the clock period wisely, as well as to minimize the number of clock cycles. Moreover, the number of clock cycles required to finish all the operations in a design also depends upon the clock period. Therefore, a bad choice of the clock period could severely affect the performance of the design.

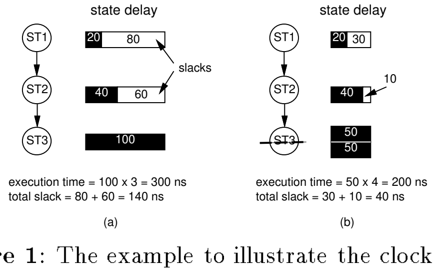

Figure 1: The example to illustrate the clock slacks

Figure 1(a) shows a behavior description that has three states executing sequentially in the order of $ST1$, $ST2$, and $ST3$. The state information is given by the user as a part of the input behavioral description to the interactive synthesis system which will schedule states to clock cycles during synthesis process. In the example, the times required to execute states $ST1, ST2$, and $ST3$ are 20 ns, 40 ns and 100 ns, respectively. If the longest state delay is used as the clock period, the clock period of this design would be 100 ns, and consequently, the total execution time would be $100 \times 3 = 300$ ns. However, states $ST1$ and $ST2$ both have much shorter delays than $ST3$ has. As a result, by using 100 ns as the clock period, the circuit would be idle for 80 ns in state $ST1$ and idle for 60 ns in state $ST2$. The amount of time that the circuit remains idle during a clock cycle is called *slack* [5]. That is, there is 80 ns slack in $ST1$ and 60 ns slack in $ST2$. Clearly, the execution time of a design can be optimized by minimizing the slack time, since minimizing slack minimizes the idle time of the circuit.

Minimizing the slack in each state can be done by executing states that have long delays in more than one cycle. For example, if state $ST3$ in Figure 1(a) is executed in two clock cycles, the clock period would be 50 ns and the total execution time would be $50 \times 4 = 200$ ns, as shown in Figure 1(b). Note that in this case, the total slack is reduced from 140 ns to 40 ns. Due to the reduction of clock slack, the execution time decreases from 300 to 200 ns.

Having observed the impact of the clock period on the total execution time of a behavior, we develop a methodology to optimize the performance of a design by first selecting the clock period and then minimizing the number of clock cycles needed when the chosen clock period is used. We also develop several quality metrics and hints to assist the user while selecting a clock period to optimize the performance of a design. For instance, *state delay* used in the discussion above, is one of the metrics that we provide for interactive performance optimization.

The rest of the report is organized as follows. In the next section, we shall briefly review previous research in this area. A detailed discussion of the proposed methodology and the quality metrics and hints needed for realizing this methodology is given in Section 3. In Section 4 and 5, we shall present the algorithm used to generate the required quality metrics and hints. Finally, we present experimental results and provide conclusions.

2 Previous Work

Several previous papers have addressed the importance of user-interaction with synthesis systems. The ACE graphical interface [2] allows the user to place and connect functional nodes to create a graph that specifies the desired behavior. After the initial graphical specification is obtained and before synthesis starts, some transformation techniques may be applied to the specification to obtain a specification that could be mapped into a more efficient hardware. The user interacts with the system by accepting or rejecting the system's transformation decisions or specifying transformations manually. Nevertheless, ACE does not allow the user to interact directly with the synthesis tasks. Thus, the user cannot perform any performance optimization interactively in ACE.

RLEXT [8] [9] is an interactive tool which allows a user to manually reschedule a design's behavior or modify a design's structure by adding or deleting components and interconnects. However, RLEXT does not provide the user with feedbacks as to the current design's quality to help the user decides how to improve either the performance or the area of the design.

The system AMICAL [7] allows the user to mix automatic and manual design. The user may start a design manually and ask AMICAL to finish it. Alternatively, the user can execute the synthesis tasks step by step. At each step, the user has the choice to continue the synthesis automatically or manually. Yet, AMICAL does not provide any hint to help the user in making important design decisions such as selecting the clock period.

On the other hand, several efforts have addressed the issue of optimizing the performance of a scheduled behavior. In [1], an algorithm is proposed to reduce the clock period at the binding phase of synthesis. However, this work was done based on the assumption that each state in the scheduled behavior cannot be executed in more than one clock cycle, while we attempt to optimize a design's performance by allowing the states with longer delays to be executed in several cycles.

A post-synthesis technique that attempts to reschedule the controller of a design, in order to optimize the performance of the design, without changing the datapath is presented in [10]. The rescheduling of the controller essentially generates a new controller that would execute the states with long delays in several clock cycles. However, this technique does not take the control unit delay into account. When the number of clock cycles is very large, the control unit tends to become very complex and the control unit delay can contribute significantly to the clock period and therefore, should not be neglected. By considering the control unit delay, our algorithm provides a more realistic clock period calculation.

3 Methodology

Figure 2 shows our proposed methodology for optimizing the performance of a scheduled behavior. This methodology basically consists of two steps. In the first step, the user selects a clock period such that the total clock slack in the given scheduled behavior is minimized. If the chosen clock period cannot satisfy the performance constraint, the user tries to further minimize the number of clock cycles required by the longest execution path. These two steps will be described in detail in following sections.

Reducing clock slack and reducing the number of cycles are basically performance optimization techniques at the behavioral level, because the user need to modify only the schedule and behavior of the design. If both of these techniques fail to satisfy the performance constraint of the design, the user may modify the design at either the structural or the physical level. At the structural level, the performance of a design can be improved by using faster components, while at the physical level, the performance can be improved by modifying the floorplan to reduce wire delays. This report focuses on performance optimization at the behavioral level.

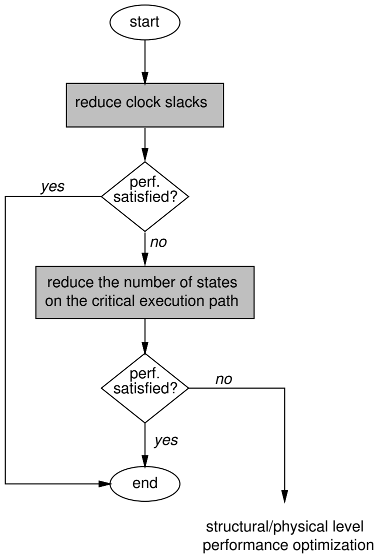

Figure 2: The Methodology for performance optimization

3.1 Reducing Clock Slacks

We will illustrate the methodology with the example in Figure 3. In Figure 3, the scheduled behaviors are represented using the state-actions table (SAT) format. In the SAT, the columns PS and NS are the "present" and the "next" states, respectively, $SCOND$ gives the condition for a next-state transition, AC shows the assignment condition for each action, and the column $ACTIONS$ lists all operations in the behavior. Also shown in Figure 3 is the quality metric *state delay (ST Delay)*, which shows the maximum time needed to execute each state.

Assume that the delay of a multiplier is 40 ns, the delay of an adder is 20 ns, and the setup time of a register is 5 ns. Figure 3(a) shows that the state delay of $ST1$ is 65 ns due to the register setup time and the chain of the addition and the multiplication operations. Similarly, the delays of the states $ST2$ and $ST3$ are 45 ns and 25 ns respectively. If the design is synthesized without further optimization, the clock period of this design would be 65 ns, which is the longest delay per state, and the total execution time would be $65 \times 3 = 195$ ns. Note that the total slack when the clock period is 65 ns is equal to 60 ns.

In order to reduce clock slack, the user can try to execute the states that have longer delays in several clock cycles. For example, the state $ST1$ can be executed in two clock cycles by splitting the chain of the addition and the multiplication operations. As shown in Figure 3(b), a new state $ST4$ is inserted between $ST1$ and $ST2$ and the addition operation from the chain is executed in $ST1$ and the multiplication operation is executed in $ST4$. As a result, the clock period now becomes 45 ns instead of 65 ns, the total slack is 40 ns, and the total execution time is reduced from 195 ns to $45 \times 4 = 180$ ns.

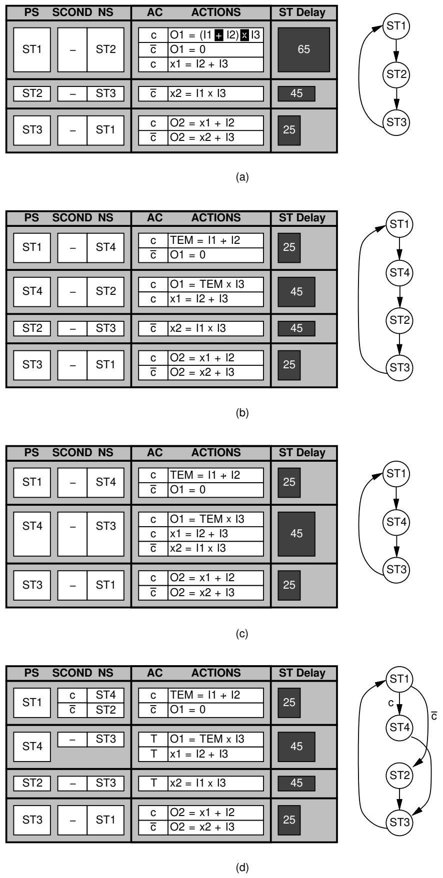

Figure 3: An example to demonstrate the performance optimization methodology

3.2 Reducing the Number of States on the Critical Path

After the clock period is determined, the user can try to further reduce the number of states in the scheduled behavior in order to reduce the total execution time. The number of states can be reduced by either *merging* states or *rewriting* the state transitions.

In Figure 3, after splitting the state $ST1$ into two states to reduce clock slack, the resulting behavior has a 45 ns clock period and four states. Note that the delays of the states $ST4$ and $ST2$ are both 45 ns and the operations in the states $ST4$ and $ST2$ are executed under mutually exclusive conditions. Therefore, if these two states are merged, the clock period after merging should remain 45 ns. Figure 3(c) shows the result of merging $ST4$ and $ST2$ into one state $ST4$. The total execution time is reduced from $45 \times 4 = 180$ ns to $45 \times 3 = 135$ ns. However, we would like to emphasize here that merging states inappropriately may increase the execution time of the design instead of decreasing it. The reason is that, if there exist data dependencies between the operations in the states that are merged, the state delay may increase, and consequently, the clock period and execution time may increase.

The number of states can also be reduced by rewriting the state transitions of the scheduled behavior. For example, consider the SAT shown in Figure 3(b). The execution of this SAT goes through all four states sequentially. However, the operations in state $ST4$ are executed only when the condition c is true, while operations in the state $ST2$ is executed only when the condition c is false. That is there is one clock cycle where there is no operation being executed. This wasted clock cycle can be eliminated by rewriting the state transitions. In this instance, the user can factor out the condition c and use it as the state transition condition to enter the state $ST4$. The result of rewriting the state transitions is shown in Figure 3(d). Notice that the resultant behavior consists of two execution paths: $ST1 \rightarrow ST4 \rightarrow ST3$ when c is true and $ST1 \rightarrow ST2 \rightarrow ST3$ when c is false. When either of the two paths is taken, the number of cycles is three. Hence, the execution time is reduced from 180 ns to 135 ns. Unlike state merging, rewriting state transitions will not increase state delay. Nevertheless, rewriting state transitions usually makes the control unit become more complicated and may consequently increases the clock period.

3.3 Quality Metrics

Having presented the methodology for performance optimization, we will now summarize the quality metrics and hints required to help the user utilize the proposed methodology.

Since the clock period of a synchronous design can be estimated by finding the maximum state delay over all states in the design, we use **state delay** as a quality metric. As shown in the previous example, the state delay metric not only gives the clock period of the current design, but also shows the slack in each state. Thus, it is one of the most important metrics in performance optimization.

Moreover, knowing from the methodology that the clock slacks can be reduced by selecting a shorter clock period and executing the states with longer delays in several clock cycles, it is clear that a useful design hint for the user is a shape function showing **clock periods versus the execution time** of the behavior. The reason for a shape function instead of a single point representing the period with minimum execution time is that there may exist more than one clock period that can produce designs that satisfy the performance constraint. In such a case, question of which clock period to choose is left to the user.

A scheduled behavior may contain state branches or iterations constructs (such as loops). This makes it difficult for the user to determine or observe execution paths. In addition, in our methodology each state in the scheduled behavior can now be executed in more than one clock cycle when different clock periods are available, the longest execution path may be different. If the user is to reduce execution time, the longest execution paths must be known. We provide the user with **execution paths** hint which highlights the states on all execution paths in a state transition graph display. The user can start from the longest path and go through each path. There are two reasons that we provide the user all the execution paths instead of only the longest one. First, there may be more than one path whose lengths are maximum. In this case, the maximum execution time of the behavior can be reduced only when all the paths are shortened. Second, the length of the second longest execution path usually gives an indication of what the execution time will be if the longest path is shortened.

In summary, to assist the user in optimizing performance of a design in interactive behavioral synthesis, we need to provide the execution paths of the scheduled behavior, the length of each execution path, the shape function of clock periods versus execution time, and the state delay metric. The state delay metric can be obtained by simply summing up all the operation delays in the critical path in each state, and we will not elaborate it further in this report. In the next section, we shall formulate the solution for our requirement.

4 Problem Definition and Algorithm Outline

Our problem can be defined as follows:
Given the following:

- a component library,
- a behavioral description which could contain state information,
- and a range of clock periods to be examined by the algorithm (with *clkmin* as a lower bound, and *clkmax* as an upper bound)

The goal of our algorithm is to determine the shortest possible execution time for each clock period within the *clkmin* and *clkmax* range; given that the resulting design can use any number of components from the library as needed.

Figure 4 illustrates our problem definition. Given are a state transition graph STG, where the states are represented using data flow graphs, DFG_0, DFG_1, DFG_2 and DFG_3, a component library, and the range of clock periods allowed (20, 200 ns). Our algorithm then produces a shape function in terms of clock periods versus execution times.

Our algorithm generates the shape function of clock period versus execution time in four basic steps.

1. **Execution path identification.** The first step in our algorithm is to identify all the execution paths in a given scheduled behavior.
2. **State shape function generation.** Next, our algorithm generates a shape function of clock period

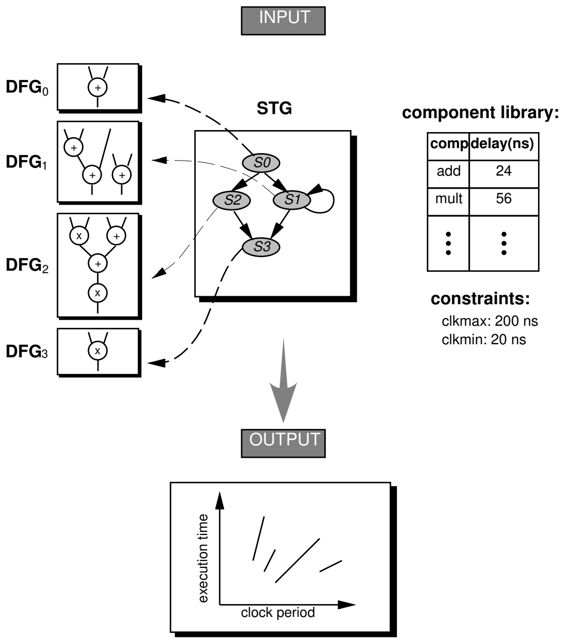

Figure 4: An example illustrating the inputs and outputs of the problem

versus number of cycle for each state in the given scheduled behavior. In this step, only the datapath delay is considered.

3. **Shape function merging.** After the shape function of each state is obtained, the algorithm then tries to merge the shape functions of all the states on each execution path. Then, all the shape functions of the execution paths are merged into a single shape function representing clock periods versus the number of clock cycles for the entire behavior.
4. **Control unit delay estimation:** Finally, our algorithm estimates the control unit delay and updates the shape function accordingly. Then the shape function of clock periods versus execution times can be generated by multiplying the clock periods to the corresponding number of cycles.

Details of each of the steps above will be discussed in the following sections.

5 Algorithm

Before we present the algorithm, we will show the underlying design model used for clock period calculation.

5.1 Design Model

The design model for clock period computation, shown in Figure 5, is similar to the one presented in [5]. In this model, the datapath consists of registers, functional units and tri-state drivers. A two level bus structure is assumed for the interconnection across the registers and functional units. A typical datapath operation involves reading operands from the registers, computing the result in the functional units, and writing the result into a destination register. Operation chaining is supported in this model by allowing connections from the output ports of functional units to the input ports of other functional units. In addition, multi-cycled operations are allowed.

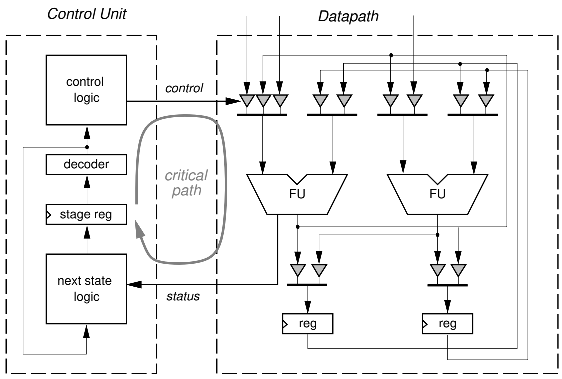

Figure 5: Design model for clock period calculation

The control unit consists of the state register, a decoder, the control logic to drive the control lines for the datapath components, and the next-state logic to compute the next state to be stored in the state register. Status lines from the datapath carry the results of comparison operations to the next-state logic.

The clock period is determined by the longest register-to-register delay. Typically, the path through the control logic has the largest delay, as shown in Figure 5. Consequently, the minimal clock period is equal to or greater than the sum of all the delays associated with the components and the wires in the path. We can formulated the computation as follows:

$$clk = Tp_{SR} + T_{DEC} + T_{CL} + T_{TR} + T_{FU} + T_{NS} + Ts_{SR} + T_{Wire}$$

where:

Tp_{SR} and Ts_{SR} are the propagation delay and the setup time of the state register, respectively,
T_{DEC} is the delay of the decoder,
T_{CL} is the delay of the control logic,
T_{TR} is the delay of the tri-state driver,
T_{FU} is the delay of the functional units,
T_{NS} is the delay of the next-state logic, and
T_{Wire} is the total delay of the wires in the path.

However, the wire delay cannot be meaningfully estimated without a floorplan, which is not available at this early stage of synthesis. Therefore, the clock period clk is approximated using the following equation:

$$clk = T_{DP} + T_{CU}$$
$$T_{DP} = T_{TR} + T_{FU}$$
$$T_{CU} = Tp_{SR} + T_{DEC} + T_{CL} + T_{NS} + Ts_{SR}$$

where T_{DP} is the delay of the datapath, and T_{CU} is the delay of the control unit.

5.2 Execution Path Identification

In the general case, a scheduled behavior may consist of state branches or loops constructs in the behavior. We assume the structured behavior, that is, *goto* constructs are not allowed.

Let STG denote a state transition graph which consists of a set of states $\{S0, S1, \cdots, Sn\}$. Assume that state $S0$ is the *initial state*. An *end state* is defined as a state that transits back to the initial state or does not transit to any other state. There may be more than one end state in an STG. An execution path of STG is a sequence of states that connect the initial state $S0$ to an end state Se. Note that there may exist loops in an execution path. We denote a loop that starts from state Si and loops back or exits at state Sj as $(Si, \cdots, Sj)*$. For example, consider the state transition graph STG shown in Figure 6. We can see that the initial state is $S0$ and there is only one end state $S3$. Clearly, there exist two execution paths, as shown in Figure 6: $S0 \rightarrow S1 \rightarrow S3$, and $S0 \rightarrow (S2)* \rightarrow S3$.

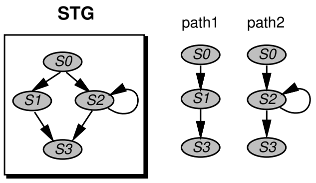

Figure 6: An example to illustrate the execution paths in the state transition graph

Our algorithm uses a depth-first traversal, starting from the initial state and finishing when all the states are visited, to identify loops in a state transition graph. During the traversal, if a state which has already been visited before is visited again then there exists a backward edge, which indicates a loop. The algorithm marks all the states in the loop, deletes the backward edge, and continues the traversal. After the depth-first traversal, the state transition graph is now an acyclic graph since all the backward edges have been deleted.

Having obtained an acyclic state transition graph, the next step is to find out all the paths in the graph starting from the initial state and ending at one of the end states. This is done by another depth-first traversal. This traversal starts from the initial state and continues traversing the next states until it reaches an end state. Reaching an end state indicates that an execution path has been found. The traversal routine then back-tracks to the closest predecessor state which has branches that haven't been traversed, and continues the traversal from another branch until it finally reaches an end state again. Clearly, after all the state branches are traversed at least once by the depth-first traversal routine, all existing execution paths in the state transition graph have been identified.

5.3 State Shape Function Generation

Given a data flow graph DFG_i of the state S_i and the range of clock period allowed, (*clkmin, clkmax*), the goal of this step is to generate the shape function of clock periods versus the minimum number of cycles that the state S_i requires.

Since the clock periods are in the real number domain, clearly it is infeasible to attempt to go through all possible clock periods and estimate the minimum number of cycles that the state requires for each of them. However, the possible numbers of cycles are in integer domain. Therefore, instead of computing the minimum number of cycles required for all possible clock periods, the shape function is generated incrementally by fixing the number of cycles, and then computing the minimum clock period for the fixed number of cycles using the procedure *MinClkPeriod* outlined in Figure 7. This process produces one point (clock period, number of cycles) in the shape function. To obtain the entire shape function, we iteratively increase the number of cycles until the clock period produced by the procedure *MinClkPeriod* is smaller than *clkmin*. If we assume that the algorithm estimates that the shortest possible clock period for executing the data flow graph in i cycles is clk_i and similarly, the shortest clock period for $i+1$ cycles is clk_{i+1}, then we can conclude that for any clock period clk_j, $clk_{i+1} \leq clk_j \leq clk_i$, the minimum number of cycles that the data flow graph would be scheduled into by using clk_j is $i+1$.

```
Procedure: MinClkPeriod
  Inputs: a data flow graph DFG, the number of cycles N;
  Output: the minimum clock period
begin Procedure
  Cstep = 1;
  ComputePathLength(DFG);
  MaxPathLength = delay of the longest path in DFG;
  MinClk = MaxPathLength/N;
  InsertReadyOps(DFG, PList);
  while (PList ≠ ∅) do
    if Cstep = N then
      schedule all the non-scheduled operations;
      MinClk = maximum state delay;
      PList = ∅;
    else
      op = First(PList);
      if op is a single-cycled operator then
        determine chaining or non-chaining;
        schedule op and update MinClk;
      else
        determine the number of cycles of op;
        schedule op and update MinClk;
      end if;
      InsertReadyOps(DFG, PList);
      Cstep = Cstep + 1;
    end if;
  end while;
  return MinClk;
end Procedure
```

Figure 7: The procedure to estimate the minimum clock period, given N cycles

The procedure *MinClkPeriod* is adapted from ASAP scheduling except that, instead of minimizing the number of cycles given a clock period, it minimizes the clock period given the number of cycles. A brief explanation follows.

Given a data flow graph DFG, the procedure $MinClkPeriod$ first computes the path length for each of the operations in DFG. The path length of an operation is defined as the longest path delay from this operation to an output node. Therefore, the maximum path length, *MaxPathLength*, of all operations in DFG is the critical path length. The next step of the procedure involves determining whether a ready operation can be scheduled. In ASAP scheduling, all ready operations are scheduled as soon as possible, as long as the clock period constraint is not violated. In our procedure, whether a ready operation can be scheduled or not and whether chaining or multi-cycling should be performed depends upon its effect on the clock period.

The variable $MinClk$ is initialized to the optimal clock period $MaxPathLength/N$, where N is the number of cycles that DFG would be scheduled into. Then, for each operation in the ready list $PList$, we first determine whether it would be a single-cycled operation or multi-cycled operation using the delay of the operation and the current clock period $MinClk$. If the operation delay is less than or equal to $MinClk$, which means it is a single-cycled operation, we then need to decide whether it could be chained with its predecessor in the current state or should be deferred to the next cycles. If the operation delay is larger than $MinClk$, which means that it is a multi-cycled operation, we must decide whether to schedule it across $\lfloor$(operation delay)$/MinClk\rfloor$ or $\lceil$(operation delay)$/MinClk\rceil$ cycles. If the scheduling of an operation increases the clock period, the variable $MinClk$ is updated. Once an operation is scheduled, other non-ready operations become ready and are inserted into the ready list. When the process reaches the last cycle, all non-scheduled operations are scheduled and the procedure returns the variable $MinClk$, which now contains the longest delay of all N cycles, that is, the clock period.

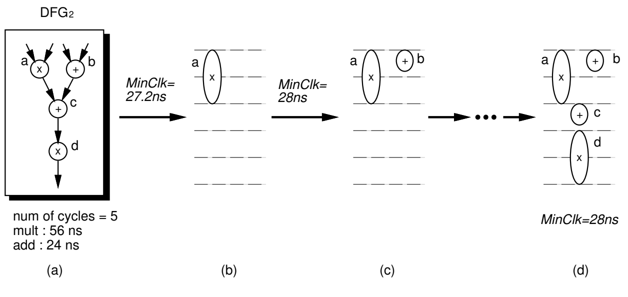

Figure 8: Determining the minimum clock period

Clearly, the quality of this algorithm depends upon how it determines chaining and multi-cycling. We now illustrate how chaining and multi-cycling are determined with the example in Figure 8.

Given that a multiplication operation takes 56 ns and an addition takes 24 ns, the procedure computes a maximum path length of 136 ns. Since the data flow graph would be scheduled into five cycles, the optimal clock period, that is, the current $MinClk$, is 136/5=27.2 ns. In the first iteration of the procedure, it would attempt to schedule the operation a. Given that the delay of operation a is 56 ns and the current clock period is 27.2 ns, a should be a multi-cycled operation and the procedure needs to determine whether to schedule it across $\lfloor 56/27.2\rfloor$=2 cycles or $\lceil 56/27.2\rceil$=3 cycles. If a is scheduled across two cycles, this means that average delay of the first two cycles would be 56/2=28 ns each. Furthermore, if a is finished in two cycles then there will be 3 cycles left (out of the maximum 5 states selected for this example) to schedule c and d, which results an estimated delay per cycle of (24+56)/3=26.7 ns. Thus, the clock period in this case, would be 28 ns. On the other hand, if a is scheduled across three cycles, this gives an average state delay of 18.7 ns for the first three cycles. However, operations c and d now need to be finished within two cycles, which gives an estimated delay per cycle of (24+56)/2=40 ns. Since scheduling the operation a into a two-cycled operation gives an estimation of shorter clock period, the procedure decides to schedule a across the first two cycles as shown in Figure 8(b).

The next iteration involves the scheduling of the operation b. Note that the clock period $MinClk$ has now been updated to 28 ns. Since the delay of the operation b is less than 28 ns, it is a single-cycled operation and its scheduling does not change the current clock period. The result of this iteration is shown in Figure 8(c). The procedure continues this process for the rest of the operations c and d, and the final result is shown in Figure 8(d). The minimum clock period for scheduling the data flow graph into five cycles is 28 ns.

Similarly, we can estimate that the minimum clock periods for scheduling the data flow graph in Figure 8(a) into one, two, three, or four cycles are 136 ns, 80 ns, 56 ns and 56 ns respectively. Therefore, we can conclude that for any clock period larger than 136 ns, the minimum number of cycles that DFG requires is one; for any clock period between 136 and 80 ns, the minimum number of cycles that DFG requires is two, etc. Figure 9(c) shows the resultant shape function.

The algorithm above is repeated for each state in the state transition graph. Finally, one shape function of clock periods versus the number of cycles is generated for each individual state. For example, the shape functions of the states $S0, S1, S2$ and $S3$ in Figure 4 are shown in Figure 9(a), (b), (c) and (d) respectively.

5.4 Shape Function Merging

The shape function merging algorithm contains two steps. It first produces the shape function for each execution path by merging the shape functions of the states on the path. It then tries to merge the shape functions of all the paths into one shape function of clock periods versus the number of cycles for the entire state transition graph. Afterwards, the shape function of clock periods versus execution times can be computed by multiplying the

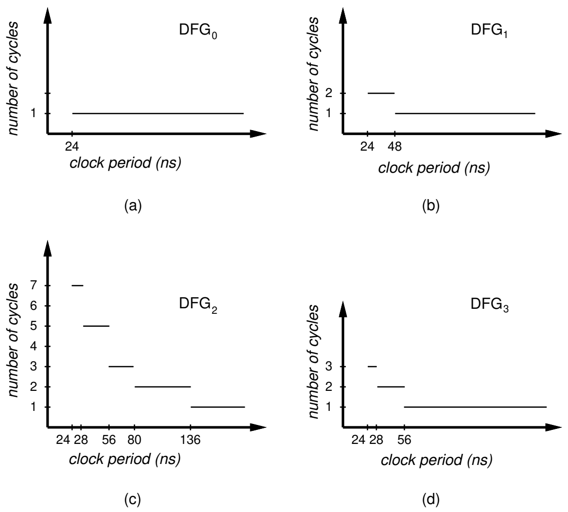

Figure 9: The shape functions of states $S1, S2, S3$ and $S4$

clock periods to the corresponding number of cycles.

If there exists loops in an execution path, the loops have to be bounded and the number of iterations of each loop has to be given to the algorithm, in order to compute the total number of cycles this execution path requires. The approach that we adopt is that our algorithm would identify all the loops in the state transition graph, as discussed in previous section, and then the user should specify the number of iterations of each loop.

Given the shape function of a state that is inside a loop and the number of loop iterations, the algorithm first updates the shape function. That is, given a point (clk, N) in the shape function, assume the number of iterations of the loop is k, then the point is updated to $(clk, k \times N)$. Consider the example shown in Figure 4, given that state $S1$ is in a loop and the loop would iterate three times, the shape function of DFG_1 is updated and shown in Figure 10.

After the shape functions of all the states inside loops are updated, the next step of the merging algorithm is to sum up all the shape functions of the states in an execution path. For example, assume there are only two states in the execution path. Given a clock period clk, if the numbers of cycles of the states when the clock period equals to clk are N_1 and N_2 respectively, then the total number of cycles of this execution path when the clock period equals to clk is $N_1 + N_2$. Figure 10 illustrates the merging of shape functions for path 1 and path 2 identified in Figure 6.

Having obtained the shape functions of clock periods versus the number of cycles for all execution path, the algorithm then tries to merge all the shape functions by, for each clock period, finding the maximum number of cycles among all execution paths. For instance, assume there are two execution paths. Given a clock period clk, if the numbers of cycles of the paths when the clock period is equal to clk are N_1 and N_2 respectively, then the maximum number of clock cycles the state transition graph requires when

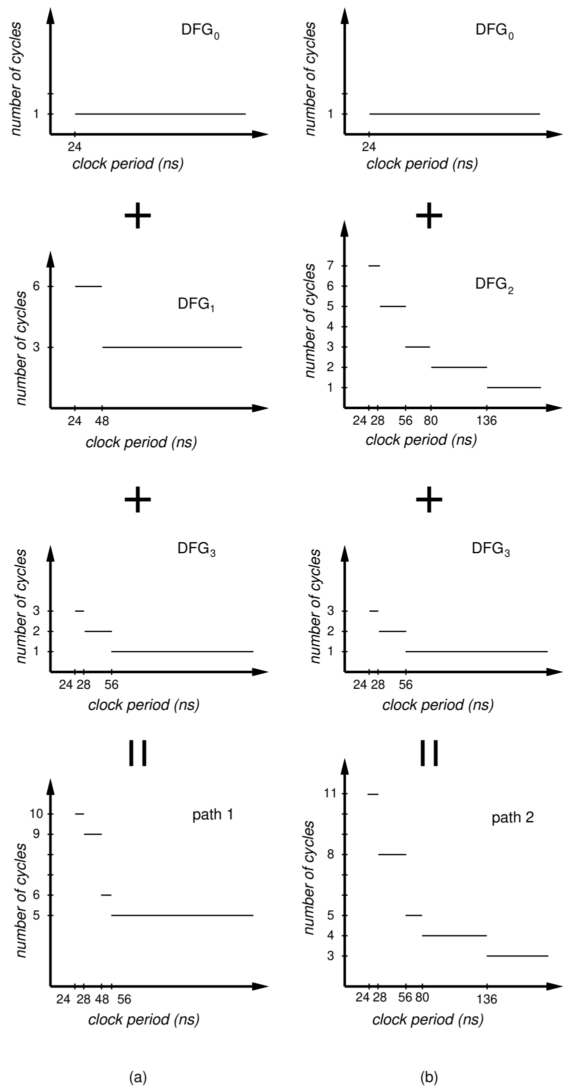

Figure 10: An example to illustrate the merging of state shape functions into execution path shape functions

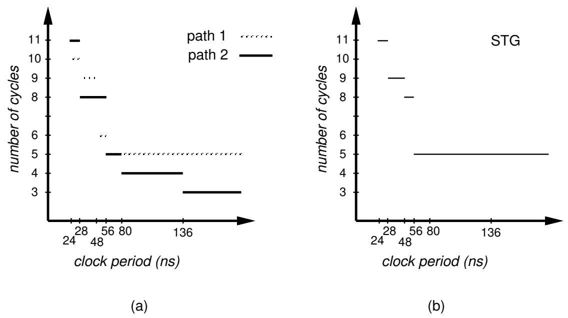

Figure 11: An example to illustrate the merging of execution path shape functions into STG shape function

the clock period equals to clk is $max(N_1, N_2)$. Figure 11(a) shows both of the shape functions of execution paths 1 and 2. Note that given different clock periods, the longest execution paths are different. For example, when the clock period is 28 ns, the execution path 1 is longer than path 2; yet, when the clock period is 24 ns, the execution path 2 is longer than path 1. Figure 11(b) shows the result of merging the shape functions of path 1 and 2, which is the shape function of the entire state transition graph.

5.5 Control Unit Delay Estimation

The control unit sequences a design through a series of cycles, each of the cycles represents the set of datapath operations performed concurrently in the same cycles of the design. In general, if a shorter clock period is used, the total number of cycles would become larger, and consequently, the control unit becomes more complex and the control unit delay is longer. When the number of cycles is very large, the control unit delay contributes significantly to the clock period and cannot be neglected. In the previous section, an algorithm used to estimate the relation between the clock period and the number of cycles by considering only the datapath was presented. In this section, we will explain how to estimate the control unit delay and update the shape function accordingly.

As illustrated in Figure 5, the control unit consists of a state register, a decoder, the control logic and the next-state logic. The control unit may be implemented as random-logic, a read-only memory(ROM), or a programmable logic array(PLA). In this report, we will assume a random-logic implementation as shown in Figure 12.

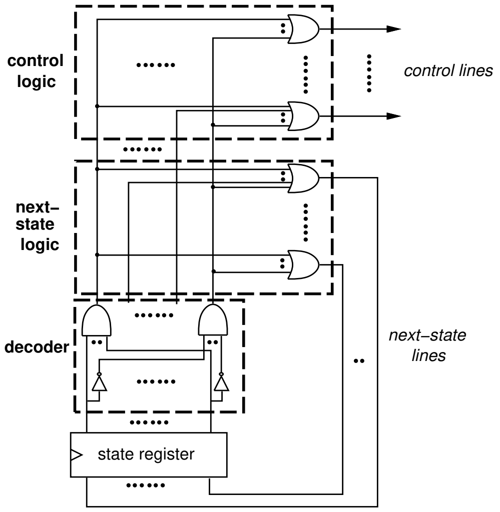

Figure 12: A random-logic implementation of the control unit

Given a clock period clk and the shape functions of all the states in the behavior, the number of cycles each state will be executed in can be obtained easily. Hence, the total number of different control words the control unit has to generate can be determined by summing up the number of cycles of each state. Consider each cycle in the behavior would be a state in the state-machine the control unit implements. We assume that the present states are encoded as binary values and are stored in the state register. Therefore, given that the total number of cycles is N, the state register bitwidth will be $B = \log_2 N$. Taking the B-bit output of the state register as input, the decoder decodes it into an N-bit output such that each bit corresponds to one cycle. The decoder consists of B inverters and N AND-gates. Each inverter is used to invert one bit of state register, and the number of inputs to an AND-gate is B.

One OR-gate is required for each control and next-state line. The size, that is, the number of inputs, of an OR-gate for a control line is identical to the number of states during which the corresponding control line is asserted. For example, a control line of a functional unit will be asserted whenever the functional unit performs an operation bound to it. In the worst case, there exist functional units that are used in every cycle and consequently, the OR-gates that generate the control lines for these functional units would have N inputs. To determine the size of an OR-gate for a next state line, we assume that each next-state line is "toggled" on the average during half of the states in the design since the state values are binary encoded. Thus, the size of each OR-gate driving a next-state line is assumed to be equal to $N/2$.

In reality, the component library usually provide AND and OR-gates with a limited number of inputs. Thus, the AND and OR-gates in our model need to be decomposed into a multi-level implementation when the large AND or OR gates are not available in the library. The multi-level decomposition aims to produce an implementation with the minimal number of levels. This is guided by the fact that a multi-level implementation of a large AND-gate with I number of inputs using AND-gates with a maximum of M inputs is in the form of an M-ary tree. And the height of the tree, which corresponds to the number of levels, is equal to $\lceil \log_M I \rceil$. Therefore, assume that the component library contains AND and OR-gates with a maximum of M inputs, and let T_{AND}, T_{OR}, and T_{INV} denote the delay of an M-input AND-gate, an M-input OR-gate and an inverter, respectively. The following equations are used to estimate the decoder, the control logic, and the next-state logic delay.

$$\begin{aligned} T_{DEC} &= T_{INV} + \lceil \log_M B \rceil \times T_{AND} \\ &= T_{INV} + \lceil \log_M \log_2 N \rceil \times T_{AND} \\ T_{CL} &= \lceil \log_M N \rceil \times T_{OR} \\ T_{NS} &= \lceil \log_M (N/2) \rceil \times T_{OR} \end{aligned}$$

Having obtained T_{DEC}, T_{CL} and T_{NS} using the equations above and the propagation delay and the setup time of the state register from the component library, the control unit delay can be computed by the equation given in previous section.

examples	input specification			slack minimization				# of cycles minimization				total imprv (%)
	clock (ns)	longest path (# of cycles)	exec. time (ns)	clock (ns)	longest path (# of cycles)	exec. time (ns)	imprv (%)	clock (ns)	longest path (# of cycles)	exec. time (ns)	imprv (%)	
addr	34.9	138	4816.2	19.9	192	3820	**20.7**	19.9	137	2726.3	**28.6**	**43.4**
compute	15.2	130	1976	15.2	130	1976	**0**	15.2	66	1003.2	**49.2**	**49.2**
deq0	19.3	134	2586.2	11.4	201	2291.4	**11.4**	11.4	135	1539	**32.8**	**40.5**
plus	15.3	130	1989	9.6	194	1862	**16.4**	9.6	130	1248	**24.9**	**37.3**

Figure 13: Experimental results on four benchmarks

Now let's consider a shape function of clock periods versus the number of cycles generated by the algorithm described in the previous section. The shape function is basically composed of a set of (clock period, number of cycles) points. Each point represents a fixed number of cycles and the minimum datapath delay of the corresponding number of cycles, since the algorithm introduced in the previous section does not take into account the control unit delay. Given a point (clk_i, N), we can use the estimation method discussed above to estimate the delay of the control unit. Assume the delay of the control unit is $T_{CU}(N)$, the algorithm would update the point (clk_i, N) to $(clk_i + T_{CU}(N), N)$. Note that given two points (clk_i, N) and $(clk_{i+1}, N+1)$, where $clk_i \leq clk_{i+1}$, it is possible that $clk_i + T_{CU}(N) \geq clk_{i+1} + T_{CU}(N+1)$. In this case, the algorithm would drop the point $(clk_i + T_{CU}(N), N)$.

After the shape function of clock periods versus the number of cycles is updated, we can obtain the shape function of clock periods versus the execution times by multiplying the clock periods to the corresponding number of cycles.

6 Experimental Results

We have implemented the proposed quality metrics and hints in our synthesis system, *Interactive Synthesis Environment (ISE)*. We have also tested our performance optimization methodology on four examples: addr, compute, deq0, and plus. These examples are modules from the MPEG algorithm, which is an ISO standard compression/decompression algorithm for moving pictures.

The initial specification of the four modules contain states which need to be further refined during interactive synthesis process. For all examples, we have used the VLSI Technology Inc. VDP370 1.0 micron Datapath Element Library [12] to obtain the delays of the components.

Figure 13 shows the experimental results of the four examples. The column *input specification* gives the clock period, the number of clock cycles of the longest execution path, and the maximum execution time of the original behavior. In the first step in our experiment, we used our algorithm to generate a shape function of clock periods versus execution times for each example, and selected the clock period that requires the shortest execution time. The results are shown in the column *slack minimization*. Also shown is the percentage in performance improvement (*imprv%*). It is computed by *(execution time of the original behavior - execution time after slack minimization)/execution time of the original behavior*.

Afterwards, we tried to minimize the number of cycles of the longest execution path by merging the states or rewriting the state transitions. The results are shown in the column *# of cycles minimization*. The results of *imprv%* is computed by *(execution time after slack minimization - execution time after minimizing the number of cycles on the longest path)/execution time after slack minimization*. Finally, the column *total imprv (%)* shows the improvement of the final performance compared to the performance of the original scheduled behavior.

In summary, the slack minimization technique improves the performance of the examples by an average of 16%, except for the *compute* module. We observe that this is due to the fact that the original description of the *compute* module has very small clock slacks. Furthermore, merging states on the longest execution path or rewriting state transitions improves the performance by another 34%. The reason of this remarkable improvement is because that these examples are all loop-intensive and we are able to minimize the number of cycles inside the loops. By reducing even a small number of cycles in the loops, the total number of cycles are reduced tremendously. Overall, we are able to obtain an average of 42.6% improvement of performance compared to the manual scheduled behavior.

7 Conclusion

In summary, we have presented a methodology for performance optimization in interactive behavioral synthesis. This methodology essentially attempts to improve the performance of a given scheduled behavior by first reducing the clock slack in the behavior and then minimizing the number of clock cycles required by the longest execution path. The experimental results on several examples support our methodology by showing an average of 42.6% improvement in performance compared to manual design.

We have also proposed several quality metrics and hints required for the user to utilize the methodology. An algorithm is developed to provide the user a useful hint of what clock period can best reduce clock slacks. Currently, we are working on design hints that can assist the user in merging states and rewriting state transitions.

8 Acknowledgements

This work was partially supported by the Semiconductor Research Corporation grant #94-DJ-146, and we gratefully acknowledge their support.

9 References

[1] S. Bhattacharya, S. Dey, and F. Brglez, "Clock Period Optimization During Resource Sharing and Assignment," in *Proceedings of the 31st ACM/IEEE Design Automation Conference,* 1994.

[2] O. A. Buset, and M. I. Elmasry, "ACE: A Hierarchical Graphical Interface for Architectural Synthesis," in *Proceedings of the 26th ACM/IEEE Design Automation Conference,* 1989.

[3] R. Camposano, and W. Wolf, *High-Level VLSI Synthesis,* Kluwer Academic Publishers, 1991.

[4] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[5] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems,* Prentice Hall, 1994.

[6] P. Hilfinger, and J. Rabey, *Anatomy of a Silicon Compiler,* Kluwer Academic Publishers, 1992.

[7] A. Jerraya, I. Park, and K. O'Brien, "Amical: An Interactive High-Level Synthesis Environment," in *Proceedings of the European Design and Test Conference,* 1993.

[8] D. W. Knapp, "An Interactive Tool for Register-Transfer Level Structure Optimization," in *Proceedings of the 26th ACM/IEEE Design Automation Conference,* 1989.

[9] D. W. Knapp, "Manual Rescheduling and Incremental Repair of Register-Level Datapaths," in *Proceedings of International Conference on Computer-Aided Design,* 1989.

[10] S. Parameswaran, P. Jha, and N. Dutt, "Resynthesizing Controllers for Minimum Execution Time," in *Proceedings of the 2nd Asia Pacific Conference on HDL Standards and Applications,* 1994.

[11] D. E. Thomas, E. D. Langese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench,* Kluwer Academic Publishers, 1990.

[12] VLSI Technology Inc., *VDP370 1.0 Micron CMOS Datapath Cell Library,* 1991.