Reclocking for High Level Synthesis

Pradip Jha
Information and Computer Science
University of California, Irvine
CA 92717, USA

Sri Parameswaran
Dept of ECE
The University of Queensland
QLD 4072, Australia

Nikil Dutt
Information and Computer Science
University of California, Irvine
CA 92717, USA

Abstract— In this paper we describe, a powerful post–synthesis approach called *reclocking*, for performance improvement by minimizing the total execution time. By back annotating the wire delays of designs created by a high level synthesis system, and then finding an optimal clockwidth, we resynthesize the controller to improve performance without altering the datapath. *Reclocking* 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 34%.

I. Introduction

High-Level Synthesis (HLS) 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 re target 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 re implemented 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.

We demonstrate the effectiveness of our approach by applying *reclocking* to various design scenarios using four sets of experiments. In the first experiment we apply reclocking to the outputs of HLS and show its effectiveness. The second experiment applies reclocking to an RT level design back-annotated with wiring delays obtained from physical design. The last two experiments establish the design reuse capability of our approach by applying reclocking in migrating designs to different bit-widths and different libraries respectively. These experiments show improvements in performance by as much as 34% by applying our reclocking approach to HLS benchmarks.

The rest of this paper is organized as follows. Section II describes related work. Section III defines the problem of *reclocking*, given an initial schedule and datapath delays. Section IV describes techniques to find the optimal clock-width for different types of input behavior and presents an algorithm for reclocking. Section V demonstrates the efficacy of our approach by applying reclocking on standard HLS benchmarks for different design scenarios. Section

VI concludes with a summary.

II. 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[6]. 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 [8] and De Micheli [4] 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 [2] describe the application of retiming to high-level synthesis. [1] describes an approach that reduces the clock-width at the resource sharing and assignment phases of synthesis. [14] 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 [9] 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 [9]. Using this optimal clock-width we then proceed to reschedule the controller to improve the design's performance.

III. 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 [5]. The datapath consists of a netlist of RT level components such as ALUs, registers, multiplexers, 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.

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:

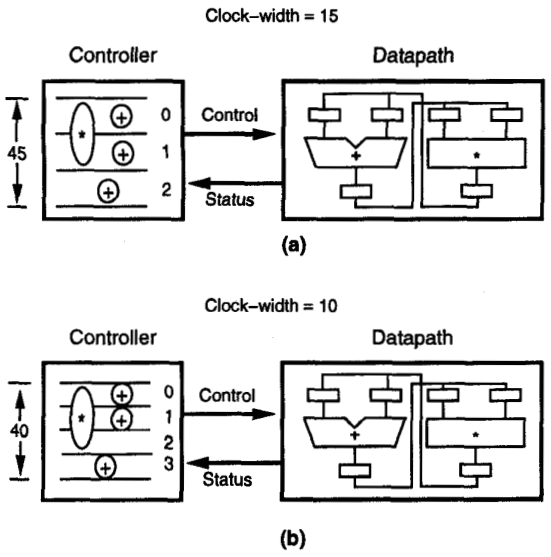

Fig. 1. Reclocking of controller (a) Initial design (b) Final design

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

Note that md_i is not associated with a specific state of the controller. For example, the controller shown in Figure 1(a) has 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. The minimum clock-width is determined by the largest of the following factors: the maximum clocking frequency of component libraries, data setup time of registers and the controller propagation time and the control setup time of the registers.

A. Assumptions

In this work, we make three assumptions: one, the rescheduling of the controller does not alter the size and therefore the delay of the controller appreciably; two the datapath – controller connectivity length remains the same after controller rescheduling; and three 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 paper, we use the term "code" and "controller" interchangeably.

IV. Reclocking

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.

A. Straight-line code with unicycle and 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.

Theorem 1 *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, multicycle delays, or on t_{min}.*

Proof: 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}. Let us consider another clock-width, t_{better}, which is smaller than $t_{optimal}$ by an infinitesimally small value δt. 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, state delays or on t_{min}, there is a slack in clock utilization for each 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. Consideration of multicycle delays are necessary to get the optimal clockwidth.

B. General code

Next, 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 2 *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 2 reduces to Theorem 1. This completes the proof.

C. Algorithm for reclocking

Algorithm IV.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 = ∞;
3 foreach d_i ∈ 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
         3.2.3.1 cw = t;
         3.2.3.2 et_min = et;
      3.2.4 end if
      3.2.5 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 IV.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 IV.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. The algorithm has a complexity of $O(n+m)^2c$, where n is the number of states in the input controller and m is the number of multicycle operations. The iteration count c represents the maximum number of integer divisions that has to be considered for a state-delay or a multicycle delay.

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 I
DELAY VALUES FOR 32-BIT COMPONENTS FROM VDP300 LIBRARY

V. EXPERIMENTAL RESULTS

In this section we present the results of our experiments on some HLS benchmarks. 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 present experimental results that demonstrate the bit-width and library migration capability of our approach.

A. Designs with realistic delays

We applied our methodology on two designs from the literature and two designs generated by high-level synthesis tools[10]. We have used VTI [12] as the target library for these examples. Table I 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 assume the code to be straight line code.

Table II shows initial and final schedules for the four examples. The initial schedule in this figure refers to the schedule from literature[13][7] or generated by synthesis tools. These schedules include a few multicycle operations. The table in Table II shows percentage improvement in performance.

Designs	Initial schedule		Final schedule		Improv.
	NC	ET(ns)	NC	ET(ns)	
HAL (2mult+2alu)	7	723.8	23	476.1	34.20%
HAL (1mult+1alu)	17	885.7	33	861.3	2.75%
Elliptic filter (1mult+2add)	21	1094.1	50	1080.0	1.29%
Elliptic filter (2mult+2add)	19	982.3	42	907.2	7.64%

TABLE II
PERFORMANCE IMPROVEMENT FOR DESIGNS FROM HLS

Designs	Initial schedule		Final schedule		Improv.
	NC	ET(ns)	NC	ET(ns)	
Elliptic filter (1mult+1add)	70	1702.4	29	1645.7	3.88%
Elliptic filter (2mult+2add)	50	1227.0	50	1227.0	0.00%

TABLE III
EXPERIMENTAL RESULTS FOR DESIGNS WITH WIRING DELAYS

For each design, we report number of states(NS) and execution time(ET) for the initial and final schedule after reclocking. We observe that substantial improvements in execution time can be achieved by reclocking the controller. The improvements for the three examples are 2.65%, 1.29% and 7.64%.

B. Back-annotation with wire delays

We now apply the reclocking algorithm to designs, taking into account wiring delays obtained from physical design. 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 [13]. 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. Table III describes experimental results for two designs.

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.

C. 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

Designs	Initial schedule		Final schedule		Improv.
	NC	ET(ns)	NC	ET(ns)	
HAL (1mult+1alu)	27	936.9	33	861.3	8.07%
HAL (2mult+2alu)	16	552.0	23	476.1	13.75%
Elliptic filter (1mult+2add)	21	1094.1	50	1080.0	1.29%
Elliptic filter (2mult+2add)	19	982.3	42	907.2	7.64%

TABLE IV
EXPERIMENTAL RESULTS FOR MIGRATING DESIGNS ACROSS BIT-WIDTH

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 V
DELAY VALUES FOR 16-BIT COMPONENTS FROM VDP300 LIBRARY

(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.

Table IV 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 V) 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 Table IV refers to this design. The final schedule is achieved by reclocking the controller based on delays of 32-bit components from the VTI library. From Table IV we observe that the reclocked designs are better than ones without reclocking in performance by as much as 13.75%.

D. Library migration

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[12] library and retargetted them onto the 16-bit Cascade[3] library. Table VI shows delays for 16-bit components from the Cascade library.

Table VII shows the percentage improvement in performance for the four designs. The initial and final schedules

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 VI
DELAY VALUES FOR 16-BIT COMPONENTS FROM CASCADE LIBRARY

Designs	Initial schedule		Final schedule		Improv.
	NC	ET(ns)	NC	ET(ns)	
HAL (1mult+1alu)	27	442.8	29	426.3	3.73%
HAL (2mult+2alu)	17	232.9	17	232.9	0%
Elliptic filter (1mult+2add)	21	617.4	40	588.0	4.76%
Elliptic filter (2mult+2add)	19	558.6	36	529.2	5.26%

TABLE VII
EXPERIMENTAL RESULTS FOR MIGRATING DESIGNS ACROSS TECHNOLOGY

refer to the VTI and Cascade libraries respectively. We observe that the improvement in performance is in the range of 0-5.26%.

VI. SUMMARY

In this paper 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 rescheduling the controller to improve performance without altering the datapath.

We described techniques for finding the optimal clock-width and proved optimality of our results under various input behavior assumptions. Furthermore, our results significantly prune the search space for finding the optimal clock-width as compared to previous approach. An algorithm for reclocking was presented based on the above results.

We ran several experiments to demonstrate applicability of our approach for back-annotation of physical design information into HLS, as well as for design reuse at the RT-level. Our experimental results show that with reclocking, the performance of the input designs can be improved by as much as 34%. Based on these experiments, we believe that our reclocking technique has tremendous applicability in linking HLS with physical design. As we approach sub-micron feature sizes, wire delays become significant, and can even approach functional unit delays; we described a concrete technique for back-annotation of these delays into the output of HLS to allow realistic modelling of physical design effects. We also believe that our techniques support the philosophy of design reuse, specifically in the areas of bit-width migration, library migration, and feature-size migration.

REFERENCES

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

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

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

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

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

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

[7] 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.

[8] 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

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

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

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

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

[13] 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.

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