Reclocking Controllers for Minimum Execution Time

Pradip K. Jha, Sri Parameswaran[1], and Nikil D. Dutt

Technical Report #94-40
September 20, 1994

Dept. of Information and Computer Science
University of California at Irvine
Irvine, CA 92717-3425
Phone: (714) 856-8059
Fax: (714) 856-4056
Email: pradip@ics.uci.edu

[1]Department of Electrical and Computer Engineering, University of Queensland, Australia

Abstract

In this report we describe a method for resynthesizing the controller of a design for a fixed datapath with the objective of increasing the design's throughput by minimizing its total execution time. This work has tremendous potential in two important areas: one, design reuse for retargetting datapaths to new libraries, new technologies and different bit-widths; and two, back-annotation of physical design information during High-Level Synthesis (HLS), and subsequent adjustment of the design's schedule to account for realistic physical design information with minimal changes to the datapath. We present our approach using various formulations, prove optimality of our algorithm and demonstrate the effectiveness of our technique on several HLS benchmarks. We have observed improvements of up to 36% in execution time after straightforward application of our controller resynthesis technique to the outputs of HLS.

Contents

List of Figures

List of Tables

1 Introduction

High-Level Synthesis is composed of many NP-Complete problems, hence many decisions such as scheduling, allocation and binding, are made at an early stage of the design process without good estimates of layout-level information (e.g., wire-lengths and exact area/delay information). Since HLS techniques traditionally do not take into account physical design effects, the performance predicted by HLS tools needs to be recalculated after back-annotation of physical design information into the RT design. One could attempt complete resynthesis of the datapath **and** control by running HLS again with the back-annotated physical design information; however, redoing the scheduling and allocation steps with the new physical design information may generate a completely new datapath for which the previously back-annotated physical design information is useless.

To overcome this dilemma of design, we suggest the resynthesis of the controller alone without changing the overall circuit connectivity. That is, to keep all datapath connectivity and all controller — datapath connectivity the same, and change the controller design itself through a technique called *reclocking*. The controller design can have a different number of states from the initial design, and the controller logic will be different from the initial design. Since resynthesis of the controller does not change the delays very much, we feel that changing the controller design will not adversely affect the wire delays.

Another important motivation for this work is design reuse. The design of datapath is a complex process and completed datapaths are often candidates for design reuse in new projects. Furthermore, with changes in technology libraries (or the requirements for faster designs), system designers would often like to retarget existing datapath designs to new libraries, migrate designs to larger bit-widths, or simply speed up the design to create newer versions with different cost/performance attributes. These design scenarios motivate the need for techniques that allow rescheduling of controllers for a fixed datapath under varying technology library or component attribute conditions. Note that controllers typically have automatic standard-cell implementation and can be easily reimplemented through logic synthesis tools.

In this work, we describe *reclocking*, an approach that modifies the controller without changing the datapath to improve the performance by reducing the total execution time. Given an initial schedule for the design behavior and the updated delays (back-annotated or with a new library) for various paths in the design, our approach first finds a clock-width (*reclocking*) that leads to minimal execution time. It then reschedules and resynthesizes the controller based on this new clock-width.

The rest of this report is organized as follows. Section 2 describes related work. Section 3 defines the problem of *reclocking*, given an initial schedule and datapath delays. Section 4 describes a few results to find the optimal clock-width for different types of codes and presents an algorithm for reclocking. Section 5 demonstrates the efficacy of our approach by applying reclocking on few examples for different design scenarios. Section 6 concludes with a summary.

2 Related work

Various techniques have been proposed to improve the performance of a given design. At the logic level, Leiserson and Sax introduced the concept of retiming[LeSa88]. The technique moves registers across combinational logic to improve performance. Retiming allows the minimization of cycle time or the reduction of the total number of registers. However, as we approach submicron feature sizes, wires contribute significantly to delays. Since wire delays can only be known after layout, the original retiming techniques cannot be applied, – the introduction of registers will change the layout, and thus the timing. Malik et. al [MSBS91] and De Mecheli [DeM91] described methods to improve upon the original approach by changing the circuit topology and using a non-constant delay model. Our work is dual to retiming in the sense that instead of modifying the datapath by moving registers and latches, we reschedule the controller by selecting the best clock-width to improve the performance.

Work has been done to improve the circuit performance at the high-level design phase. Camposano and Ploger [CaPl92] describe the application of retiming to high-level synthesis. Kanehara and Gajski [KaGa91] apply pipelining techniques to improve the performance of a design. This technique inserts latches or registers on critical paths, thus shortening the clock period. [BDBr94], on the other hand, tries to reduce the clock-width at the resource sharing and assignment phases of synthesis. [ZaGa88] maps the RT-level components of the design in such a way so as to meet a required performance bound. They apply a combination of microarchitectural and logic optimization techniques to synthesize RT-level components. The above mentioned works either modify the datapath or incorporate performance improvement techniques during the high-level design phase. Ours is a post-synthesis technique that can incorporate detailed physical-design information and therefore more accurately model the final design.

Narayan and Gajski [NaGa92] use a simple method to estimate clock-widths in high-level synthesis. This method exhaustively searches through the possible clock cycles in 1 ns increments to estimate a clock-width for high-level synthesis. They have not taken wiring or placement into account, nor have they taken the critical paths into account. Since they consider all possible latch to latch timing including false paths to estimate the clock-width, their clock-estimation may be pessimistic. No results are available as to the differences between estimation and final results. There is also no suggestion as to how to find the best clock-width which does not lie on an integer nanosecond clock-width.

In this work, we show that the optimal clock-width lies on an integer division of the largest delays of each state, and that it can be found by searching fewer points in the delay space than the method proposed in [NaGa92]. Using this optimal clock-width we then proceed to reschedule the controller to improve performance.

3 Problem Definition

The output of high-level design is typically specified by a datapath and a controller in a Finite State Machine with Datapath (FSMD) model [GDWL92]. The datapath consists of a netlist of RT level components such as ALUs, registers, muliplexers, etc. The controller generates control signals for each component in the datapath based on the status signals generated by the datapath components. The controller is represented by a finite state machine that specifies what operations are to be performed in each state. Figure 1(a) shows an example design that consists of a 3-state controller and a datapath with an adder and a multiplier. Note that all the "+" functions in this design are single-state operations, whereas the "*" function is a two-state operation. In other words, the data transfer for the unicycle "+" function is completed in a single clock, whereas the data transfer for the multicycle "*" function requires two clock cycles.

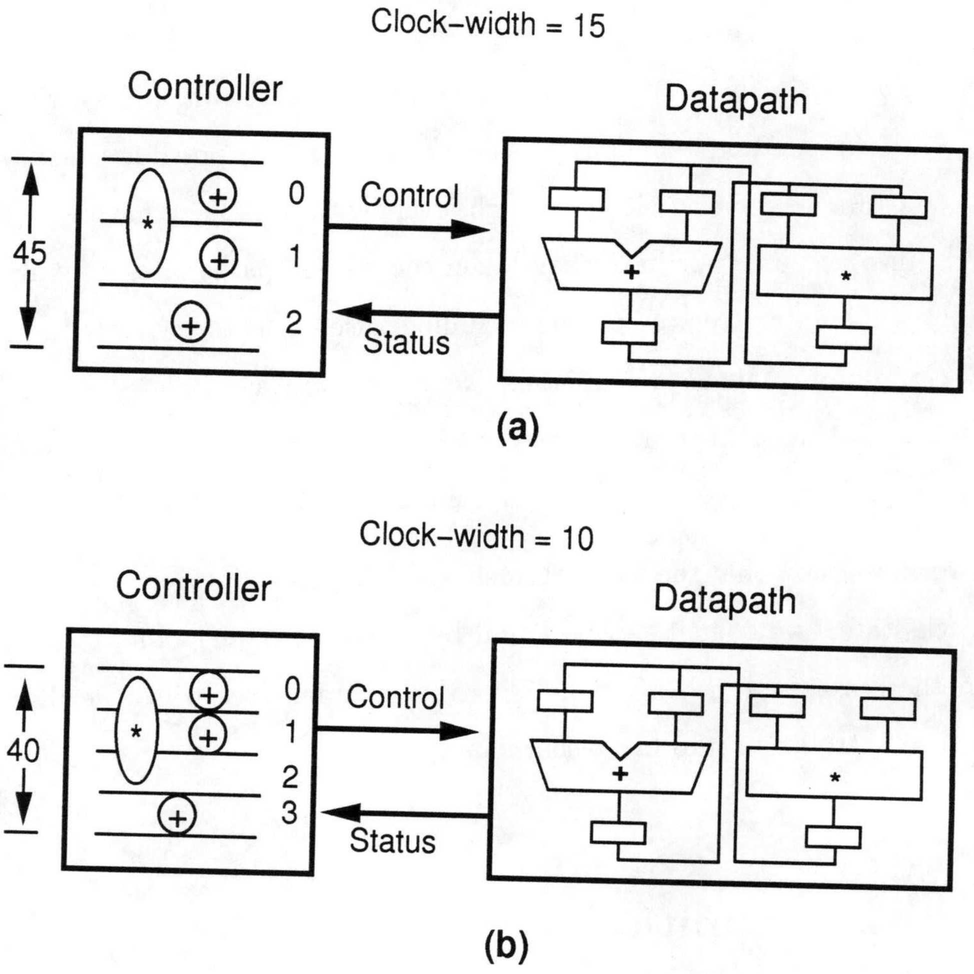

Figure 1: Reclocking of controller (a) Initial design (b) Final design

Given a datapath and an initial schedule (FSM of the controller), *reclocking* finds a new clock-width that minimizes the execution time. This could mean that some of the data transfers that were scheduled to execute in one clock cycle may now take multiple clock cycles to execute. Alternatively, in another design, some functional units which took multiple clock cycles can now take just one clock cycle. Figure 1(b) shows the design after reclocking. The "*" function now takes 3 clock cycles. The clock-width has been reduced from 15ns to 10ns, which in turn leads to the reduction of the execution time (45ns to 40ns). Note that only the structure of the controller has changed. Neither the datapath nor the connectivity between the datapath and the controller (control and status lines) have changed. Since the datapath and the connectivity remain unaltered in reclocking, we will consider only the controller in our problem formulation and examples.

Given a scheduled behavior, the problem of reclocking is defined in terms of the set of states, the delays associated with each state, and the delays of multicycle operations. Let S be the set of states in the controller:

- S = $\{s_i | s_i$ is a state in a controller.$\}$

Each state s_i activates a set of data transfers. Each data transfer incurs a delay, given by the maximum delay value for various paths that are activated for this data transfer. We define *state-delay*, d_i, as the maximum delay for all the data transfers activated in s_i. Note that for reclocking purposes, we need to consider only the maximum delay in each state, that is, *state-delay*. For example, the controller in Figure 1(a) has three state-delays d_0, d_1 and d_2, one for each state. The state-delay d_0 (for s_0) is the sum of the delays involved in moving data from registers to adder inputs, performing "+" operation and storing the result back into the register.

We also define, MD, the set of delays associated with multicycle operations:

- $MD = \{md_i | md_i$ is the delay of a multicycle operation.$\}$

Note that md_i is not associated with a specific state of the controller. For the controller shown in Figure 1(a), we have only one multicycle delay md_0 associated with the "*" operation.

The *Execution time* (ET) for a design is given by the product of the number of clock cycles(NC) required to perform the intended behavior of the design and the clock-width(CW):

$$ET = CW * NC$$

Given a controller with set of states S and a set of state-delays, along with a set of multicycle delays, *reclocking* first finds the optimum clock-width with minimum execution time for the design. It then reschedules the controller in order to fit the new clock-width.

3.1 Determination of t_{min}

An optimal clock-width is the one that leads to minimum execution time by reducing the slack in clock-utilization. It would seem that the greatest common denominator (GCD) of state-delays should provide an optimal clock-width, as it reduces the slack to zero. However, GCD of state-delays could result in very small clock-width. In a realistic problem domain, we can not use an infinitely small clock-width. Let t_{min} be the minimum value of clock-width that can be used. Thus, the *optimal* clock-width, $t_{optimal}$, is a clock-width that is greater than or equal to t_{min} and leads to minimum execution time.

There are two factors that determine the value of t_{min}. The first factor is the minimum clocking frequency of component libraries. Each component library gives a maximum operating frequency below which the synchronous components must be switched.

The second factor is illustrated in figure 2. The straight lines indicate the initial clock, the initial availability of data at the output of register A, and the control line $B.load$ which loads the value of register A into B. The dotted lines indicate the new clock, and two possible loadings of the data via the control line $B.load$, given as $B.load(1)$ and $B.load(2)$.

In $B.load(1)$, the control signal is active during all the clock cycles into which the initial state was partitioned. In $B.load(2)$, the control signal is only active during the last cycle of the clocks into which the initial description was partitioned. If $B.load(1)$ is to be considered, then

$$t_{min} \geq (C_{pd} + I_{st}(B))/n \qquad (1)$$

$$t_{min} \geq D_{st}(B)/n \qquad (2)$$

where C_{pd} is the controller propagational delay, $I_{st}(B)$ is the load (control) setup time of the register B, $D_{st}(B)$ is the data setuptime of register B, and n is the number of states into

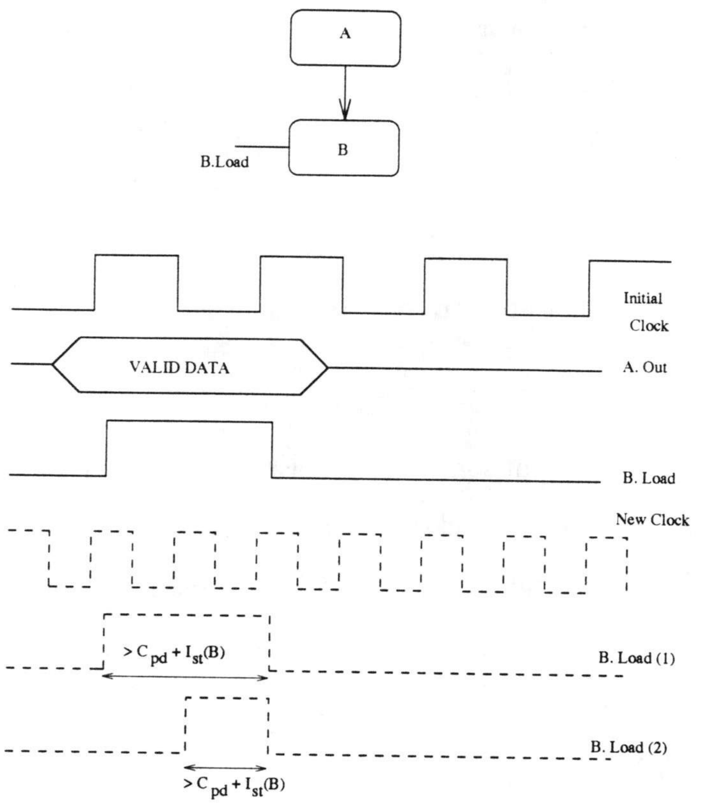

Figure 2: Old and new clocks and controller timing

which the state in the initial description has been partitioned. If $B.load(2)$ is considered, then

$$t_{min} \geq (C_{pd} + I_{st}(B)) \tag{3}$$

$$t_{min} \geq D_{st}(B)/n \tag{4}$$

The control signal $B.load(1)$ will almost always yield a smaller t_{min}, which is desirable, in order to find a good solution. However, if there are storage units which change value with each clock cycle (when control signal is enabled), then the control signal $B.load(2)$ has to be used in order to maintain correct values in storage units such as counters, shift-registers, and registers within pipelined components.

3.2 Assumptions

In this work, we make the following assumptions:

- The rescheduling of the controller does not alter the size and therefore the delay of the controller appreciably;
- The datapth – controller connectivity length remains the same after controller rescheduling;
- If the behavior contains non-straight line code, an execution trace (or branch probabilities) are given.

In the next section, we present some results for reclocking. We use these results to develop an algorithm to find the optimal clock-width and reschedule a given design. For the rest of the report, we use the term "code" and "controller" interchangeably.

4 Reclocking

We first present results for simple straight-line code without multicycle operations. Then we extend this result to capture more realistic controllers that encompass multicycle operations as well as non-straight-line code.

4.1 Straight-line code with no multicycling

In straight line code, we don't have branches; the control flows sequentially through all the states of the code. Thus the execution time (ET) for straight line code is given by:

$$ET = CW * NS$$

where NS is the number of states in the controller.

Also, without multicycle operations, only delays that are to be considered are state-delays. The following theorem sets the basis of the reclocking for simple code. In this theorem, integer divisions of a state-delay d_i are given by: $\{d_i/j \mid \text{j}=1,2,3,\ldots\}$.

Theorem 1 *In a straight-line controller with single–state operations the optimal clock-width, with minimum execution time, will lie on one of the integer divisions of one of the state-delays or on the minimum clock-width,* t_{min}.

Proof Let us assume that the optimal clock-width, $t_{optimal}$, is not equal to t_{min} or integer divisions of one of the state-delays. Let us consider another clock-width, t_{better}, which is smaller than $t_{optimal}$ by an infinitesimally small value δt.

$$t_{better} = t_{optimal} - \delta t$$

Since $t_{optimal}$ is not equal to t_{min}, t_{better} is a valid clock-width. Also, since $t_{optimal}$ does not lie on one of the integer divisions of any of the state-delays, there is a slack in clock utilization for each state s_i. Thus, with t_{better} which is smaller than $t_{optimal}$ by an infinitely small value, each of the critical operations will require same number of states as is required with $t_{optimal}$. Note that if t_{better} is equal to an integer division of a state-delay d_i, then the above statement is not true. This is because in the schedule with t_{better}, the critical operation in state s_i would require one extra state as compared to the schedule with $t_{optimal}$. Thus, the schedule with t_{better} requires the same number of states as is required by the schedule with $t_{optimal}$. Hence, t_{better} reduces the execution time as compared to $t_{optimal}$. This contradicts our assumption that $t_{optimal}$ is the optimal clock-width. Hence, the optimal clock-width will lie on the integer divisions of one of the state-delays or on t_{min}.

4.2 Straight-line code with multicycle operations

As previously mentioned, a multicycle operation requires more than one state for its completion. With the introduction of the multicycle operations, we need to consider both the state-delays and the set of multicycle delays, MD. Given a straight–line code with multicycle and unicycle operations, we have to reschedule the controller, given a minimal clock-width t_{min} such that the execution time is minimized.

Corollary 2 *In a straight-line code with single-state and multi-cycle operations the optimal clock-width with minimal execution time will lie on an integer division of one of the state-delays, on a multicycle delay, or on* t_{min}.

Proof: A simple extension of the proof for Theorem 1 will suffice. Let us assume that the optimal clock-width, $t_{optimal}$, is not equal to an integer division of a state-delay, a multicycle delay or t_{min}. As before, let t_{better} be given by:

$$t_{better} = t_{optimal} - \delta t$$

Since $t_{optimal}$ does not lie on one of the integer divisions of any of the multicycle delays, there is a slack in clock utilization for each multicycle operation. Thus, with t_{better} which is smaller than $t_{optimal}$ by an infinitely small value, each of the critical operations including the multicycle operations will require the same number of states as is required with $t_{optimal}$. Thus, the schedule with t_{better} requires the same number of states as is required by the schedule with $t_{optimal}$. Hence, t_{better} reduces the execution time as compared to $t_{optimal}$. This contradicts our assumption that $t_{optimal}$ is the optimal clock-width. Hence, the optimal clock-width will lie on the integer divisions of one of the state-delays, multicycle delays or on t_{min}.

The above proof establishes sufficient conditions for the optimal clock-width. The following example shows that the multicycle delays are a necessary requirement for getting the optimal solution.

In Figure 3, if the multicycle delay of 110ns is ignored, the best schedule will have a clock-width of 19.67ns (59/3). The execution time would then be 157.36ns. If on the other hand, the multicycle operation is taken into account, the best schedule will have a clock-width of 10ns, with an execution time of 150ns[2].

[2]We have assumed a t_{min} of 6ns

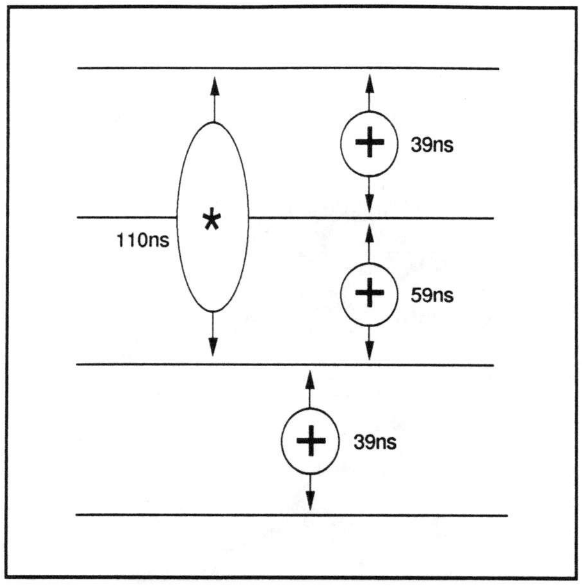

Figure 3: An example with a multicycle operation

4.3 General code

Finally, we consider an unrestricted controller. In general, a controller could have branches and loops. We assume that a static trace of the schedule is given. That is, we know, in advance, how many times each of the states are executed. Given an unrestricted code with unicycle as well as multicycle operations, a trace of the execution (states $s_1, s_2, \cdots s_n$ occurring $i_1, i_2 \cdots i_n$ times respectively), and a minimal clock width t_{min}, we need to find the optimal clock-width and then reschedule the controller such that the execution time is minimized.

Corollary 3 *In an unrestricted code with static trace, unicycle and multicycle operations the optimal clock width will lie on one of the integer divisions of state-delays or multicycle delays or t_{min}.*

Proof : From the static trace of the unrestricted code we know that each state occurs an integer number of times. Since each of the states must occur an integer number of times, the unrestricted code can be "unrolled" to make it a straight-line code. Thus, Corollary 3 reduces to Corollary 2. This completes the proof.

4.4 Algorithm for reclocking

Algorithm 4.1 : Reclocking for straight-line code

```
INPUT: A controller($CU_i$) and $t_{min}$
OUTPUT: Rescheduled controller($CU_o$) with optimal clock-width
1 $D$ = state-delay($CU_i$);
2 $et_{min} = \infty$;
3 foreach $d_i \in D$ loop
    3.1 $j = 1$;
    3.2 while $d_i/j > t_{min}$ loop
        3.2.1 $t = d_i/j$;
        3.2.2 $et$ =EXECUTION_TIME($t, CU_i$);
        3.2.3 if ($et < et_{min}$) then $cw = t$;
        3.2.4 increment j;
    3.3 end loop
4 end loop
5 $CU_o$ = RESCHEDULE($CU_i, cw$);
6 Return $CU_o$;
```

Now we incorporate the above results into an algorithm that finds the optimal clock-width for a controller and then reschedules it to fit the optimal clock-width. Algorithm 4.1 lists the steps for reclocking of a controller. This algorithm takes as input a controller specification in terms of states, state-delays, multicycle delays and minimum clock-width, t_{min}. The first section of the algorithm extracts the state-delays and the multicycle delays. The second section of the algorithm finds new clock-widths by dividing each of the delays by incremental integers until a specified low clock-width is achieved. For each of these clock-widths, the algorithm computes the execution time. The clock-width yielding the minimum execution time is chosen as the optimal clock-width. The final section of the algorithm reschedules the operations with the optimal clock-width.

In Algorithm 4.1, CU_i, CU_o and t_{min} refer to the input controller, output controller and the minimum clock-width respectively. For a given clock-width t and input controller, function EXECUTION_TIME(t, CU_i) finds the execution time. Note that in order to find the execution time, the controller is to be rescheduled for the given clock-width t. Also, for non straight-line code, the trace counts of CU_i have to be converted to the trace counts of CU_o. Function RESCHEDULE(CU_i, cw) reschedules the input controller CU_i for the given clock-width cw. The variables et_{min} and cw represent the minimum execution time and the current best clock-width respectively.

4.4.1 Complexity analysis

The above algorithm has a complexity of O$(n + m)^2$, where n is the number of states in the input controller and m is the number of multicycle operations. The loop at statement 3 investigates each state-delay and multicycle delay. For each delay, the algorithm tries each integer division that is greater than t_{min}. Since each of d_i, md_i, and t_{min} are finite, the loop at statement 3.2 runs for a constant number of times. The statement at 3.2.2 requires rescheduling, and the complexity of rescheduling is $O(n + m)$. Thus the complexity of the whole algorithm is $O(n + m)^2$.

In the next section we apply this algorithm on few examples and show the effectiveness of the reclocking scheme.

5 Experimental Results

In this section we present the results of our experiments on few designs from the literature. First, we demonstrate reclocking on designs with realistic component delays. Then we apply our technique on designs with physical design information such as wiring delays. Finally, we

Component	*Delay type*	*Delay value*
Register	set-up	3.3ns
Register	propagation	3.3ns
2-input mux	propagation	5.7ns
3-input mux	propagation	6.0ns
4-input mux	propagation	6.0ns
5-input mux	propagation	6.8ns
6-input mux	propagation	6.8ns
Alu	propagation	18.4ns
Multiplier	propagation	80.5ns

Table 1: Delay values for 32-bit components from VDP300 library

present experimental results that demonstrate the bit-width and library migration capability of our approach.

5.1 Designs with realistic delays

We applied our methodology on two designs from the literature and two designs generated by high-level synthesis tool[RaGa91]. In this experiment, we demonstrate that reclocking with realistic component delays can make substantial improvements in design performance. We have used VTI [VTI91] as the target library for these examples. Table 1 lists the delay values for the relevant components: a register, multiplexers, a multiplier and an ALU. For the examples in this section, we assume that the minimal clock width (t_{min}) is provided by the user and that it is $20ns$. Also, since the trace of execution for the non-straight line designs is not given, we approximate the execution-time (ET) by the product of clock-width (CW) and number of states (NS) in the design:

$$ET = CW * NS$$

First, we walk through a simple example design that does not have multicycle units. Figure 5(a) shows a scheduled design for a behavior that solves a second order differential equation[PKGr86]. This schedule has been generated by [RaGa91] based on an allocation of two multipliers and two ALUs that can perform addition, subtraction and comparison. Note

State	*Worst case path*	*Delay value*
0	control,mux,reg	10.3+5.7+3.3 = 19.3ns
1	reg,control,mux,alu,reg	3.3+10.3+6.0+18.4+3.3 = 41.3ns
2	reg,control,mux,mult,reg	3.3+10.3+6.0+80.5+3.3 = 103.4ns
3	reg,control,mux,mult,reg	3.3+10.3+6.0+80.5+3.3 = 103.4ns
4	reg,control,mux,mult,reg	3.3+10.3+6.0+80.5+3.3 = 103.4ns
5	reg,control,mux,alu,mux,reg	3.3+10.3+6.0+18.4+5.7+3.3 = 47.0ns
6	reg,control,mux,alu,reg	3.3+10.3+6.0+18.4+3.3 = 41.3ns

Table 2: Maximum state delays for HAL example

d_i	n_i	*Clock-width*	*Num-clocks*	*Execution time*
41.3	1	41.3	14	578.2
41.3	2	20.7	26	538.2
47.0	1	47	13	611.0
47.0	2	23.5	22	517.0
103.4	1	103.4	7	723.8
103.4	2	51.7	10	517.0
103.4	3	34.5	16	552.0
103.4	4	25.6	19	486.4
103.4	5	20.7	23	**476.1**

Table 3: Various clock-widths for HAL example

that there are 7 states in the initial schedule. In this design, clock-width (worst case state-delay) is given by the sum of the register propagation delay, controller delay, multiplexer delay, multiplier delay and register set-up delay. Using the delay values provided in Table 1, we get $3.3 + 10.3 + 6.0 + 80.5 + 3.3 = 103.4ns$ as clock-width. Thus the execution time is $103.4 * 7 = 723.8ns$.

Now we apply our clock-width determination algorithm on this schedule. Table 2 lists maximum delays for each state. We observe that we have only three distinct delay values (d_i): $41.3ns$, $47.0ns$ and $103.4ns$. Table 3 lists each clock-width and the corresponding execution time that is considered by the clock-width determination algorithm. For each delay (d_i), we consider various fractions till we hit t_{min}. From Table 3, we observe that the best performance is achieved with clock-width = $20.7ns$. The execution time with this

Designs	Initial schedule			Final schedule			Improvement
	CW	NC	ET	CW	NC	ET	
HAL (1mult+1alu)	52.1	17	885.7	26.1	33	861.3	2.75%
Elliptic filter (1mult+2add)	52.1	21	1094.1	21.6	50	1080.0	1.29%
Elliptic filter (2mult+2add)	52.1	19	982.3	21.6	42	907.2	7.64%

CW : clock-width(ns) NC : num of states ET : execution time(ns)

Figure 4: Performance improvement for designs from HLS

clock-width is $476.1ns$. Thus we get

$$\frac{723.8 - 476.1}{723.8} * 100 = 34.2\%$$

improvement over the initial schedule. The final schedule is shown in Figure 5(b). Note that the datapath has not changed at all. Some of the operations have been multicycled, but their connectivity is untouched. Only the controller has to be modified.

Figure 4 shows initial and final schedules for few more examples. The initial schedule in this figure refers to the schedule from literature[WuCG91] or generated by synthesis tools. These schedules include a few multicycle operations. The table in Figure 4 shows percentage improvement in performance for the final schedule as compared to the initial schedule for three examples from the literature or generated by HLS tools. The first design is a different schedule for the above mentioned HAL example [PKGr86] that uses only one multicycle multiplier. The second example is a schedule for an elliptic filter [WuCG91] using a single multicycle multiplier whereas the third example is a different schedule for the elliptic filter with two multipliers.

For each design, we report clock-width(CW), number of states(NS) and execution time(ET) for the initial and final schedule after retiming. Clock-width and execution time are in

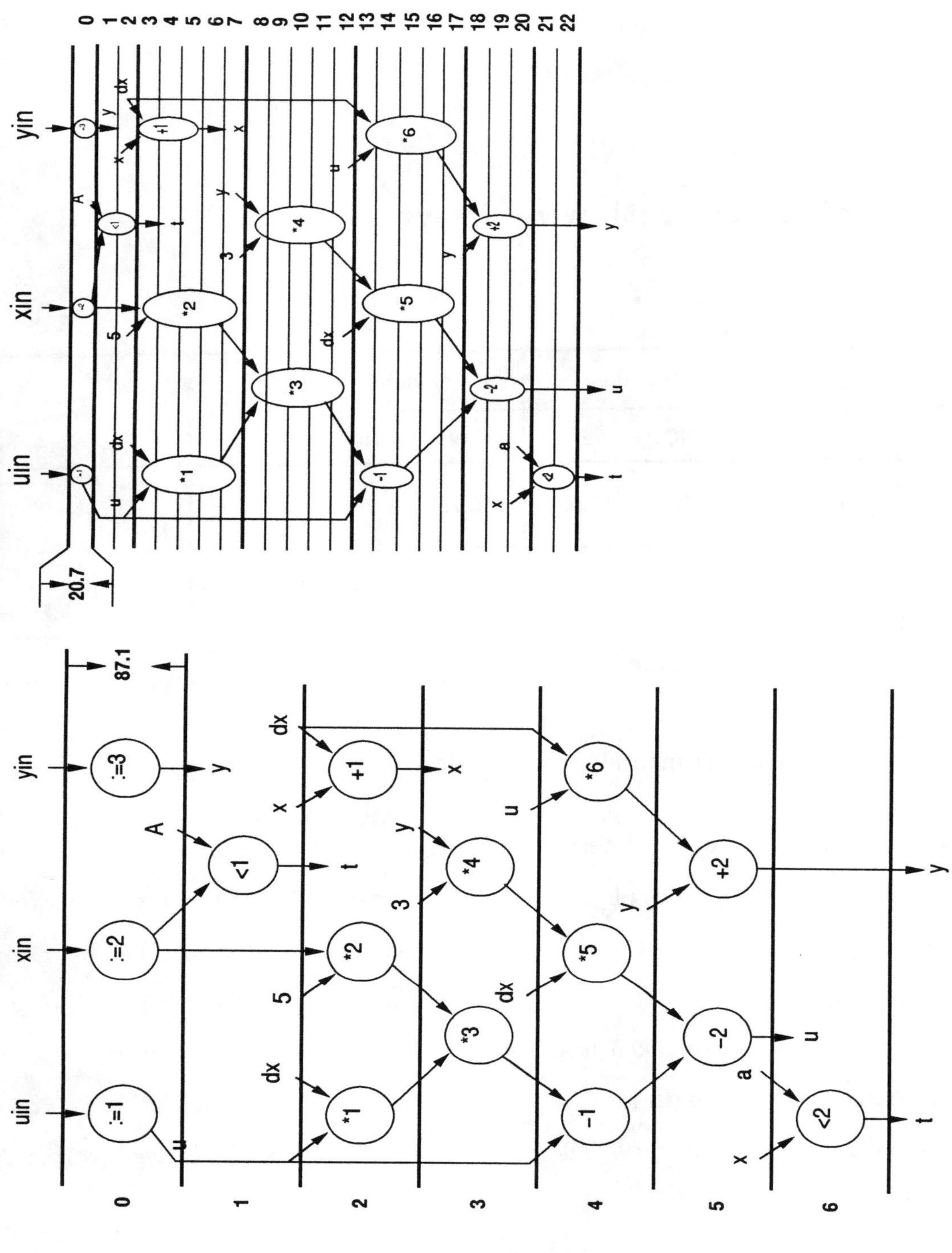

Figure 5: Hal example with an allocation of 2mult+2alu (a) Initial schedule from HLS (b) Final schedule from our approach

nanoseconds. We also report the percentage improvement in execution time for the final schedule as compared to the initial schedule. We observe that substantial improvements in execution time can be achieved by retiming the controller. The improvements for the three examples are 2.65%, 1.29% and 7.64%. Note that we have been able to achieve such improvements in performance by keeping the datapath unchanged.

5.2 Back-annotation with wire delays

Designs	Initial schedule			Final schedule			Improvement
	CW	NC	ET	CW	NC	ET	
Elliptic filter (1mult+1add)	24.32	70	1702.4	56.75	29	1645.7	**3.88%**
Elliptic filter (2mult+2add)	24.54	50	1227.0	24.54	50	1227.0	**0.00%**

CW : clock–width(ns) NC : num of states ET : execution time(ns)

Figure 6: Experimental results for designs with wiring delays

We now apply the clock-width determination algorithm to designs, taking into account the wiring delays. Recall that since physical design information such as wiring delays are not available during synthesis, the clock-width and the schedule generated may not be optimal. In this experiment, we demonstrate how reclocking can improve the performance of the designs when wiring delays are taken into account. We considered two designs for elliptic filter and estimated the wire-length for each net in the design [WuCG91]. The estimation was based on the 3.0 micron VTI library. Then we recalculated the state delays incorporating these wire delays.

In order to perform a comparative study, we first calculated the clock-width with state-delays that do not consider wire delays. Then, we find the clock-width and the schedule for the state-delay that incorporates wire-delays. Figure 6 describes experimental results for two designs. In this figure, the initial schedule refers to the optimized schedule without considering wire delays; the final schedule refers to one with wire delays. We observe that we have been able to improve the performance for at least one design with our reclocking technique.

Note that this experiment is based on a 3.0 micron technology. In this technology, wire delays are significantly smaller as compared to the component delays. For example, in our experiments, wire delays are of the order of 3.0 ns as compared to 150 ns delay of multiplier. However, as we move to sub-micron technologies, wire delays become major factors. Hence, designs targeted to sub-micron technologies must back-annotate wire delay; our reclocking scheme describes a technique to accomplish this and achieves considerable improvements in performance.

5.3 Bit-width migration

Next we discuss experimental results for bit-width migration. In bit-width migration, the design has been generated for a particular bit-width and is now being reused (with the same schedule) for a different bit-width. An increase in the bit-width increases the delay in some components while keeping it constant in others. If the same schedule is used as before, we would get sub-optimal performance; reclocking can improve the performance of the new design.

Figure 7 presents experimental results that compare designs with and without reclocking for migrating 16-bit designs to 32-bits. In this experiment, we first calculate the state delays using delay values for 16-bit components (Table 4) from the VTI library. Using these delays, we find an optimal schedule for the 16-bit design. Next, this design is upgraded to 32-bits without reclocking, i.e., without changing the schedule. The initial schedule in Figure 7 refers to this design. The clock-width in this column is given by the minimum clock-width that satisfies this schedule for 32-bit components. The final schedule is achieved by reclocking the controller based on delays of 32-bit components from the VTI library. From the table in Figure 7 we observe that reclocked designs are better than ones without reclocking in

Component	*Delay type*	*Delay value*
Register	set-up	3.3ns
Register	propagation	3.3ns
2-input mux	propagation	5.7ns
3-input mux	propagation	6.0ns
4-input mux	propagation	6.0ns
5-input mux	propagation	6.8ns
6-input mux	propagation	6.8ns
Alu	propagation	15.4ns
Multiplier	propagation	43.7ns

Table 4: Delay values for 16-bit components from VDP300 library

Designs	**Initial schedule**			**Final schedule**			**Improvement**
	CW	NC	ET	CW	NC	ET	
HAL (1mult+1alu)	34.7	27	936.9	26.1	33	861.3	**8.07%**
HAL (2mult+2alu)	34.5	16	552.0	20.7	23	476.1	**13.75%**
Elliptic filter (1mult+2add)	52.1	21	1094.1	21.6	50	1080.0	**1.29%**
Elliptic filter (2mult+2add)	51.7	19	982.3	21.6	42	907.2	**7.64%**

CW : clock–width(ns)   NC : num of states   ET : execution time(ns)

Figure 7: Experimental results for migrating designs across bit-width

Component	*Delay type*	*Delay value*
Register	set-up	0.9ns
Register	propagation	2.8ns
2-input mux	propagation	2.3ns
3-input mux	propagation	3.4ns
4-input mux	propagation	3.4ns
Alu	propagation	13.1ns
Multiplier	propagation	27.0ns

Table 5: Delay values for 16-bit components from Cascade library

performance by as much as 13.75%.

5.4 Library migration

Designs	**Initial schedule**			**Final schedule**			**Improvement**
	CW	NC	ET	CW	NC	ET	
HAL (1mult+1alu)	16.4	27	442.8	14.7	29	426.3	**3.73%**
HAL (2mult+2alu)	13.7	17	232.9	13.7	17	13.7	**0%**
Elliptic filter (1mult+2add)	29.4	21	617.4	14.7	40	588.0	**4.76%**
Elliptic filter (2mult+2add)	29.4	19	558.6	14.7	36	529.2	**5.26%**

CW : clock-width(ns) NC : num of states ET : execution time(ns)

Figure 8: Experimental results for migrating designs across technology library

We also applied our technique for porting designs from one library onto another library. We considered four designs that have been optimized for the 16-bit VTI[VTI91] library and retargetted them onto the 16-bit Cascade[Casc92] library. Table 5 shows delays for 16-bit components from the Cascade library.

Our approach is similar to the bit-width migration process described earlier. We first calculate the state delays with the component delays shown in Table 5 and then run our clock-determination/retiming algorithms to find the optimum clock-width and schedule.

Figure 8 shows the percentage improvement in performance for the four designs. Once again, the clock-width in the initial schedule column is given by the minimum clock-width that would satisfy the given schedule. The final schedule refers to the optimized controller for the 16-bit Cascade components. We observe that the improvement in performance is in the range of 0-5.26%.

Note that as we migrate designs across bit-widths or technology libraries, the component delays do not change in same proportion. For example, register and multiplexer delays remains invariant across bit-width, whereas ALU and Multiplier delays change significantly. Furthermore, the interconnect delays can change based on the topology and layout style. Thus the clock-width which suggests the minimal execution time design for a specific bit-width and technology library may not (and in fact in most cases does not) represent the optimal clock for performance when the design is ported to a different bit-width or/and library. This is why we have been able to improve the performance of the design by retiming the controller. We have been able to do so with minor or no modification to the datapath. In summary, our technique not only makes the task of retargetting feasible, but in most cases improves the performance of the design by reducing the execution time.

6 Summary

In this report we have described *reclocking*, a powerful post–synthesis approach for performance improvement by minimizing the total execution time. We can accommodate designs created by a high level synthesis system and back annotate the wire delays to the design. Having extracted the delays we are able to resynthesize the controller to improve performance without altering the datapath.

We have presented and proven some interesting results for finding the optimal clock-width for a RT-level design. These results significantly prune the search space for finding the optimal clock-width. An algorithm for reclocking has been presented based on the above results.

Our approach is versatile and can be applied not only for wire delay consideration, but also for bit–width migration, library migration and for feature size migration supporting the philosophy of design reuse. Experimental results show that with reclocking, the performance of the input designs can be improved by as much as 36%.

7 Acknowledgements

This research was supported by SRC contract #93-DJ-146. We thank Viraphol Chaiyakul for providing the RT-level designs and wiring information for the elliptic filter examples.

References

[BDBr94] S. Bhattacharya, S. Dey and F. Brglez, "Clock Period Optimization During Resource Sharing and Assignment," *Proc. Design Automation Conference,* pp. 195-200, June 1994.

[CaPl92] R. Camposano and P. G. Ploger, "Retiming and High Level Synthesis," *Proceedings of the Sixth High Level Synthesis Workshop,* pp191–201, 1992

[Casc92] "Cascade Design Automation Databook," *Cascade Design Automation, Bellevue, WA,* 1992.

[DeM91] G. DeMicheli, "Synchronous Logic Synthesis: Algorithms for Cycle–Time Minimization," *IEEE Transactions on Computer–Aided Designs,* pp63–73, 1991.

[GDWL92] D. Gajski, N. Dutt, A. Wu and S. Lin, "High-Level Synthesis: Introduction to Chip and System Design," *Kluwer Academic Publishers* 1992.

[KaGa91] K. Kanehara and D. Gajski, "Pipelining of Register Transfer Netlists," *Technical Report 91-12, University of California at Irvine* 1991.

[LeSa88] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry," *Laboratory for Computer Science, MIT,* 1988

[PKGr86] P. G. Paulin, J. P. Knight and E. F. Girczyc, "HAL: A Multi-paradigm Approach To Automatic Data Path Synthesis," *25 Years of Electronic Design Automation,* pp587-594, 1986.

[MSBS91] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques," *IEEE Transactions on Computer-Aided Design,* pp74–84, 1991

[NaGa92] S. Narayan and D. D. Gajski, "System Clock Estimation based on Clock Slack Minimization," *European Design Automation Conference (EuroDAC),* September 1992.

[RaGa91] L. Ramachandran and D. Gajski, "An Algorithm for Component Selection in Performance Optimized Scheduling," *IEEE International Conference on Computer-Aided Design,* pp92-95, November 1991.

[Tosh90] "Toshiba ASIC Gate Array Library," *Toshiba Corporation, Tokyo, Japan,* 1990.

[VTI91] "VDP300 CMOS Datapath Library," *VLSI Technology, Inc., San Hose, California,* November 1991.

[WuCG91] A. C. Wu, V. Chaiyakul and D. D. Gajski, "Layout-Area Models for High-Level Synthesis," *Proc. International Conference on Computer-Aided Design,* pp. 34-37, November 1991.

[ZaGa88] N. Vander Zanden and D. D. Gajski, "MILO: A Microarchitecture and Logic Optimizer," *Proc. Design Automation Conference,* pp. 403-408, June 1988.