ABSTRACT
A new approach is presented for simultaneous scheduling and connectivity binding in a behavioral synthesis system. A branch-and-bound algorithm is applied for scheduling, with connectivity binding performed at each intermediate step. Costs from the connectivity binder are used to direct the search for optimal solutions. This approach allows the program to optimize user defined objectives without any implicit biases, such as trying to achieve the fastest schedule. Some heuristics are presented to estimate the cost of a partially scheduled and bound graph. This helps to prune the search space. The performance of this algorithm is presented using examples from the literature.

1. INTRODUCTION
Datapath synthesis comprises two sub-tasks: behavioral synthesis and structural synthesis. Behavioral synthesis translates a high level functional description into a register-transfer level structure. Structural synthesis maps this structure into layout.

The input to a behavioral synthesis system is a high-level language, e.g. Pascal or ISPS [1], that is compiled into an intermediate representation, like a control/data flow graph (CDFG) [10] or the Value Trace [9]. This representation defines data and control constraints on the ordering of operations. Traditionally, the process of mapping this representation into a structure has been divided into two major tasks; scheduling and connectivity binding. Scheduling partitions the operations into states i.e. assigns each operation to be executed during one or more states. Scheduling is constrained by dependencies specified in the intermediate representation. Connectivity Binding creates a mapping between elements of the graph representation and structural units. It assigns registers to variables, function units to operations and uses multiplexers and buses to interconnect registers and function units. The other tasks include, selecting a set of function units, determining an appropriate clocking scheme, and generating the control unit. A high-level behavioral synthesis system should ideally attempt to optimize the circuit performance and cost within a constrained region of the design space.

2. PREVIOUS WORK
The simplest scheduling technique, called "As soon as possible" or ASAP (e.g. Emerald/Facet [16]), assigns operations to the earliest cycle. ASAP scheduling extracts the maximum parallelism in the design but, in many instances, uses excessive hardware. A class of algorithms called list scheduling algorithms try to overcome this shortcoming by assigning scheduling priorities to operations. Amongst the various heuristics used are; node urgency in Elf [4]; node mobility in Slicer [11]; and operation freedom in MAHA [14]. All these heuristics assign maximum priority to operations that lie on the critical path. Another scheduling technique is the force directed approach that has been used in HAL [13]. This approach tries to spread out operations of the same type across machine cycles so that minimal hardware is required.

The connectivity binding algorithms that have been reported mostly use two types of connectivity models. One is the point-to-point connectivity model (e.g. EMUCS [15], HAL [13] and MAHA [14]) and the other is the bus based model (e.g. SPLICER [12], Facetherald [16] and DAA [7]). In the first model, multiplexers are used at the inputs of registers and function units, where as in the bus based model the outputs of components are connected to buses and the buses are connected to component inputs. The bus based model is more general and is roughly equivalent to using two levels of multiplexers in terms of connectivity. The scheduling and connectivity binding tasks are highly interrelated so that decisions made in one task impact the other. However, in most systems, including EMUCS [15], Emerald [16] and Splicer [12] the two tasks are done independently, with the scheduling preceding connectivity binding. The aim of the scheduler is to produce a design which executes in the minimum number of cycles. The connectivity binder tries to produce a minimum cost binding for that schedule. In this approach the scheduler does not consider connectivity costs at all. HAL [13] also performs scheduling and connectivity binding separately, but uses the following mechanism so that connectivity costs can influence the schedule. First, connectivity binding is done on a coarse schedule. The results from the connectivity binder are then used to refine the scheduling. However, the feedback in this arrangement is at a very coarse level, as either the scheduling is fixed during the entire connectivity binding process or vice-versa. A notable exception is Elf [4] which uses a constructive approach for simultaneous scheduling and connectivity binding. For each machine cycle, operations with the maximum urgency are scheduled and bound using the minimum cost. Operations that are not urgent can be
delayed depending on their connectivity costs. Since this algorithm is constructive, the cost of delaying an operation is not known. Hence it will only find local optima.

Schalloc, the algorithm proposed in this paper overcomes the shortcomings of the previous systems by performing scheduling and connectivity binding simultaneously. A branch-and-bound approach is employed to search through possible schedules for the given graph. The algorithm is similar to the branch-and-bound approach used for local microcode compaction [8]. The search through the design space is directed by invoking the connectivity binder for each partial schedule and employing connectivity costs to optimize a user defined cost function. A technique similar to A* [6] is presented for improving the search speed by predicting the total cost of partial designs.

Section 3 presents an overview of Schalloc, and Sections 4, 5 and 6 present the algorithm and various heuristics. Methods to explore the design space, and some examples are presented in Sections 7 and 8. Our conclusions are presented in Section 9.

3. OVERVIEW OF SCHALLOC

Schalloc fits within the Chippe [3] design paradigm, in which the design synthesis knowledge is considered separable from the design analysis knowledge. In this paradigm, the design synthesis tools operate within constraints provided by the design analyzer. The design analyzer evaluates the work of synthesis tools and makes global decisions to direct the design process towards its goals. The design analyzer decides on the function unit set and the clock cycle length used by Schalloc and other synthesis tools. Splicer [12] does the connectivity binding. The input language is a Pascal/ISPS like language that is compiled into a CDFG.

Schalloc is a branch-and-bound, recursive algorithm which uses a depth-first strategy for scheduling. The recursive procedure begins around two entities - max-micro-group and ready-list. A micro-group is defined as a group of operations that can be executed in one cycle. The operations that are eligible for a micro-group are partially determined by the available components. A max-micro-group is defined as a micro-group which is not a proper subset of any other micro-group. The ready-list for any cycle is the list of operations that have their input dependencies satisfied at the beginning of that cycle. Initially, max-micro-group is null, and the ready-list consists of all operations whose inputs are initially valid. The algorithm is described in Figure 1.

At each step, the algorithm calculates the possible max-micro-group and ready-list pairs, i.e., the corresponding set of operations that are ready for scheduling when the operations in max-micro-group are scheduled. Since the algorithm uses a depth-first search, the order in which max-micro-group, ready-list pairs are arranged is significant. A good first solution can significantly prune the search space. In Section 5, we present some heuristics for ordering the max-micro-group and ready-list pairs.

Splicer performs the connectivity binding for each max-micro-group. The design statistics returned from Splicer are used to determine the quality of a schedule. The algorithm keeps track of the best solution and compares all other solutions with it. The search continues until the ready-list is exhausted, or the cost exceeds the cost of the best solution. Each time a lower cost solution is found, it becomes the new best solution.

4. GENERATING MAX-MICRO-GROUPS

A depth-first, recursive procedure (outlined in Figure 2) is used to generate the set of max-micro-groups given a set of operations that are already scheduled, and a set of operations that are ready to be scheduled. The max-micro-groups are constrained by the data/control constraints between operators and the available components or resources.

The ready-list initially contains all operations that have their dependencies satisfied. Max-micro-group is initialized to null. At the root of the call, one operation from the ready-list is considered for inclusion in the max-micro-group. If feasible, the operation is removed from the ready-list and the operation is considered for inclusion in the max-micro-group. If an operation x ∈ ready-list is compatible with a max-micro-group c if a suitable function unit is free to execute x. Operations in the ready-list are ordered according to the number of dependencies on them. The first operation in the max-micro-group gets the fastest function unit, the later ones get slower function units. Thus, the fastest unit that can execute the operation x, and that is not already bound to some operation in max-micro-group is chosen. Because of this, operations with more dependents will be assigned faster function units. To accommodate multi-cycle and pipelined operations, the procedure in Figure 2 also generates max-micro-group that can extend across more than one micro-instruction.

5. CONNECTIVITY BINDING USING SPICER

Splicer [12] is a connectivity binder that uses a bus-style connectivity model rather than the multiplexer-based point-to-point connectivity model. It keeps track of the statistics of the design, such as the number of registers, multiplexers,
max-micro-group \leftarrow \emptyset; \\
S \leftarrow \emptyset; \quad \text{"S is the set of (max-micro-group, ready-list) pairs"} \\
generate_max_micro_groups (schedule, max-micro-group, ready-list) \\
{} \\
\text{if (ready-list \neq \emptyset) then} \\
\quad \text{for each } x \in \text{ready-list do} \\
\quad \quad \text{if compatible (max-micro-group, x) then} \\
\quad \quad \quad \text{ready-list' \leftarrow ready-list + dependent(x) - x; } \\
\quad \quad \quad \text{"dependent(x) returns all operations dependent on } x, \text{excluding those that are also dependent on some other operation } y, \text{where } y \text{ is not already scheduled and not in max-micro-group"} \\
\quad \quad \text{max-micro-group' \leftarrow max-micro-group + x; } \\
\quad \quad \text{generate_max_micro_groups (schedule, max-micro-group', ready-list'); } \\
\quad \} \\
\} \\
\} \\
\} \\
\text{FIGURE 2. Procedure to generate max_micro_groups} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \\
{} \

1. Number of cycles - the longest path in the remaining graph is identified. Since an underestimate is required, the fastest function units available are assumed for each operation. The operations on the longest path are temporarily scheduled into machine cycles to give an estimate of the number of cycles required for the remaining graph. This heuristic has an order of complexity $O(n^2)$ [2], where $n$ is the number of operations in the remaining graph.

2. Urgency of operations - the number of operations that are dependent on the operations in the max-micro-group under consideration are counted. This gives an estimate of the "urgency" of the max-micro-group where a greater number of dependent operations imply a higher urgency for scheduling.

7. EXPLORING THE DESIGN SPACE

As discussed earlier, Schalloc is based on the Chippe paradigm, where the design analyzer makes the global decisions, such as selecting the function unit set and the length of the clock cycle. With these fixed, Schalloc finds the best design based on a cost-function. The design analyzer can further control the search using three mechanisms:

1. Maximum constraints: the user can specify an upper bound on any design parameter, for example, the number of cycles, the number of registers, etc.

2. Minimum constraints: the user can specify a set of minimum design parameters, where any design satisfying these parameters is considered good enough, and the search stops there.

3. Maximum iterations: the search stops when the number of recursive calls to Schalloc exceeds the specified maximum.

8. IMPLEMENTATION AND EXAMPLES

The code for Schalloc is written in C and runs under 4.2 BSD UNIX. The examples in this section were chosen because they have often been referenced by other papers in this field. The CPU times presented for the designs are for a SUN 3/260.

8.1. DIFFERENTIAL EQUATION EXAMPLE

This example is from HAL [13]. Figure 3 shows the CDFG for this example. The function units used for the design were 2 multipliers, 1 adder, 1 subtractor, and 1 comparator. All units execute in 1 machine cycle.

The designs produced by Schalloc are shown in Table 1. Design style in the table represents the objective function used, which is a weighted function of the parameters specified in the column. The weights attached to parameters decrease from left to right.

The critical path for this CDFG is 4 states. The HAL design for this example has 4 cycles and uses 14 multiplexer inputs, 6 multiplexers, and 6 registers. This design was generated in 140 CPU seconds on a Xerox 1109 Lisp machine. Schalloc's design for minimum cycles and multiplexer inputs has 4 cycles and uses 12 multiplexer inputs, 5 multiplexers and the same number of registers. This design was generated in 388 CPU seconds on a SUN 3/260. For this design operations were more or less uniformly distributed across machine cycles. The first three states had 3 operations each and the last one had 2.
Schalloc's design for minimum number of multiplexer inputs uses 4 multiplexers, 11 multiplexer inputs, and executes in 6 machine cycles. Figure 3 presents the state partitioning for this design. Once again a uniform distribution of operations amongst the machine cycles resulted in a lower cost design. Usually when there is a non-uniform distribution of operations amongst states, the broader states create excess connections that are not utilized in the narrower states.

![Figure 3 A uniform distribution of operators across cycles.](image)

The design for minimum number of buses uses just 3. This design takes 11 machine cycles, i.e. one operation per cycle. Since all operators in this example have two inputs and one output, the minimum number of buses required for any design is 3 (assuming separate buses for inputs and outputs). For designs 1 and 2, all micro-groups are tried rather than just the max-micro-groups. The procedure to generate these is the same as that for max-micro-groups (see Figure 2) except that the restriction that a max-micro-group may not be a subset of another is removed.

8.2. ELLIPTICAL FILTER EXAMPLE

This is an example of a fifth-order Elliptical Filter used by Paulin [PaKGB86]. The function units used are: 1 multiplier that executes in 2 machine cycles, and 2 adders that execute in 1 machine cycle.

Table 2 shows the designs produced by Schalloc. Designs 1 & 2 illustrate the advantage of using heuristics to project the cost of partial solutions. Both aim for minimum cycles. Design 1 uses the heuristic for minimum cycles, whereas design 2 is pure branch-and-bound. The ratio of their CPU times is 1.234. This is representative of the computation time savings achieved by using the current set of heuristics.

This example has also been referenced in a paper by Haroun and Elmasry [5] which describes the SPAID system. In SPAID's model registers are grouped in register files with a dedicated bus used for input and output connected to each register file. The SPAID design for the same function unit set takes 22 cycles, 5 buses, 19 registers, and 17 multiplexer inputs. SPAID's multiplexer input count does not include the multiplexers required to select registers from register files, or those required to connect function unit outputs to buses. The actual multiplexer count depends on their design which we did not have available. For comparison purposes only, a worst case mux-input count for SPAID's design using our connectivity model can be computed as follows: assume that there are at least 2 registers in every register file and every function unit output is connected to all the buses, then a worst case figure would be (19(registers) + 3(function units)) * 5(buses) + 17(muxin)) = 51 multiplexer inputs for the SPAID design. Obviously, their results could be better than this, but they would be no worse in terms of multiplexer inputs. Also the cycle time comparisons are not equivalent. Since SPAID assumes that buses are used for both input and output, SPAID's clock cycle should be longer than Schalloc's which uses separate input and output buses. If using buses for both input and outputs only increased the clock time by 20%, then SPAID's 22 cycle design would be time equivalent to approximately a 26 cycle design for Schalloc.

Schalloc's design for minimum cycles and multiplexer inputs takes 21 cycles, 45 multiplexer inputs, 13 registers and 32 buses. In contrast the design for minimum cycles and buses takes 9 buses, 53 multiplexer inputs and the same number of cycles. The design for minimum buses, takes just 4 buses and executes in 42 cycles. Like the Differential Equation example for minimum buses, all the states have one operation except two states, which have two operations.

8.3. OVEN CONTROLLER EXAMPLE

This example is from Elf [4]. The component set for this example comprises: 1) an ALU that performs add, subtract, and shift operations, 2) a comparator, and 3) a function unit that can perform logical AND. All function units execute in one cycle.

The Elf design uses 19 multiplexer inputs, 8 multiplexers, and 8 registers. The count for the number of cycles for that design were not available. This design was generated in 19 CPU seconds on an IBM 3033. Schalloc's design for minimum cycles and multiplexer inputs executes in 19 machine cycles and uses 15 multiplexer inputs, 5 multiplexers and 10 registers. This design was generated in 26 CPU seconds on a SUN 3/260. By relaxing the time constraint to allow one more cycle, Schalloc produced a design that reduces the multiplexer inputs to 13, and multiplexers to 3 while adding a register. The design for minimum cycles and buses takes 19 cycles and 9 buses. Again, by relaxing the time constraint Schalloc produces a design that takes 23 cycles and 7 buses. Schalloc thus allows the user to explore cost versus performance tradeoffs.
schedule and thus ignore a complete depth function of the design space. Using the HAL example is it shown that Schalloc can be used as an effective tool for exploring cost versus performance tradeoffs. A uniform distribution of operations across states generally results in lower connectivity costs. Design number 2 and 3 for the HAL example illustrate this.

The heuristics developed for estimating the cost of a partial solution speed-up the algorithm considerably. More work needs to be done to develop heuristics for estimating multiplexers and multiplexer inputs.

10. ACKNOWLEDGMENTS
The work was supported in part by NSF grant No. MIP-8809250.

11. REFERENCES

---

TABLE I DIFFERENTIAL EQUATION EXAMPLE

<table>
<thead>
<tr>
<th>Design</th>
<th>Design Style</th>
<th>Cycles</th>
<th>Buses</th>
<th>Regs</th>
<th>Mux</th>
<th>Muxin</th>
<th>CPU (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>min bus</td>
<td>11</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>min muxin</td>
<td>6</td>
<td>11</td>
<td>4</td>
<td>4</td>
<td>11</td>
<td>548</td>
</tr>
<tr>
<td>3</td>
<td>min cycles,</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>12</td>
<td>388</td>
</tr>
<tr>
<td>4</td>
<td>min cycles,</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>8</td>
<td>18 28</td>
</tr>
<tr>
<td>5</td>
<td>min cycles,</td>
<td>4</td>
<td>17</td>
<td>6</td>
<td>7</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>HAL [13]</td>
<td>4</td>
<td>-</td>
<td>6</td>
<td>6</td>
<td>14</td>
<td>140</td>
</tr>
</tbody>
</table>

TABLE II ELLIPTICAL FILTER EXAMPLE

<table>
<thead>
<tr>
<th>Design</th>
<th>Design Style</th>
<th>Cycles</th>
<th>Buses</th>
<th>Regs</th>
<th>Mux</th>
<th>Muxin</th>
<th>CPU (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>min cycles</td>
<td>21</td>
<td>11</td>
<td>13</td>
<td>13</td>
<td>57</td>
<td>116</td>
</tr>
<tr>
<td>2</td>
<td>min cycles</td>
<td>21</td>
<td>11</td>
<td>13</td>
<td>13</td>
<td>57</td>
<td>271</td>
</tr>
<tr>
<td>3</td>
<td>min cycles,</td>
<td>21</td>
<td>21</td>
<td>19</td>
<td>13</td>
<td>13</td>
<td>179</td>
</tr>
<tr>
<td>4</td>
<td>min cycles,</td>
<td>21</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>min cycles,</td>
<td>21</td>
<td>19</td>
<td>13</td>
<td>5</td>
<td>45</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>min bus</td>
<td>42</td>
<td>4</td>
<td>13</td>
<td>5</td>
<td>37</td>
<td>143</td>
</tr>
<tr>
<td>7</td>
<td>SPAID [5]</td>
<td>22</td>
<td>5</td>
<td>19</td>
<td>17</td>
<td>17**</td>
<td>~60</td>
</tr>
</tbody>
</table>

*SPAID cycle count is not equivalent to Schalloc's cycle count.
**SPAID muxin count does not include multiplexer inputs required for decoding register files and connecting function unit outputs to buses.

TABLE III OVEN CONTROLLER EXAMPLE

<table>
<thead>
<tr>
<th>Design</th>
<th>Design Style</th>
<th>Cycles</th>
<th>Buses</th>
<th>Regs</th>
<th>Mux</th>
<th>Muxin</th>
<th>CPU (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>min cycles</td>
<td>19</td>
<td>11</td>
<td>11</td>
<td>6</td>
<td>17</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>min cycles,</td>
<td>19</td>
<td>16</td>
<td>10</td>
<td>5</td>
<td>15</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>min cycles,</td>
<td>19</td>
<td>9</td>
<td>11</td>
<td>6</td>
<td>19</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>min muxin</td>
<td>20</td>
<td>9</td>
<td>11</td>
<td>3</td>
<td>13</td>
<td>63</td>
</tr>
<tr>
<td>5</td>
<td>min buses</td>
<td>23</td>
<td>7</td>
<td>11</td>
<td>4</td>
<td>18</td>
<td>93</td>
</tr>
<tr>
<td>6</td>
<td>ELF [4]</td>
<td>-</td>
<td>-</td>
<td>8</td>
<td>8</td>
<td>19</td>
<td>19</td>
</tr>
</tbody>
</table>