Title

CHASSIS : a combined hardware selection and scheduling technique for performance driven synthesis

Authors

Ramachandran, Loganath
Gajski, Daniel D.

Publication Date

1991-02-10

Peer reviewed

CHASSIS : A Combined Hardware Selection and Scheduling technique for Performance Driven Synthesis

Loganath Ramachandran
Daniel D.Gajski

Technical Report #91-20
February 10, 1991

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

ramachan@ics.uci.edu

Abstract

This report describes a new technique that combines the *Hardware Scheduling* and *Component Selection* phases for High Level Synthesis. Our technique simultaneously selects components from a given library while it schedules the operations into different control steps. The algorithm improves previous work in scheduling because component costs and performance are considered during the scheduling process, enlarging the design search space and resulting in better optimized designs.

Contents

List of Figures

It would be ideal to perform all the synthesis subtasks including *hardware selection, scheduling, binding and allocation* simultaneously, but the complexity of these problems has made it computationally infeasible to combine them together. Hence most synthesis systems resort to performing them in some predetermined order. As a result of this ordering inefficiencies are introduced into the synthesis process. In particular, we believe that independent solutions to component selection and hardware scheduling cannot produce efficient designs. This is because some of the tradeoffs made during the component selection phase may make it impossible to share components across the various states leading to inefficient schedules. Similarly tradeoffs related to scheduling without taking into account the entire library of components, may result in faster components unnecessarily being used. This leads to increased layout area.

In order to solve the above problems we propose a new technique called *CHASSIS* . Some of the important features of *CHASSIS* include:

- It combines *component selection* and *operator scheduling* and hence provides for combined selection and scheduling tradeoffs leading to better designs.
- It provides *true component selection* capability where similar operators could have several physical implementations.
- Since *CHASSIS* is a performance driven synthesis algorithm it takes into account register and interconnect (multiplexers) delays.
- It also takes into account layout related parameters like area and actual cell delays. This results in realistic evaluation of decisions at the higher level.
- The design model can very easily accommodate wiring delays by extracting wiring delay information from a floor plan or an estimation program.

The following figure (Fig. 1) demonstrates why our technique leads to better designs. The upper half of the figure shows the general methodology currently being used in almost all scheduling systems, where it is assumed that all add operations take a fixed amount of time to complete. Thus the results indicate two adders whose delay is one clock period are being used to implement the

design. In our new technique, the scheduler works with the library of available components and the scheduling results show that the final design is implemented using one adder with one clock period delay and another slow adder with a two clock period delay. Although the number of components are the same in both cases, the *CHASSIS* results are obviously better since the slow adder is less expensive compared to the fast adder.

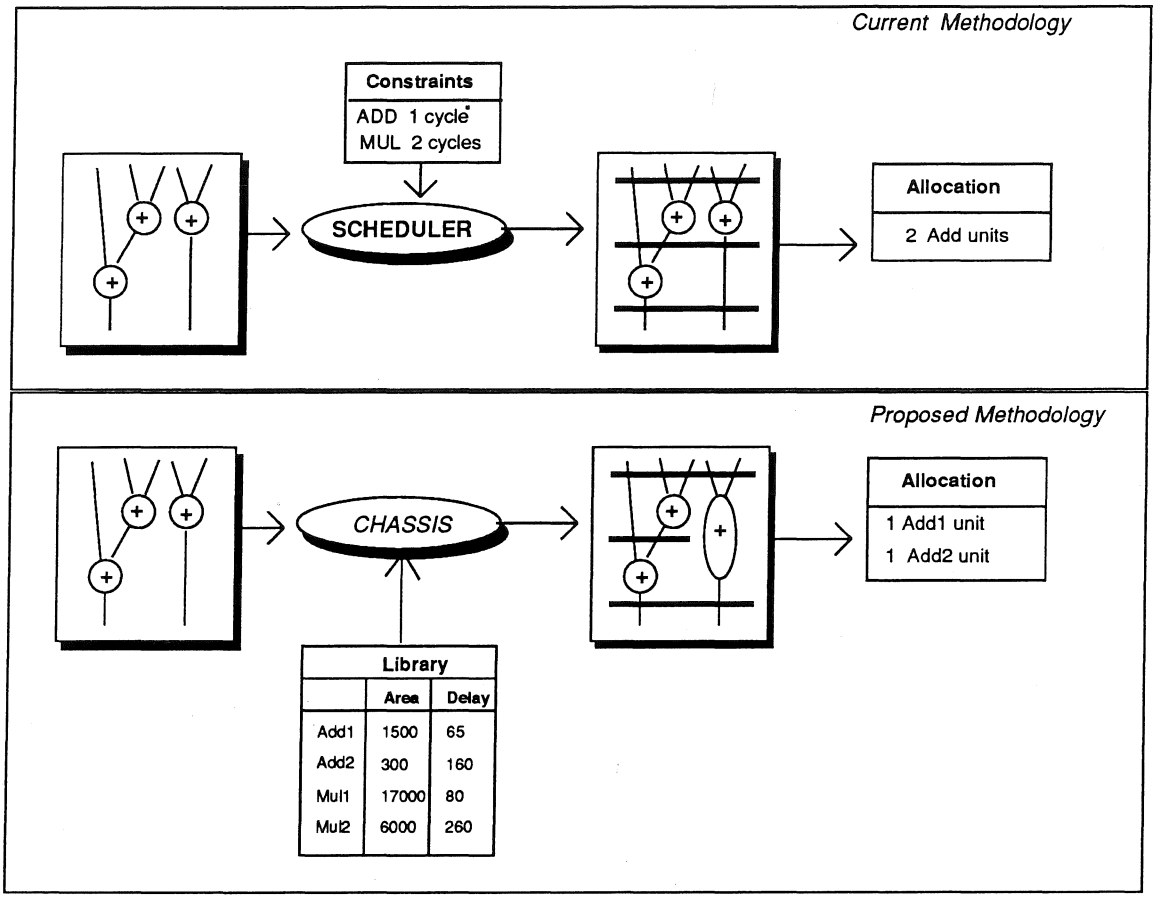


Figure 1: Current and Proposed Approaches

The rest of this report is organized as follows. In Section 2 we discuss some of the scheduling and component selection work that has been discussed in the literature and examine their strengths and weaknesses. Section 3 presents *CHASSIS* in detail. In Section 4 we provide a very simple example and step through the various important stages of *CHASSIS* . In Section 5 we present results on a

couple of standard benchmarks. In the concluding section we also discuss how our work could be extended.

2 Previous Work

We categorize the previous work mentioned in the literature into two classes based on its relevance to scheduling or its relevance to component selection.

2.1 Scheduling Related Work

We will further subdivide the work related to scheduling into two categories.

- Scheduling algorithms that work with Resource Constraints
- Scheduling algorithms that work with Performance Constraints.

2.1.1 Scheduling with Resource Constraints

In this class of algorithms the scheduler is provided with a set of components to be used during the scheduling process. For example, the scheduler could be constrained to use three adders and two multipliers to schedule a given CDFG. The algorithms attempt to maximize the utilisation of the given resources in each of the control steps.

One of the elementary scheduling algorithms that has been used widely is the ASAP scheduling approach. In this approach the operations are scheduled whenever their input datum are available (i.e as soon as all the input nodes of an operation are scheduled).

The synthesis systems that have used this algorithm include the Emerald/Facet system from CMU [2]. Another equivalent scheduling algorithm is the ALAP scheduling algorithm where all operations are scheduled as late as possible.

List scheduling algorithms are more complicated than the ASAP scheduling approach. In this technique, a list of operations that are *ready* to be scheduled is constructed. If possible, all the operations in the ready list are scheduled with the allocated hardware. If there are resource conflicts due to insufficient hardware then some of the operations are postponed based on a *priority* value assigned to each of the operations in the *ready* list.

A number of scheduling systems have used list scheduling approaches. But the systems vary in the way they assign priority to each of the operations. In SLICER [9], the operations are postponed based on a functional value called *mobility*, where *mobility* is defined as the difference between the ALAP and ASAP schedules. The mobility value is actually an indication of the number of scheduling options that are available. If the mobility of a node is 2, it implies that there are two control steps to which the node can be assigned. Essentially, if the mobility is higher the number of scheduling choices are more. Hence,when a resource crunch occurs at a particular control step, an operation with a higher mobility has greater chances of getting postponed to the next control step.

In [6] the operations are postponed based on violation of timing constraints. When scheduling operations in a particular control step, all operations that violate the minimum timing constraint are postponed. On the other hand, certain operations cannot be postponed because they would violate the maximum timing constraint, if they were scheduled in the next control step.

Force Directed List Scheduling (FDLS), as the name implies is a *list scheduling* algorithm [11]. The main difference is the evaluation of the *priority* function to determine which operations should be postponed during a resource crunch. In FDLS, a global function called *force* (which actually is an approximate measure of the concurrency in the schedule) is calculated. If the force for a particular assignment is minimal then the concurrency is maximal. Thus at any given instance the assignment that produces the lowest force is given the highest priority. FDLS looks at a more global picture than the other list scheduling approaches because *all* unscheduled operations are also considered during the force calculaton.

2.1.2 Scheduling with Performance Constraint

In this type of scheduling algorithms, the component allocation is not specified. Instead a global *performance constraint* or a *timing constraint* is provided to the scheduling algorithm. This constraint could either be in terms of the maximum delay or could be in terms of the maximum number of clock cycles within which all operations have to complete executing. The primary consideration of these algorithms is to minimize the number of functional units that they require to complete the scheduling within the timing constraint.

One of the general approaches involves determining the critical path and dividing the path into n equal time steps, where n is the number of clock cycles specified in the performance constraint. Then, sufficient functional units are allocated to ensure that all the critical path operations can be executed. The remaining operations that are not on the critical path are then scheduled.

This approach was used in MAHA [3] where the nodes on the critical path were scheduled first and the remaining nodes were allocated based on a *degree of freedom* measure. The *degree of freedom* is very similar to mobility that was mentioned for list scheduling approaches.

In Force Directed Scheduling [10] an entity called *force* is calculated before each node is scheduled in a control step. The force calculation as we discussed earlier in FDLS is a more global entity which takes into account all the nodes scheduled or unscheduled. The node that produced the minimal force is scheduled first. This ensures that the final schedule has as much concurrency as possible.

In SAM [1] the tasks of scheduling, allocation and mapping were combined into a single algorithm. They used a variation of the Force Directed Scheduling Algorithm to simultaneously schedule and map the operations onto the hardware adding new hardware when required.

All the above scheduling algorithms have one major disadvantage. It is assumed that all operators of the same type would be bound to similar functional units. For example, if there were many add operations that were being performed in the CDFG, these algorithms assumed that all the add operations would be implemented with a single type of adder (eg ripple-carry adder). They could

not work with a library of adders and decide on the most optimal allocation and utilisation of the range of adders available in the library. This made them impractical for real-life applications where typically a bigger library of pre-characterised macrocells are used to build designs. Ideally the synthesised designs should use slower components in paths where speed is not the most important criteria and use faster components especially for the critical path.

To solve this problem there are research efforts in the area of component selection. We discuss below some of the important work related to optimally selecting components from a given library.

2.2 Component Selection Related Work

One of the first solutions to the component selection problem for non-pipelined designs was proposed by Leive [7]. Here, component selection was based on a *goodness measure*. The *goodness measure* consisted of evaluating a function dependent on area, delay and power. Foo and Kobayashi [14] use a rule based approach to solve the component selection problem. However in these approaches local optimization of individual modules are performed rather than a global optimization effort.

Hafer and Parker [4] presented an unified approach to scheduling, allocation, component selection and module binding. They proposed a mixed integer linear programming approach for the combined solution of the above problems for non pipelined circuits. However, the computer time required for evaluating the solution was excessive and thus the scheme was impractical in terms of computational complexity.

In [12] a limited solution to the Module Selection problem was proposed by Jain et. al. Given a design and a module set their technique predicts the area-time tradeoff curve. They did not allow for multi-cycle operations. This restriction was however removed in a later result in MOSP [5]. But the component selection problem was not entirely solved even in MOSP. MOSP could at best be classified as a *partial selection algorithm* since it chose a single module for implementing all similar operations in the flowgraph and did not select individual modules for each operation in the flowgraph.

All these approaches had one major drawback. The scheduling and selection worked independently (except for [4] which provided an integrated solution). There was no attempt to evaluate how decisions made during one of the phases would affect performance in the other phases of synthesis. Although each of the above research efforts have achieved good results in the individual phase of synthesis we believe that independent solutions to the scheduling and selection problem cannot lead to an overall efficient design.

We now present *CHASSIS*, a technique that performs scheduling and selection in an interlinked fashion to ensure better quality of design.

3 *CHASSIS* - the Approach

This section discusses the details of *CHASSIS* which is a combined solution to the selection and scheduling problem. We first present the design model that is used by *CHASSIS*. Then we provide an overview of the algorithm and then elaborate the important steps.

The *CHASSIS* technique is based on a **probabilistic cost function** which is related to the functional area in a given design. The heart of the technique consists of simultaneously evaluating the different scheduling and selection options for each node in the dataflow graph by associating an *estimated cost* for each option. The *cost* is actually an estimate of the layout area of the functional units for a particular scheduling/selection choice. From the list of possible options, we choose one, that we expect, would lead to a minimal area.

When a node is being considered for scheduling and selection it would be impossible to determine the exact cost of the design since all the nodes in the flowgraph would not have been selected and scheduled. Hence a probabilistic estimate of the area is used for nodes that have not been scheduled while an exact area figure is available for all nodes selected and scheduled.

Before we can discuss the actual cost function and the selection and scheduling processes let us examine the *design model* that is envisaged for *CHASSIS*.

4 *CHASSIS* - The design Model

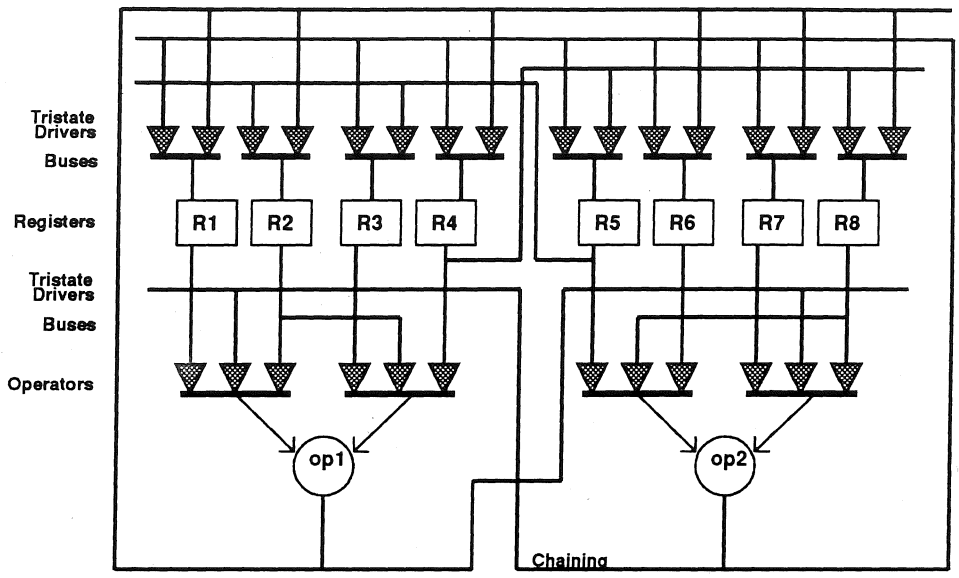


Figure 2: *CHASSIS* Design Model

The design model used by *CHASSIS* is shown in Fig. 2. A two level bus structure is assumed for the interconnection across the registers and functional units. This model allows for easy analysis of performance issues since the delay of the tristate driver can be considered to be constant with respect to the number of the tristate drivers driving a bus.

To calculate the actual delays associated with our model we consider two distinct types of register transfers.

- **Single Operation Register Transfers.** These are register transfers in which operands are read from the register, a *single* operation is performed on the operands and the results are finally stored in another register. The delays associated with this type of Register Transfers are the following.
  - Delay of the operation.

- Delay associated with two levels of tristate drivers.
- Register setup time and propogation delays.

The total delay can be expressed as :

Delay = Delay of functional unit + 2 * (delay of tristate drivers) +
Register setup time + Register propagation delay.

- **Multiple Operations Register Transfers.** The main difference in this mode of register transfers is that *multiple* operations are performed before storing the result finally in a register. As shown in the figure there is one tristate buffer between two operations.
  - Delay of 'n' operations that are chained.
  - Delay of the tristate driver between each chained operator.
  - Delay of the tristate driver before the register.
  - Register setup time and propogation delays.

The total delay in this case can be expressed as :

Delay = $\sum_{i=1}^{n} d_{opi}$ + (n+1) * (delay of tristate drivers) +
Register setup time + Register propagation delay.

With this notion of delays let us now examine the details of the algorithm. The brief sketch of the algorithm is shown next.

4.1 *CHASSIS* - The Algorithm

```
main_algorithm
begin
        perform_fine_grain_asap( all_nodes);
        perform_fine_grain_alap( all_nodes);
        calculate_span(all_nodes);
        for all nodes in the graph
        begin
                valid_choices_list = evaluate_valid_assignments();
                for all choices in the valid_choices_list
                begin
                        make_assignment();
                        calculate_span (affected_nodes);
                        update_probability_table();
                        area_estimate = estimate_probabilistic_area();
                end
                assign_best_time_step();
                select_best_component();
        end
        perform_iterative_improvement();
end
```

4.2 *CHASSIS* Details

4.2.1 Fine Grain ASAP and ALAP

After the input description is compiled into a CDFG, an ASAP and ALAP scheduling is performed on the flowgraph. The ASAP schedule of an operation indicates the earliest possible time that the operation could be scheduled. The ALAP schedule indicates the latest possible time that the operation could be scheduled.

Although ASAP and ALAP scheduling methods have been extensively used in the past, we have used a fine grain scheduling technique. it is important to distinguish between **fine grain and coarse grain scheduling** techniques. Although the implementation mechanisms for fine grain and coarse grain schedulers are very similar, they differ in the *granularity of the schedule information.*

In coarse grain scheduling it is sufficient for the scheduler to indicate the *state* in which an operation is scheduled. It is not required to pin-point the *exact time* at which the operation is scheduled. So an operation beginning at 14 ns and an operation scheduled at 94 ns could be indicated to be in control step 1 for a clock period of 100 ns. For *CHASSIS* this information will not suffice, since the exact time values are required in order to be able to select components from the library during the scheduling process. Thus *CHASSIS* uses fine-grain scheduling methods where exact timing values are available for each scheduled operation.

There are a couple of other important differences in *CHASSIS* elated to *chaining of operators* or *multi-cycle operations. Chaining* refers to a schedule in which two or more data-dependent operators could share a same control step. The first operation could begin at the state boundary, while the second operation could begin in the center of the state period. *Multicycle* operators are exactly the opposite of chained operators. They require two or more cycles to complete operation and they would cut across state boundaries.

CHASSIS does not treat *chaining* and *multiclock* operations as anything special, since an operator which is multiclock for one clock period would become a single clock element for another clock period. In other words, every operator is a multicycle operator for some clock period. The main concept used in *CHASSIS* is **multicyle paths** where 'n' operations can

be performed over 'm' clock cycles without intermediate register storage provided this would minimize area of the design.

Thus a multicycle path can be characterised by the following equation:

$$(D_{op1} + D_{op2} + .. + D_{opn}) + D_{registerstorage} \leq m * T_{clock}$$

The above equation implies that 'n' operations are performed sequentially over 'm' clock cycles with a single storage operation at the mth clock cycle. D_{opi} is the actual delay of operation i plus the delay of the multiplexer which is required for chaining(Fig. 2. Thus

$$D_{opi} = d_{opi} + d_{mux}$$

and

$$D_{registerstorage} = d_{registersetup} + d_{registerpropagation} + d_{mux}$$

After calculating the fine-grain ASAP and ALAP schedule, a quantity called *SPAN* is calculated for each of the nodes.

4.2.2 SPAN

The *SPAN* of an operation is defined as the total time range within which an operation has to begin and complete execution. Although the significance of *SPAN* may not be very apparent at this point, it actually reflects the flexibility that is available for both scheduling and component selection.

A higher 'SPAN' value indicates that the node has more freedom from the scheduling viewpoint. For example if the span of a node is 200 ns and the clock period T_{clock} is 100 ns the node could be scheduled into two possible states. The exact relation between the scheduling options and span value for a node i (i.e $span_i$) can be expressed as follows. Let C_i be a set of all scheduling options for node i. Then

$$C_i = \{x | x \geq \lfloor (ASAP_i/T_{clock}) \rfloor \wedge x \leq \lceil (ALAP_i/T_{clock}) \rceil\}$$

The total number of scheduling options is given by $|C_i|$ where

$$|C_i| = \lceil (SPAN_i/T_{clock}) \rceil$$

The SPAN value also helps us selecting the subset of components for node i. Let L the set of all components in the library.

Let $S_{opi} \subset$ L such that S_{opi} consists of all components capable of performing operation *opi*. To derive S_i which is a set of all valid selection options for node i we have:

$$S_i = \{x | x \in S_{opi} \wedge d_x \leq span_i\}$$

where d_x is the the delay of functional unit x. In essence, all components for the operator type whose delay is greater than the span time is not considered as a valid component for selection.

Thus given the SPAN value for a node, one can very easily determine the set of valid schedule and selection combinations for the node.

4.2.3 Valid Schedule and Selection Combinations

We have seen how to calculate the set of possible control step assignments C_i and a set of possible selections S_i for a node i given its SPAN value and ASAP schedules. It must be noted that both C_i and S_i were calculated independently. The next step consists of deriving the set V_i which is a set of all valid combinations.

Each element in set V_i is actually an ordered pair (x,y) such that $x \in S_i \wedge y \in C_i$. It is important to note that all possible combinations of elements in the two sets would not be valid combinations. Let us demonstrate this by an example. Assume that an adder node i has the following characteristics. $ASAP_i = 100$, $ALAP_i = 260$ and $SPAN_i = 200$. The library is shown in fig 12 and the clock period is 100 ns; let us now evaluate the set of valid combinations.

$C_i = \{2, 3\}$ since the node could be scheduled in control steps 2 or 3. Since the SPAN value is 200, only two of the three components shown in figure 12 are possible selection choices. Thus the set $S_i = \{ADD101, ADD102\}$. We could now be tempted to believe that the set V_i has 4 elements since there are 4 possible combinations of elements from S_i and C_i. However, this is not true since a combination like (ADD102, 3) is not valid. This implies that if we schedule the node to control step 3 then ADD102 is not a valid component selection since its delay is 170 ns and it would not be able to complete the operation within the SPAN value.

Thus the set V_i contains the following elements.

$$V_i = \{(ADD101, 2), (ADD102, 2), (ADD101, 3)\}$$

The cardinality of the set V_i is the number of valid choices of combinations that we would consider for the node i. At this stage we assume that any of the 3 choices indicated in V_i is equally probable. In the above example the probability of using ADD101 in control step 2 is 0.33. Similarly the remaining probabilities can be calculated. All information related to the probability that a particular component will be used at a particular control step is stored in the *probability table* described in the next subsection.

4.2.4 Probability Table

The probability table maintains the probability that a particular component in the library will be used in a control step. If the library consists of 'L' components and the number of control steps is 'T' then the dimension of the probability table is 'L * T'. The probability table provides information about the number and type of components required in a given control step. Similarly, we could also retrieve information about the control steps in which a given component could be used.

If a node 'p' has been scheduled and a component 'x' has already been selected to implement that node operation, then the set $V_p = \{(x, y)\}$ where 'x' is the selected component and 'y' is the scheduled control step. The corresponding entry in the probability table is '1'. If the node p had not been finalized and there are 'c' valid choices then the probability for each of the c entries would be 1/c. Thus the entries in the probability table indicate the probability of a component being used in a given control step.

Mathematically, an entry in the probability table is defined as the sum of the probability contributions from all nodes 'N' in the CDFG.

$$P(x, y) = \sum_{i=1}^{N} p_i(x, y)$$

where $p_i(x, y)$ is the probability that node i in CDFG would use component x in control step y and could be defined as

$$p_i(x,y) = 1/|V_i| \wedge (x,y) \in V_i$$

Let us illustrate the above concepts with a simple example. Given the sets of valid choices for three nodes n1, n2 and n3:

$$V_{n1} = \{(ADD101, 1)\}$$

$$V_{n2} = \{(ADD101, 1), (ADD102, 1), (ADD101, 2)(ADD102, 2)\}$$

$$V_{n3} = \{(ADD101, 2), (ADD102, 2), (ADD103, 3), (ADD101, 3)(ADD102, 3)\}$$

If the above nodes are being scheduled into 3 control steps the probability table is shown in figure 3.

	ADD101	ADD102	ADD103		COMP L
cs-1	1 + 0.25	0.25			
cs-2	0.25 + 0.2	0.25 + 0.2			
cs-3	0.2	0.2	0.2		

Figure 3: Example Probability Table

The table entry for (ADD101,1) is 1.25 since the probability that node *n1* would be implemented with ADD101 in control step 1 is 1. and there is a probability of 0.25 that node *n2* would also be implemented similarly. The other entries in the table are calculated similarly. After calculating all the entries in the probability table we can evaluate the cost function which reflects the layout area of the functional components in the circuit.

4.2.5 Cost Function Evaluation

Given the probability table, it is quite simple to estimate the functional area of the resultant circuit. A single column in the probability table gives the probability that a particular

component would be used in each of the control steps. If a particular entry in a column is 'p' then we know that the component represented by that column would be required to be allocated with a probability p. If all other entries in the column are less than p we do not need to allocate any extra hardware since the hardware allocated for the entry p would be available to be used in other clock cycles.

Thus the probability that a component would be used in the design is the biggest entry in the column which represents that particular component. If the probability is greater than '1' then the design would require more than one component of that particular type. This probability value multiplied by the area of the component (data available in the library) gives the total area of this component in the whole design.

The total area of the design can be similarly estimated from the probability table. The area of the design can be expressed as

$$Estimated\_Area = \sum_{i=1}^{L} p_{max}(i) * Area(i)$$

where $p_{max}(i)$ is the maximum entry in column i of the probability table and $Area(i)$ is the area of the component in column i.

In the probability table of figure 3 the probability that ADD101 would be used in the design is 1.25. The probabilistic area estimate for ADD101 is thus 1.25 * 100. Similarly the area estimate for ADD103 is 0.2 * 40.

We have just seen how we could estimate the total area of the design for a particular configuration represented in the probability table. The next step would indicate how we accept the best selection and schedule for a node.

4.2.6 Finalisation of Schedule and Selection

Given the list of all valid choices V_i for a node, an area estimate as shown in the previous section, is obtained for each of the possible choices. The choice that results in the lowest area estimate is accepted to be the final schedule and selection for the node.

When a node's selection and schedule is finalised, the ASAP and SPAN values of some of the

nodes in the CDFG would be affected. The delta-scheduling step in the algorithm adjusts the ASAP values to reflect the current configuration of the design.

This process of evaluating the various choices for a node and accepting the best possible choice is continued till all the nodes in the flowgraph are scheduled and selected.

We will now demonstrate *CHASSIS* with a small example.

5 An Example

To illustrate *CHASSIS* we will step through a simple example consisting of 3 add operations. A sample library containing different types of adders with different delay and area is chosen for the example. To make this example simple we will ignore the register and multiplexer delays. The high level description of the design, the library and the overall design constraints are shown in figure 4.

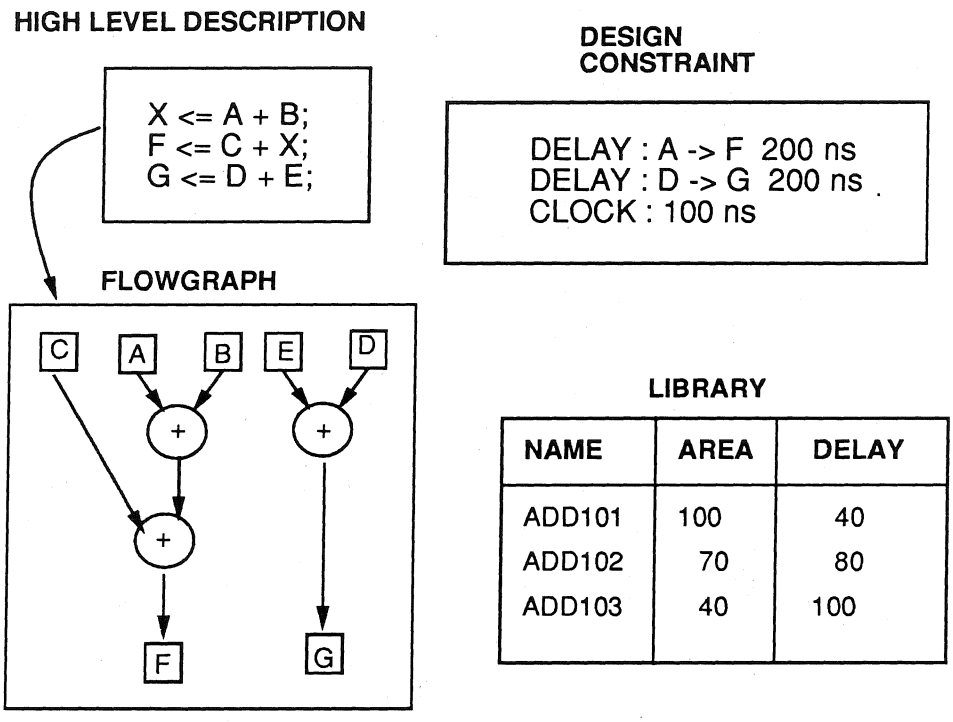
Since the time constraint is 200 ns and the clock period is 100ns it is obvious that we have to schedule all the operations in two clock cycles, choosing the components in a manner that reduces the overall area of the design.

We start the design process by assuming that the fastest components from the library shown in figure 4 (i.e ADD101) will be used for each operation. We then perform an ASAP and ALAP scheduling of the flowgraph and evaluate the SPAN values. The results of these scheduling operations and the span values for each of the nodes are shown in figure 5.

From the ASAP and SPAN values, we can now derive the possible scheduling and selection choices for each node. Let us look at node A. Its span value is 160. This implies that ADD103 cannot be used to implement the addition operation node A. Hence there are only two possible selections for node A. Thus the set S_A contains two elements.

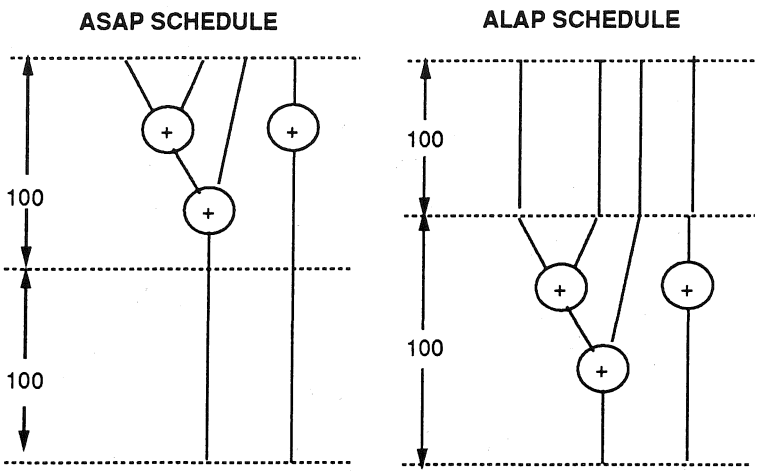
$$S_A = \{ADD101, ADD102\}$$

Similarly there are two possible selections for node B. However node C's span value is greater than 180 which is the delay of the component ADD103. Hence C can be implemented with



NAME	AREA	DELAY
ADD101	100	40
ADD102	70	80
ADD103	40	100

Figure 4: An example to demonstrate Selection-Scheduler



NODE	ASAP	ALAP	SPAN
A	0	120	160
B	40	160	160
C	0	160	200

Figure 5: ASAP - ALAP - SPAN calculations

any of the 3 adders in the library.

$$S_B = \{ADD101, ADD102\}$$

$$S_C = \{ADD101, ADD102, ADD103\}$$

By considering the ASAP and SPAN values for the nodes we can obtain the possible choices for scheduling the nodes. For node A the ASAP value is 0 and the SPAN is 160. There are two choices for scheduling node A (control steps 1 and 2). Similarly, we obtain the choices for nodes B and C.

$$C_A = \{1, 2\}$$

$$C_B = \{1, 2\}$$

$$C_C = \{1, 2\}$$

From the sets S and C for each node we derive the set of valid options for the nodes. We have a total of 4 possibilities for node A when we combine its scheduling and the selection options. The set of valid choices V for each operation can be expressed as

$$V_A = \{(ADD101, 1), (ADD101, 2), (ADD102, 1), (ADD102, 2)\}$$

$$V_B = \{(ADD101, 1), (ADD101, 2), (ADD102, 1), (ADD102, 2)\}$$

$$V_C = \{(ADD101, 1), (ADD101, 2), (ADD102, 1), (ADD102, 2), (ADD103, 1)\}$$

From the sets V_A, V_B and V_C we can derive the probability that a particular operation would be implemented by a particular component. There is a 0.25 probability that node A would be implemented with ADD101 and scheduled in control step 1. The equivalent probabilities for nodes B and C are 0.25 and 0.2. Thus the total probability that component ADD101 would be used in control step 1 is 0.25 + 0.25 + 0.2 which is 0.7. Thus the probability table contains the value 0.7 for the location (ADD101, 1)

Using the probability table and the set V_A we estimate the area of the final design for each of the 4 choices of node A. If we temporarily finalise the selection of node A to be ADD101 and the schedule to be control step 1, then the probability table would change to the table shown in figure 7.

	ADD101	ADD102	ADD103
CS 1	0.7	0.7	0.2
CS 2	0.7	0.7	0.0

Figure 6: Probability Table for Example

The probability value of the entry (cs-1, ADD101) has changed to $1 + 0.2 + 0.25 = 1.45$. The other entries have changed correspondingly to denote that node A does not contribute to any other probability entry.

The area can be estimated using the estimation procedure in the previous section. The first row in figure 8 shows the probabilistic area of the design if we choose to implement node A with ADD101 in control step 1. The area estimates for the remaining three choices for node A are also shown in the figure.

	ADD101	ADD102	ADD103
CS 1	1.45	0.45	0.2
CS 2	0.45	0.45	0.0

Figure 7: Prob. Table when A is assigned

It is important to note that the algorithm estimates an area value of infinity for the combination (ADD102, cs-2) for node A. The reason for this is very simple. If node A were to be scheduled in cs-2 with ADD102 then there is no way we can implement the design since node B cannot be scheduled within the overall performance constraint. The SPAN value for node

B would become 20 and no components with delay less than 20 ns are available.

From figure 8 we can conclude that the best way to implement node A is by using ADD102 and scheduling it in control step 1. Our results exactly match what we would have intuitively expected since it is clear that using ADD102 (a component with a lower area) would be better than using ADD101 if there are stringent time constraints as in this case.

Control Step	Library Component	Estimated Area
1	ADD101	216
2	ADD101	256
1	ADD102	182
2	ADD102	INFINITY

Figure 8: Area Estimation for node A

We now evaluate the various choices available to B. The ASAP values of B would have changed since A has been assigned to a fixed component from the library. The ASAP value for B is 100. The span value for node B is 100. The choices for node B are reduced to 2. We estimate the probabilistic area for the two choices of B as shown below.

Control Step	Library Component	Estimated Area
2	ADD101	212
2	ADD102	146

Figure 9: Area Estimation for node B

Let us examine the results for node B. It is clear that implementing node B in control step

2 with ADD102 produces the minimal area. This corresponds to our intuition; if we had already used component ADD102 in the previous time step to implement node A, we would automatically try to use the same component in control step 2, since it would not require any additional allocation.

Finally, we evaluate the probabilistic area for node C and the best choice for this node is ADD103 since this adds the minimal area to our design. The area estimates for the various choices for node C is shown in figure 10. The estimates indicate that the best selection for node C is ADD103 since that would lead to the minimal area. Intuitively this appears to be the best choice, since C has so much freedom in terms of time, it would be wise to choose the component with minimal area.

Control Step	Library Component	Estimated Area
1	ADD101	170
2	ADD101	170
1	ADD102	140
2	ADD102	140
1	ADD103	110

Figure 10: Area Estimation for node C

The final design consists of one adder ADD102 and one adder of type ADD103. The implementation is shown in the figure 11.

6 Experiments and Results

We have implemented *CHASSIS* using the 'C' Programming Language on a SUN Workstation(Sparc). We have verified our technique by performing a few experiments on a couple of standard benchmark examples. In this section, we will discuss our experiments with the

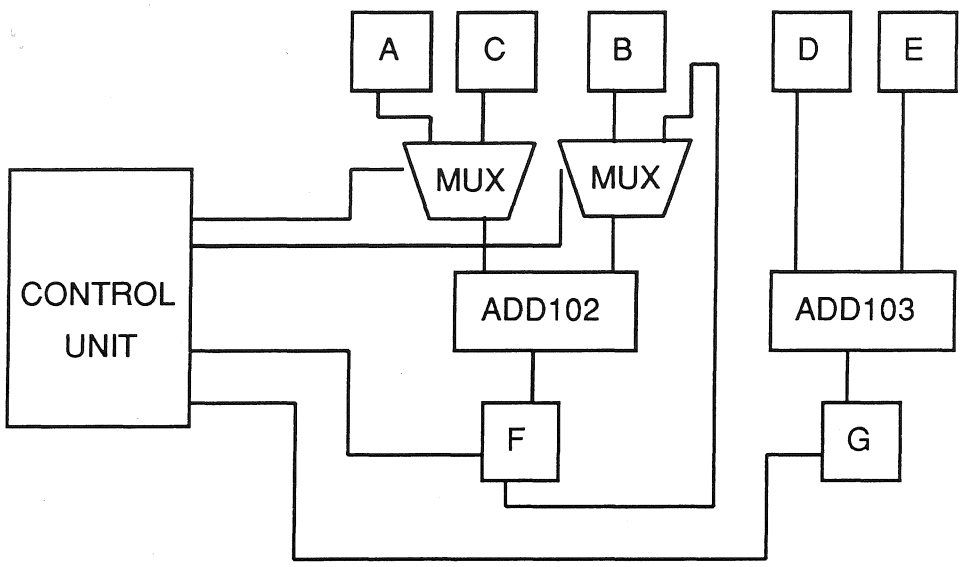


Figure 11: Implementation of the final design

elliptic filter example which is a standard benchmark used by many research groups. This example, originally from the signal processing book by Kung et.al [13] was used as a standard example in the High Level Synthesis Workshop 1988.

The VHDL code for the body of the elliptic filter is shown below. The results of *CHASSIS* for both the libraries are shown.

6.0.7 VHDL Description

```
entity ELLIPTIC_FILTER is
        port (In_port: in BIT; Out_port: out BIT);
end ELLIPTIC_FILTER;

architecture EX of ELLIPTIC_FILTER is
begin

process
        variable m1,m2,m3,m4,m5,m6,m7,m8: BIT ;
        variable n2, i, ott : BIT ;
        variable n40,n41,n42,n43,n44,n47,n46,n48 :BIT;
        variable n49,n50,n51,n52,n53,n55,n56,n57, n58 :BIT;
        variable n59,n60,n61,n18,n13,n26,n54,n63 : BIT;
        variable n64,n65,n66,n38,n33,n67,n39: BIT;
begin
        i := In_port;
        n40 := n2 + i;
        n43 := n33 + n39;
        n41 := n40 + n13;
        n42 := n41 + n26;
        n44 := n42 + n43;
        n47 := n44 * m1;
        n46 := n44 * m2;
        n48 := n47 + n41;
        n49 := n46 + n43;
        n51 := n48 + n41;
        n52 := n49 + n43;
        n50 := n48 + n44;
        n53 := n51 * m3;
```

```
        n54 := n52 * m4;
        n26 := n49 + n50;
        n55 := n53 + n40;
        n63 := n54 + n39;
        n59 := n48 + n55;
        n64 := n63 + n49;
        n56 := n55 + n40;
        n65 := n64 + n38;
        n60 := n59 + n18;
        n67 := n63 + n39;
        n57 := n56 * m5;
        n66 := n65 * m6;
        n58 := n57 + i;
        n38 := n66 + n38;
        n2 := n58 + n55;
        n33 := n38 + n65;
        n61 := n60 * m7;
        ott := n67 * m8;
        n18 := n61 + n18;
        n39 := ott + n63;
        n13 := n18 + n60;
        Out_port := ott;
end process;
end EX;
```

To synthesize the elliptic filter we created a library of components shown in figure [12]. The library has three different components capable of performing the standard operations like addition and multiplication. Each component varies widely in its delay and area characteristics. As an example there are three adders whose delay varies from 55 ns to 220 ns. In a design

that uses a clock period of 100 ns, an add operation could take one, two or three clock cycles based on which component is being used to perform the addition. The library also contained other components like drivers and registers that are required to complete the design.

COMPONENT	NAME	AREA	DELAY
ADDER	ADD101	1500	55
	ADD102	500	170
	ADD103	300	220
SUBTRACTOR	SUB101	1500	55
	SUB102	500	170
	SUB103	300	220
MULTIPLIER	MUL101	17000	80
	MUL102	8000	180
	MUL103	3000	260
COMPARATOR	COM101	300	20
REGISTER	REG101	1200	
(setup)			2
(hold)			2
TRISTATE DR.	TRD101	200	3

Figure 12: Component Library

Using the above library Fig. [12] we synthesized the elliptic filter for a wide range of performance constraints ranging from 1800 ns to 4000 ns. The clock period in all our experiments was 100 ns. *CHASSIS* was used to schedule each operation into an appropriate control step and simultaneously select the best component from the library to perform the operation. The

results of are shown in figure [13]. The results clearly indicate that in general a mixture of components is required to implement the elliptic filter efficiently. Let us consider the case where the performance constraint was 2900 ns. *CHASSIS* used three different types of adders to implement the design. (ADD101, ADD102, ADD103). Thus some add operations were performed in one clock cycle, some in two clock cycles and some in three clock cycles based on which component it was bound to. When the performance constraint was 2200 ns, three adders were required, but it selected all three fast adders (ADD101). The functional area of the design with 2200 ns constraint was higher than the functional area of the design with the 2900 ns constraint since faster adders are more expensive.

In general, *CHASSIS* was able to determine an efficient mix of components required to complete the design within the given performance constraint. It ensured that critical operations were being performed in fast components and non critical operations were being performed by slower components.

With the next experiment we actually verified the fact that better designs are indeed obtained by combing selection and scheduling. In order to ascertain this, we compared the designs produced by *CHASSIS* when it worked with the full library, to the designs obtained when using *Reduced Libraries* containing single implementations of each operation type.

Since there were three adders and three multipliers in the original library we created all the possible Reduced Libraries containing a single adder and a single multiplier and other components used to complete the design. The add and multiply components in the the nine *Reduced Libraries* were RL1(ADD101, MUL101), RL2(ADD102, MUL101), RL3(ADD103, MUL101), RL4(ADD101, MUL102), RL5(ADD102, MUL102), RL6(ADD103, MUL102), RL7(ADD101, MUL103), RL8(ADD102, MUL103), RL9(ADD103, MUL103).

With each of these libraries, we synthesized the elliptic filter for all the performance constraints (1800 ns to 4000 ns). In figure [13] we have shown the Area Time tradeoff curves for the entire library shown in Fig. 12 and the *Reduced Libraries.* The curve for RL9 is not shown since the critical path was 4200 ns for the fastest design that is implementable with RL9. This is due to the fact that the components belonging to RL9 are ADD103 and MUL103 which take three clock cycle delays to complete their operations.

DLY	LIB	ADD 103	ADD 102	ADD 101	SUB 103	SUB 102	SUB 101	MUL 103	MUL 102	MUL 101	FU AREA
1800	FULL	2		2						1	20600
1900	FULL			2				2	1		17000
2000	FULL		1	2				1	1		14500
2100	FULL		2	2				4			16000
2200	FULL			3				2			10500
2300	FULL			2				2			9000
2400	FULL			2				2			9000
2500	FULL		2	1				2			8500
2600	FULL		1	1				2			8000
2700	FULL		1	1				2			8000
2800	FULL		1	1				2			8000
2900	FULL	1	1	1				2			8300
3000	FULL	1		1				2			7800
3100	FULL	1		1				2			7800
3200	FULL		2	1				2			8500
3300	FULL	1	1	1				2			8300
3400	FULL		3					2			7500
3500	FULL	1		1				2			7800
3600	FULL	1		1				1			4800
3700	FULL	1		1				1			4800
3800	FULL		2					1			4000
3900	FULL		2					1			4000
4000	FULL		2					1			4000

Figure 13: *CHASSIS* Results - Elliptic Filter Example

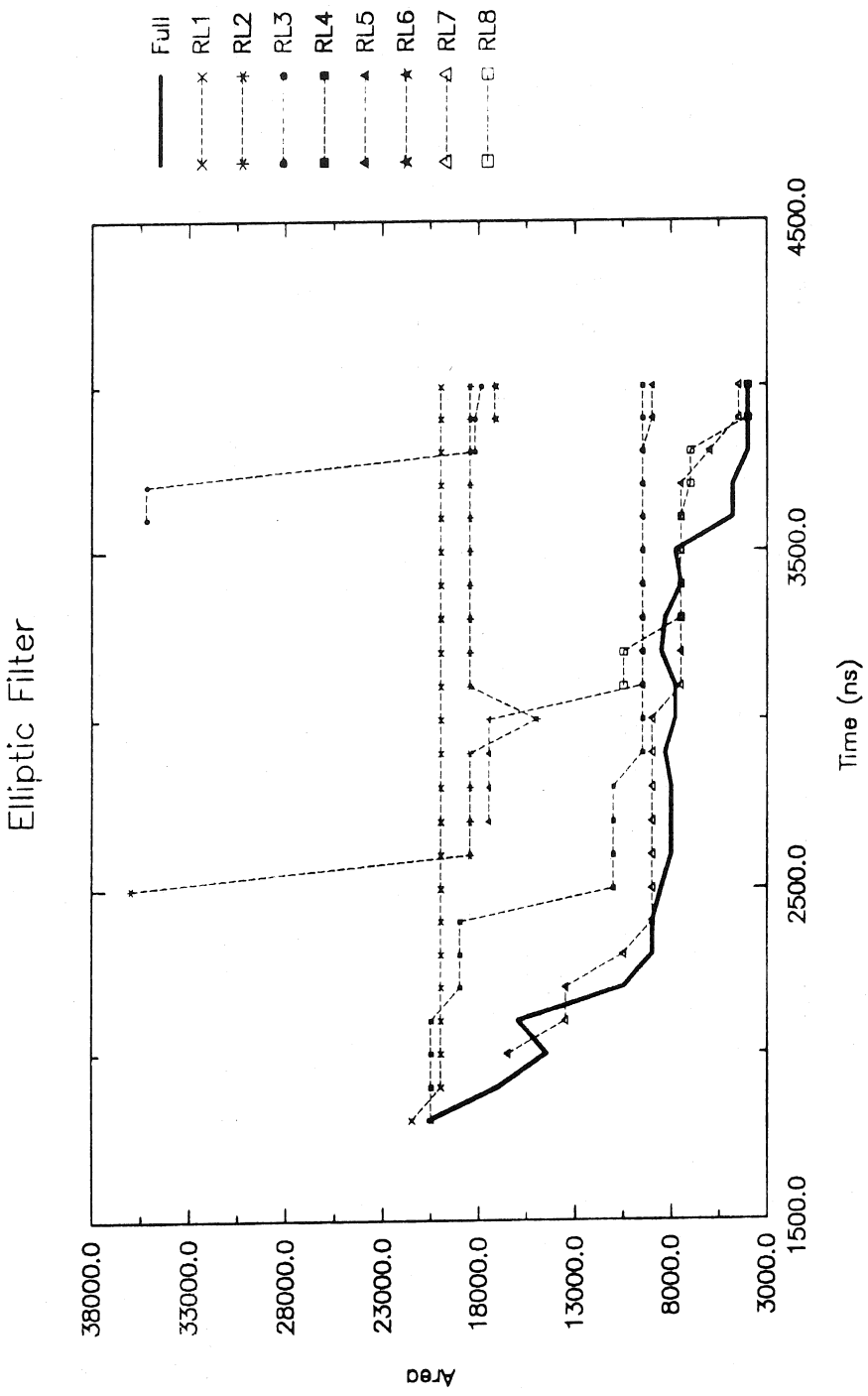


Figure 14: AT Curve - Elliptic Filter

In general the Reduced Libraries could not produce better designs. This is because the only available implementation of a component (however expensive it is!) has to be used to implement all critical and non-critical operations resulting in inefficient designs. There were a couple of instances (less than 1 % of the cases) where *CHASSIS* could not converge on the minimal design. We are currently improving the iterative improvement techniques to encompass a wider search space and converge on to the minimal design in all cases.

However, we can conclude, that performing scheduling with single implementations of components cannot lead to optimal designs. It is important to consider component tradeoffs when performing scheduling in order to be able to produce high quality designs.

7 Conclusions

In this paper we have clearly demonstrated how the quality of synthesized designs could be improved by performing scheduling and selection as an interlinked operation. We have described a new technique *CHASSIS* which is one of the few research efforts to actually combine these two tasks into one and perform scheduling while simultaneously utilizing the wide range of library components.

As an extension to this effort, we are currently working on expanding the design model to incorporate interconnect delay estimates obtained from actual floorplan or from wire-length estimates obtained from an estimation tool. We are also studying the *CHASSIS* heuristic on large designs to examine whether any improvements are necessary.

8 Acknowledgements

This work was supported by grants from the National Science Foundation (#MIP-8922851) , the Semiconductor Research Corporation (#90-DJ-145), the California MICRO program (# 90-047), TRW and Texas Instruments. We are grateful for their support. We would also like to thank all members of the UCI CADLAB for their helpful suggestions.

9 References

[1] Richard Cloutier and Donal E Thomas. The Combination of Scheduling, Allocation and Mapping in a Single Algorithm. In *Proc. of the 27th Design Automation Conf.* ACM/IEEE, June 1990.

[2] C.Tseng and D.P. Siewiorek. Automated Synthesis of Datapaths in Digital Systems. *IEEE Transactions on CAD*, pages 379–395, July 1986.

[3] Alice C. Parker et el. MAHA: A Program for Datapath Synthesis. In *Proc. of the 23rd Design Automation Conf.*, pages 22–226. ACM/IEEE, June 1986.

[4] L. Hafer and Alice Parker. A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic. *IEEE Transactions on CAD*, pages 11–14, January 1983.

[5] Rajiv Jain. MOSP: Module Selection for Pipelined Designs with Multi-Cycle Operations. In *Proc. of the IEEE Conference on Computer Aided Design.*, pages 212–215. IEEE, September 1990.

[6] J.Nestor and D.E Thomas. Behavioral Synthesis with Interfaces. In *Proc. of the IEEE Conference on Computer Aided Design.*, pages 112–115. IEEE, November 1986.

[7] G.W. Leive. *The Design, Implementation and Analysis of an Automated Logic Synthesis and Module Selection System.* PhD thesis, Carnegie-Mellon University, January 1981.

[8] Alex Orailoglu and D. D. Gajski. Flow Graph Representation. In *Proc. of the 23rd Design Automation Conf.*, pages 503–509. IEEE/ACM, June 1986.

[9] B.M Pangrle and D.D Gajski. State Synthesis and Connectivity Binding for Microarchitecture Compilation. In *Proc. of the IEEE Conference on Computer Aided Design.*, pages 210–213. IEEE, November 1986.

[10] Pierre G Paulin and John P Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on CAD*, pages 661–678, June 1989.

[11] Pierre G Paulin and John P Knight. Scheduling and Binding Algorithms for High-Level Synthesis. In *Proc. of the 26rd Design Automation Conf.*, pages 1–6. ACM/IEEE, June 1989.

[12] Alice Parker Rajiv Jain and Nohbyung Park. Module Selection for Pipelined Synthesis. In *Proc. of the 25th Design Automation Conf.*, pages 542–547. IEEE/ACM, June 1988.

[13] H. J. Whitehouse S.Y Kung and T. Kailath. *VLSI and Modern Signal Processing.* Prentice Hall Information and Systems Sciences Series, 1985.

[14] Y-P.S.Foo and H. Kobayashi. A Knowledge Based System for VLSI Module Selection. In *Proc. of the IEEE Conference on Computer Aided Design.*, pages 212–215. IEEE, January 1983.