Path-Based Scheduling for Synthesis

Raul Camposano

Abstract—**In the context of synthesis, scheduling assigns operations to control steps. Operations are the atomic components used for describing behavior, for example, arithmetic and Boolean operations. They are ordered partially by data dependencies (data-flow graph) and by control constructs such as conditional branches and loops (control-flow graph). A control step usually corresponds to one state, one clock cycle, or one microprogram step. This paper presents a new, path-based scheduling algorithm. It yields solutions with the minimum number of control steps, taking into account arbitrary constraints that limit the amount of operations in each control step. The result is a finite state machine that implements the control. Although the complexity of the algorithm is proportional to the number of paths in the control-flow graph, it is shown to be practical for large examples with thousands of nodes.**

I. Introduction

THE SCHEDULING problem arises in several contexts during the synthesis of digital systems, for example, in the areas of microcode compilation [1]–[4] and high-level synthesis [5]. Briefly stated, scheduling consists of determining the time at which different jobs are performed. Even quite simple scheduling problems are *NP*-complete [6]. If we restrict ourselves to synchronous digital systems, then the "jobs" usually degenerate to single operations such as addition, multiplication, logic operations, assignment, etc., and time is given in so-called control steps which correspond to a basic machine cycle, i.e., a state in a controlling finite state machine or a microprogram step. Control steps will be called states throughout the rest of the paper. (If multiphase clocking is used, a finer granularity of time may be required). The problem is then to determine in what control state(s) each operation is executed, so that a given cost function is minimized. In microprogram compilation, the cost function is usually the number of microprogram steps required for a given benchmark, while in high-level synthesis the cost function is usually a combination of the amount of hardware, the cycle time, and the number of control states required. The amount of hardware is a measure which may include functional units such as ALU's, adders and simple gates, storage units such as memories and registers, and communication units such as buses and multiplexers.

The classical problems in scheduling are machine scheduling and project scheduling [6]–[10]. A typical formulation of the machine scheduling problem is to perform n number of jobs using m number of machines in a given (partial) order such that the corresponding cost is minimized. The techniques developed for this sort of problem are not used in synthesis, because jobs take many time units as opposed to operations which usually take only one or a few control states. Furthermore, they do not deal with loops and conditional branches.

Manuscript received January 1, 1990. This paper was recommended by Guest Editor A. Sangiovanni-Vincentelli.

The author is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

IEEE Log Number 9039382.

In microprogram optimization several techniques are used. Reference [1] discusses first-come-first-served (FCFS) scheduling, list scheduling, scheduling the critical path first, and exhaustive search using branch-and-bound techniques. FCFS and list scheduling basically schedule operations in the topological order given by the data dependencies. List scheduling uses an additional priority criterion to order operations, e.g., the path length from each operation to the end. Another method consists of scheduling first all operations on the longest path (the critical path according to data dependencies). The remaining operations are then scheduled "around" the critical path. Exhaustive search obviously guarantees the optimum but is computationally too expensive. These techniques do not deal directly with loops and with conditional branches, hence, they are often applied locally within so called basic blocks that do not contain these constructs. Transformational techniques such as trace scheduling [2] and percolation scheduling [3] rely on code motion and are global, i.e., not restricted to basic blocks. (These techniques were not intended for synthesis originally). More specialized approaches deal with pipeline scheduling of loops [4], [11].

These techniques give good results and many of them have been used in high-level synthesis. For example, extensions to list scheduling with different priority criteria are used in BUD [12] and Elf [13]. MAHA [14] in essence uses a critical path first schedule, scheduling operations that are not on the critical path in order of increasing freedom (mobility). Mobility is the difference between the earliest and latest possible schedules.

Also, transformational approaches such as trace scheduling and percolation scheduling have inspired high-level synthesis. For example, the CADDY/DSL system [15] and the V system [16] rely on moving operations between control states, considering the constraints imposed by data dependencies, loops, and conditional branches (similar to [2]). Retiming [17] can be seen as a special case of transformational scheduling.

Force-directed scheduling [18] allows the exploration of global tradeoffs between the hardware and the number of control states used. This statistical method uses the probability distribution of operations being executed in each control state to balance the amount of hardware required in each control state.

More detailed discussions of scheduling for high-level synthesis can be found in [5], [18].

The approach in this paper is quite different from all of the above. The problem addressed is scheduling for synchronous digital systems, minimizing the number of control states under given constraints such as timing and area. The scheduling problem is represented as a directed graph where the nodes represent the operations and the edges the precedence relationship. This graph may contain conditional branches (IF, CASE in a procedural language) and loops (WHILE, UNTIL in a procedural language). The most important difference to other approaches, such as critical path first and force-directed scheduling, is that these deal mainly with potential concurrency, in the sense that

they use a data-flow representation indicating the dependencies that "force serialization" and schedule, taking advantage of the freedom left to move operations among control states. In path-based scheduling, conditional branching is emphasized instead. Scheduling in this case ensures that each of the possible paths generated by conditional branches is scheduled optimally (in the minimum number of control states). This requires scheduling one operation into different control states depending on the path, a capability which no other scheduling algorithm has. In designs with numerous conditional branches, this aspect is most important. As will be described with more detail in Section IV (examples), processor design and control dominated applications are of this kind.

Instead of minimizing a cost function, path-based scheduling minimizes just the number of control states for given constraints. In practice, we have experienced that this is indeed adequate: for the applications mentioned above, it is approximately known what constraints apply, e.g., how big a chip will be and what cycle times can be expected with a certain technology. "Design space exploration" in these cases will consist of scheduling only a few dozen designs, changing slightly the approximately known constraints—very much like designs that are done in practice.

The constraints considered are of two types. Intrinsic constraints arise from the fact that, in synchronous systems, hardware can be used only once in each control state, e.g., a register can receive only one new value, a port can transmit or receive only one value. External constraints, such as the available amount of hardware, must be specified by the designer.

The scheduling algorithm proceeds as follows. Each possible path is scheduled independently in an optimal fashion, in the sense that the minimum number of control states for the given constraints is found. Then the schedules for each path are overlapped, again in an optimal way. Loops and conditional branches are handled as an integral part of the problem. We call this as-fast-as-possible (AFAP) scheduling. The technique is an exact solution to the scheduling problem formulated in the Yorktown Silicon Compiler (YSC) [19]. In the YSC it was solved heuristically, scheduling paths individually using backtracking, and overlapping them in a greedy fashion. Although the worst-case complexity of AFAP scheduling is *NP*, we have exercised it for large real designs with thousands of operations. Faster heuristics that yield suboptimal results are also suggested.

There are two other approaches known to the author that also deal with conditional branching explicitly. Bridge [20] uses a Boolean condition to identify when operations are activated. Operations in mutually exclusive conditional branches can be identified by these conditions and scheduled (heuristically) in the same state sharing hardware. A more systematic approach also based on such conditions extends list scheduling to take into account mutually exclusive operations [21]. Both approaches do reduce the number of states required for a schedule, but they do not schedule operations in more than one state nor do they minimize the path length globally. Some comparisons in Section V will clarify this.

The paper is structured as follows. The next section states the scheduling problem formally. The scheduling algorithm is presented in Section III, including the representation of constraints. The main issues introduced are the treatment of loops and conditional branches, and scheduling in the minimum number of control states. Heuristics to speedup processing are also suggested. Section IV gives results for several examples. Section V compares results to other scheduling techniques. The paper ends with conclusions and an outlook.

II. Definition of the AFAP Scheduling Problem

A behavioral description of the problem to schedule is given by the *control-flow* directed graph $B = (V, E)$. The nodes $v \in V$ represent *operations* to be scheduled, and the edges give the *precedence relation*, i.e., $(v_i, v_j) \in E$ iff v_i is an immediate predecessor (called just predecessor) of v_j. v_j is called an immediate successor (or just successor) of v_i. The interpretation of B is imperative: an operation is executed after one of its predecessors is executed.

If a node v has more than one successor, v is said to be a *conditional branch*. Only one of the successors will be executed. The decision of which successor is chosen is taken according to a *condition* predicate $cond(v, v_i)$ attached to the corresponding edge. If $cond(v, v_i)$ is true, then v_i is executed after v. The conditions on outgoing edges from conditional branches must be all mutually exclusive. Conditions are arbitrary Boolean functions that can be directly derived from conditional constructs such as IF, CASE, WHILE, and UNTIL in procedural languages.

Notice that mapping of procedural (imperative) languages, such as C or Pascal, onto this graph is trivial using the order of operations given in the program. Fig. 1 gives an example. The VHDL program corresponds to the control-flow graph. The node numbers are indicated as a comment (starting with "--") in the program. Nodes 1 and 2 correspond to signal assignments; in our example corresponding to output ports (declared in the entity statement). Nodes 4 and 7 are conditional branches, their outgoing edges are labeled with the corresponding conditions. Nodes 5, 8, and 10 are variable assignments. Nodes 3 and 9 are additions (merged with a signal/variable assignment, respectively). Node 6, finally, is a dummy node corresponding to the "end if" statement. The program is assumed to loop endlessly, i.e., node 1 executes after node 10.

The control-flow graph has a unique *first operation* v_1 at which execution starts; in the example this is node 1. It should be possible to reach all other operations from v_1, otherwise there are dead operations in B that can never be executed.

A longest path through the control-flow graph is a path starting at v_1 and ending at an operation with no successors. Repetition of operations are not considered, i.e., cycles in the graph are traversed just once for longest path computation. The set of all longest paths is denoted as $\{p_l\}$. It represents all different operation sequences (again, excluding repetition of cycles) that the specified behavior allows. As an example, consider a processor: each longest path corresponds to the execution of an instruction (if we ignore exceptions).

The AFAP scheduling problem is then formulated as follows. Given $B = (V, E)$ and a set of constraints, schedule all operations $v \in V$ such that all possible longest paths $\{p_l\}$ execute in the minimum number of control states and all constraints are met.

Several comments are necessary at this point. The exact representation of constraints will be introduced in the next section. In the absence of any external constraints on area (the amount of hardware) and time (the maximal delay that is allowed within one control state), no more data than the precedence relation of the operations and their data dependencies are needed to derive intrinsic constraints such as writing a register only once per control state. If there are constraints on area and time, then an

```
entity prefetch is
  port (branchpc, ibus  : in bit32;
        branch, ire     : in bit;
        ppc, popc, obus: out bit32);
end prefetch;

architecture behavior of prefetch is
begin
  process
   variable pc,oldpc : bit32 := 0;
  begin
    ppc  <= pc;              --1
    popc <= oldpc;           --2
    obus <= ibus + 4;        --3
    if (branch = '1')        --4
     then
      pc := branchpc;        --5
    end if;                  --6
    wait until (ire = '1'); --7
    oldpc := pc;             --8
    pc   := pc + 4;          --9,10
  end process;
end behavior;
```

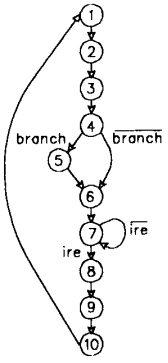


Fig. 1. Behavioral description example.

operator library that contains the delay and the area used by each operation is required. Notice that it is difficult to estimate these numbers accurately at this level.

Without loss of generality, we may assume that each operation can be executed in one control state. If this is not the case, the operation has to be split into several operations, which may be done automatically in the presence of a time constraint.

Scheduling has been formalized using a given precedence relation of operations. This precedence relation may be derived partly from the data flow, ordering only such operations that have data dependencies. However, the precedence relation always contains procedural (imperative) elements such as conditional branches and loops which are difficult to represent as pure data flow. They are represented naturally in the control flow. Operations that can be executed in parallel may be clustered in one node or ordered arbitarily. If they are clustered in one node, they will be always scheduled in one control state (they will be treated as one larger operation). If they are ordered, they may be scheduled in one or more control states; if they are scheduled in more than one control state, the given order will be maintained. So B may reflect a given ordering of operations chosen by the designer, or may be obtained from the data dependencies, keeping only the necessary orderings.

The output of AFAP scheduling is the exact specification of a finite state machine that implements the control of B.

III. Algorithm

In this section, the algorithm for AFAP scheduling is given. It involves keeping all paths in the control-flow graph and several *NP*-complete steps. At the end, substantial simplifications are suggested. The algorithm consists of four main steps.

1) Transforming the contol-flow graph B into a directed acyclic graph (DAG) and keeping lists for the loops.

2) All paths in the DAG are scheduled AFAP independently, according to the data-flow constraints in each path.

3) The schedules of step 2) are overlapped in a way that minimizes the number of control states.

4) The finite state machine for control is built.

A. Loops

Let v_F be the first operation in a loop body and v_L the last one. Each loop L is "broken," removing the feedback edge (v_L, v_F), and storing v_F, v_L and the condition c of the feedback edge (v_L, v_F) (Fig. 2).

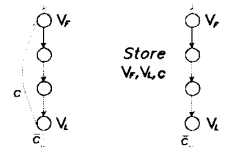


Fig. 2. Transformations for loops.

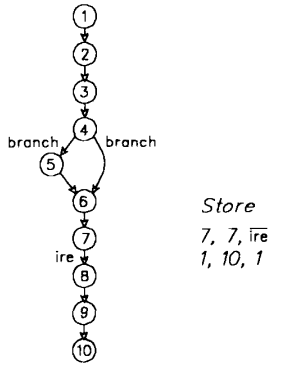


Fig. 3. Prefetch example after loop elimination.

The result is a DAG, and a list of removed feedback edges with their conditions. The intention is to allow the execution of a loop body only once. If the loop body has to be repeated, execution gets trapped in v_L, since c is false. We will schedule this problem optimally and then add transitions in the control finite state machine to repeat loops an arbitrary number of times. Any kind of loop unfolding or optimization must be done prior to scheduling; we do not deal with pipeline scheduling of loops such as in [11], [22].

The transformation given above applies to loops that have the exit condition at the end (UNTIL loops). A similar transformation applies to loops with the exit condition at the beginning (WHILE loops).

Fig. 3 gives the result of eliminating loops for the example of Fig. 1. For the loop at node 7, node 7 represents v_F and v_L. For the endless outer loop, node 10 represents v_L and node 1 represents v_F. The conditions on the feedback edges are $\overline{ire}$ and 1.

Notice that loops are detected easily during syntax analysis of structured languages so that the control-flow graph can be marked accordingly. Thus in practice, loop detection presents no problems. For a general graph, finding the minimum set of edges that will break all cycles is *NP*-hard. This can be done using Johnson's algorithm [23].

B. AFAP Scheduling of Single Paths

The idea is to schedule each path AFAP independently. Paths arise from the conditional branches in B. A path corresponds to one possible execution sequence, so the number of different paths is a measure of how many different functions a design can perform. Although the number of paths in a graph can grow worse than exponentially, in practice we have found on the order of 10^3 paths for the execution unit of a microprocessor.

We first compute all paths, then the constraints for each path, and finally an AFAP schedule for each path.

1) All longest paths in B are computed, i.e., paths that start

at the first node or at nodes v_F, and end with nodes with no successors. Efficient algorithms for path computation use the depth-first search construction of the transitive closure [24]. Remember that a path represents a possible sequence of operations, hence, paths which start at loop beginnings v_F must also be considered (loop bodies may be repeated, so there is a sequence that starts at the first operation of the loop body v_F). In the Prefetch example, there are 3 paths: *path1* = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, path2 = {1, 2, 3, 4, 6, 7, 8, 9, 10}, and path 3 = {7, 8, 9, 10}.

2) For each path, all constraints are computed. Constraints are the following

- Variables can be assigned only once in one control state. Notice that variables may have been disambiguated using global data-flow analysis (replicating them so that they are assigned only once) [25]. In this case, no constraints will be generated. Variables explicitly meant to be registers should not be disambiguated, thus allowing them to generate constraints.
- IO ports can be read or written only once in one control state.
- Functional units can be used only once in a control state. This constraint is only relevant if the amount of hardware is constrained. In this case, the operations that can be scheduled in one control state are limited by the available hardware.
- The maximal delay within one control state limits the number of operations that can be chained (that feed data to each other and are executed in the same control state).

The amount of storage (registers and memories) and communication (buses, multiplexers) is not constrained presently. Notice that the amount of storage and communication cannot be influenced significantly by the schedule, e.g., [26] reports randomly distributed changes from 13 to 16 registers for schedules varying from 17 to 31 states for the filter from [27] and similar results for two other examples. Obviously, storage and communication can be optimized during allocation.

The constraints are kept as sets of operations $\{v\}$, so that if any $v \in \{v\}$ is the first operation in the next state, the constraint is met. For one path, the nodes are totally ordered. Thus each constraint (set of nodes) can be interpreted as an interval. Fig. 4 illustrates the above concepts for path 1 of the Prefetch example. Constraint 1, for instance, is generated due to the fact the variable pc is written twice (we assume that pc is not to be replicated). The constraint indicates, that path 1 has to be "cut" between operations 6 and 10, so that the two assignments to pc are not in the same control state.

It is easy to see that a constraint on the amount of hardware (or on the maximum cycle time) just generates a series of intervals obtained by adding sizes (times) along the path, starting at each operation, until the constraint is violated. In the example Prefetch, if the maximum functional unit area is limited to 100 cells and the incrementer necessary for the "+4" increment uses 80 cells (so only one incrementer can be used), then constraint 2 is generated (Fig. 4). It states that operations 3 and 9 must be scheduled in different control states.

If the cycle time is constrained to T, then the execution times of the operations are added along the path starting at operation 1, according to the data dependencies. At some operation v this addition will be larger than T, and a constraint must be generated covering the interval from operation 2 to operation v, meaning that the path has to be cut at one of these points to meet the cycle time T. The process is then repeated, starting the addition at operation 2, 3, etc. All these constraints must obviously be met. In the example no such constraints are generated, because the operations do not have data dependencies (i.e., they may all be executed in parallel). External constraints such as protocols, that require, for example, operations to be executed in successive cycles, are formulated as intervals with just one operation.

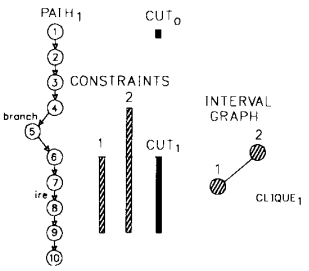


Fig. 4. Constraints and interval graph for one path in the Prefetch example.

3) The interval graph for the set of constraints of each path is formed, and a minimum clique covering is computed (Figs. 4 and 5). In the interval graph each node corresponds to an interval and edges indicate that the corresponding two intervals overlap. A clique is a complete subgraph, with all possible edges. A minimum clique covering is a minimal number of cliques, so that each node is in one clique.

By construction, it is clear that the solution to the minimum clique covering gives the minimum number of control states. A "cut" corresponds to each clique. It represents the possible operations at which a state starts. States are ordered along one path. In addition, a cut for the first operation along the path is added (the first state on the path starts with the first operation). The cuts give the minimum number of control states to execute this path (a state starts at a cut which corresponds to a clique, and a minimum clique covering is generated). Since each interval will be in one clique, all constraints are met.

In the example of Fig. 4 only one clique exists, and the two intervals indeed overlap. So only two cuts are generated, cut_0 for the first operation, and cut_1 for the clique corresponding to the two constraints. A more complicated example with two cliques is given in Fig. 5. Cuts are kept as the maximal set of overlapping operations, indicating all the possible positions for that cut, for example, the cut_1 in Fig. 4 will consist of operations 6, 7, 8, 9, indicating that the next state may start at any of them. Cut_2 in Fig. 5 is also an interval.

The above algorithm obtains the AFAP schedule for each path individually. Besides the fact that the number of paths may explode, all steps can be performed efficiently. Clique covering is in general *NP*-complete, but can be computed efficiently in a single pass through the path for interval graphs [28] (also known as "left edge" algorithm). Notice that there are many possible minimum clique coverings, e.g., in Fig. 5 one clique could contain only node 2 and the other one nodes 1, 3, 4, 5. All solutions obviously correspond to the minimum number of control states.

C. Overlapping of Paths

To find the minimum number of states for all paths, the schedules for each path must be overlapped. Again, the observation that overlapping cuts can be merged helps to formulate the problem formally. A graph is formed, such that the nodes correspond to the cuts (set of nodes) defined in the previous step

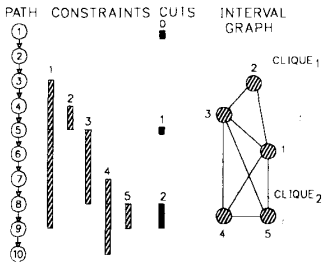


Fig. 5. Constraints and interval graph for a more complex example.

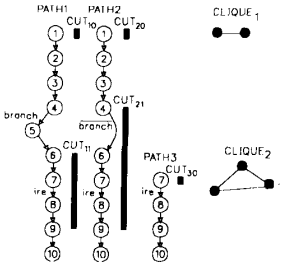


Fig. 6. Overlapping cuts for different paths.

and edges join nodes corresponding to overlapping cuts (the intersection of the sets of operations is $\neq \phi$). A minimum clique cover of this new graph will clearly give the minimum set of cuts that fulfills the fastest schedule for each path, and thus the minimum number of control states (control states start at cuts).

The AFAP schedules for the three paths of the Prefetch example are shown in Fig. 6. Cuts are subindexed with the path number first and then with an increasing index starting at 0. Cut_{10} and cut_{11} correspond to the cuts discussed in Fig. 4. Cut_{20} and cut_{30} correspond to the initial operation in paths 2 and 3, respectively. Cut_{21} is generated by the area constraint along path 2 and avoids the two increment operations (3 and 9) that are scheduled in the same control state. There is the trivial clique containing the cuts representing the first operation 1, in the example clique 1. All other cuts overlap at operation 7, forming clique 2. This clique represents the start of the second state at operation 7. A clique may still contain more than one overlapping operation. In this case one operation is selected as "the cut" for the construction of the control automaton. Any operation in the set of overlapping operations can be selected. Operations easily related to the original specification, such as conditional branches and joins, are a good choice.

Since this graph does not seem to have any special property, the problem is most likely *NP*-complete. It is not an interval graph due to the fact that operations in alternative branches of a conditional branch are not ordered (this is not seen in the example, but would be the case for an operation x between operations 4 and 6 in path 2: operation x and operation 5 would not be ordered). Although several heuristics for the clique covering problem exist [29], we implemented an exact solution by exhaustive search and have not experienced problems with excessive runtimes yet (see Section IV).

Notice that minimizing the number of cuts only minimizes the number of control states. Not solving this optimally will not change the fact that the schedule for each path is the fastest possible.

D. Control Finite State Machine

So far we have obtained the cut positions for all paths. The control automaton is built by merging as many path segments as possible into one state. We now construct a finite state machine that implements the control for the schedule.

Let the set of paths be $\{p_l\}$. Let the ordered set of cutting points for a path p_l be $\{cut_{lj}\}$. Let the ordered set of last operations for control states be $\{ce_{lj}\} = \{(v \in p_l \mid v = pred(cut_{l,j+1})), v_{lL}\}$, i.e., the ordered set of immediate predecessors of cutting points along the path, including the last operation v_{lL} of the path (clearly, the last operation on a path will also be the "last" operation in a control state). Let the path interval $r_{lm} = [cut_{lm}\ ce_{lm}]$ contain operations on path p_l starting at cut_{lm} and ending at ce_{lm}. By construction, all operations in a path interval can be scheduled in one control control state.

The intervals for the Prefetch example are shown in Fig. 7. Path one, for instance, is divided into the path intervals $r_{10} = \{1, 2, 3, 4, 5, 6\}$ and $r_{11} = \{7, 8, 9, 10\}$ (the numbers in the set represent the operations or nodes in the graph). The cuts (first operations) are $cut_{10} = 1$ and $cut_{11} = 7$. The last operations are $ce_{10} = 6$ and $ce_{11} = 10$.

Let a *state* be denoted by s_k. The set of all operations that are (conditionally) executed in s_k is said to be *scheduled* in s_k. A state transition is denoted $s_i \rightarrow s_k$ ($cond_{ik}$ = *Boolean expression*), with $cond_{ik}$ being the condition that enables this state transition given as a Boolean expression.

The control automaton is constructed in 3 steps.

1) Overlapping of intervals to form states: All intervals starting with the same operation cut_{lj} can be merged into one state and their operations are all scheduled in this state. Let the necessary states be s_k, $k = 1, 2, 3 \cdots$. Intervals with the same first operation are trivial to identify. Conditional branches within one state are handled by combinational logic (the enable signals). In the Prefetch example there will be two states necessary, s_1 starting with operation 1 and s_2 starting with operation 7 (Fig. 7).

2) Construction of the state transitions: For each pair of states s_i, s_k such that s_i is "previous" to s_k, i.e., for some ce_{lm} scheduled in s_i, cut_{ln} scheduled in s_k, $ce_{lm} = pred(cut_{ln})$, one state transition is added:

$$s_i \rightarrow s_k(cond_{ik}).$$

Also, state transitions to allow the repetition of loops must be added: $s_i \rightarrow s_k(cond_{ik})$, where the last loop operation v_L is scheduled in s_i and the corresponding first loop operation v_F is the first operation scheduled in s_k. Conditions are constructed in the next step.

In Fig. 7, there is a state transition $s_1 \rightarrow s_2$ because operation 6 (the last in s_1) is a predecessor of operation 7 (the first in s_2). The loops add transitions $s_2 \rightarrow s_2$ and $s_2 \rightarrow s_1$.

3) Construction of state transition conditions: An operation in the control-flow graph is executed, if the previous operation was executed and the condition on the incoming edge is true. Thus along a path, an operation is executed, if all previous conditions were true, i.e., the AND of these conditions is true. If an operation has more than one predecessor, the conditions along those paths must be ORed. A state transition $s_i \rightarrow s_k$ takes place, if a last operation ce_{lm} scheduled in s_i is executed, such that $cut_{ln} = succ(ce_{lm})$ is the first operation scheduled in s_k, and the condition on the edge (ce_{lm}, cut_{ln}) is true. Hence, the condition is

$$cond_{ik} = \bigvee_{lm} cond(ce_{lm}, cut_{ln}) \bigwedge_{j} cond(v_{jlm}, v_{(j+1)lm})$$

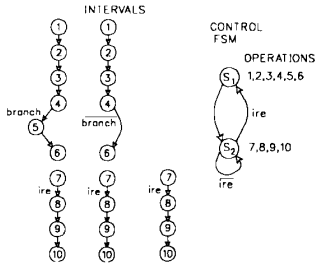


Fig. 7. Building the control finite state machine for the Prefetch example.

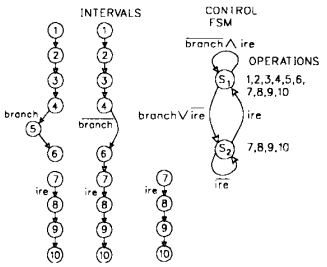


Fig. 8. Building the control finite state machine for Prefetch with no area constraints.

with lm ranging over all intervals r_{lm} scheduled in s_i, and j ranging over all operations of the interval except for the last.

In Fig. 7, for example

$$cond_{12} = branch \vee \overline{branch} = 1,$$

$$cond_{21} = ire,$$

$$cond_{22} = \overline{ire}.$$

The proof that the above algorithm indeed yields a control automaton that implements the behavior of the original control-flow graph is straight forward (by construction) and is omitted here. The proof that all transformations used are behavior preserving can be found in [30].

Notice that conditional branches can be scheduled completely in one state (e.g., operations 4, 5, and 6 in Fig 7). This is usually desirable, if the delay in the data path is larger than in the control. However, if a new state for each conditional branch is desired, it can be easily accomplished by adding constraints that force new states.

The execution of operations must be controlled by combinational logic. For this, an enable signal e_v is defined for each operation v. This signal is used to enable register loading, to select appropriate multiplexer inputs, to enable bus drivers, etc. Reasoning in the same way than deriving the state transition conditions, the enable signal is defined as

$$e_v = \bigvee_k s_k \wedge \bigvee_{lm} \bigwedge_j cond(v_{jlm}, v_{(j+1)lm})$$

where s_k is used as a Boolean variable that is "1" if the control automaton is in state s_k and "0" otherwise, k ranges over all states into which v is scheduled, lm ranges over all intervals r_{lm} scheduled in s_k, and j ranges over all operations along the interval that are previous to v.

In Fig. 7 for example, operation 5 has the condition $e_5 = branch \wedge s_1$, indicating that the operation (loading of pc, see Fig. 1) is to be executed in state 1 if *branch* is true. Operations 8–10 have an enable equal to $ire \wedge s_2$. The Boolean expressions for enable signals should be minimized on the fly, taking advantage of the structure of the control-flow graph, e.g., conditions of a particular conditional branch simplify to "1" after closing the branch.

Since each path is scheduled independently for speed, one operation may be scheduled in more than one state. Since states are mutually exclusive, this does not create problems. Consider, for example, Prefetch without the area constraint (Fig. 8). In this case there is no constraint along path 2 and all the operations along this path can be scheduled in a single state s_1. A second state is needed for operations 7, 8, 9, 10 which are on a second path; these operations are thus scheduled in both states. It is easy to see that the given FSM is indeed the optimal control for this case. If branch is false and *ire* is true, only one cycle is needed for the complete execution! The enable signal for operations 8, 9, 10 (which are scheduled in two states) is equal to $(s_1 \vee s_2) \wedge ire$.

AFAP scheduling is complex and computationally intensive. A simpler heuristic is to examine one path at a time, and to cut the complete graph just according to this one path. Cutting the complete graph consists of removing one edge in the path under consideration; all other paths containing this edge are cut as well, thus reducing considerably the effort for computing cuts and making the overlapping of paths unnecessary. This technique, combined with a greedy cutting criteria for a path that cuts whenever one constraint is violated, was implemented in the YSC. An even simpler method consists of traversing the graph only once doing a depth-first search, and cutting the graph whenever a constraint is found.

IV. Results

The examples used in this section are benchmarks from the 1989 Workshop on High-Level Synthesis [31], with the exception of EXE. COUNTER is a 4-b counter. GCD is a greatest common divisor calculation. PREFETCH is an instruction fetch unit for a microprocessor (different from the example used in the previous figures). TLC is a traffic light controller excluding the necessary timer (the timer is called). OTPT writes data onto a bus with a handshake which is used in KALMAN. KALMAN is a Kalman filter without the bus interface. TX8251 is the transmitter part of an Intel 8251 UART chip; HUNT implements the hunt mode of the Intel 8251. EXE is the execution unit for a streamlined microprocessor with over 100 instructions (not given in the benchmark). The examples were written in VHDL [32], V [16], and ISPS [33]. The graphs were generated using the V compiler, examples in VHDL and ISPS were manually translated to V.

AFAP scheduling was implemented in APL and APL2. All examples were run on an IBM 3090/200 machine under CMS VM/SP 4.2. APL is interpreted, which accounts partly for long runtimes. All examples could be executed with less than 7Mbytes of memory.

Table I gives the execution times for different steps in the algorithm. The *compile* phase is included for informative purposes only. Compiling includes the conversion into the internal representation, global data-flow analysis to disambiguate variables (replicating them until single assignment is obtained) and grouping bit arrays into fields that are always used together (and can be treated as units). Detection of *loops* is syntax driven and correspondingly very fast. Computation of *constraints* constructs intervals. Notice that constraints can be computed before

TABLE I
EXECUTION TIMES IN CPU SECONDS FOR SEVERAL EXAMPLES

Design	*Compile*	*Loops*	*Constr.*	*Paths*	*Cuts*	*Cliques*	*Control*
COUNTER	0.74	0.01	0.04	0.02	0.01	0	0.07
GCD	0.95	0.03	0.05	0.05	0.01	0	0.15
PREFETCH	1.12	0.02	0.04	0.06	0.02	0.01	0.32
TLC	0.64	0.02	0.04	0.05	0.06	0.05	0.42
OTPT	0.45	0.02	0.02	0.02	0.01	0.01	0.13
KALMAN	6.6	0.1	0.3	2.5	8.1	4.7	11.8
HUNT	2.7	0.03	0.1	0.2	0.02	0	0.94
TX8251	5.3	0.1	0.2	9.2	55.1	177	116
EXE	146	0.2	5.9	53.6	0	0	10.5

TABLE II
RESULTS FOR SEVERAL EXAMPLES

Design	*Nodes*	*Edges*	*Loops*	*Constr.*	*Paths*	*Cuts*	*Cliques*	*States*	*Exec.*
COUNTER	12/21	14/19	1	0	3	3	0	1/1	1/1
GCD	15/24	18/21	3	0	7	7	0	2/4	1/2
PREFETCH	18/32	22/25	3	1	9	11	1	4/9	1/3
TLC	14/25	16/18	3	4	19	34	3	8/18	2/6
OTPT	9/16	11/8	3	2	5	7	1	2/4	1/2
KALMAN	100/168	111/192	11	19	258	839	7	23/115	1/17
HUNT	40/67	46/64	6	2	28	28	0	6/25	1/6
TX8251	92/141	111/144	10	15	1594	6319	11	22/112	2/18
EXE	808/1162	950/1936	1	0	1596	1596	0	1/1	1/1

all paths are computed—they are later projected on the paths by simply determining if all operations of a constraint are on a particular path. Constraint computation is fast and is by no means a bottleneck. *Paths* computation can be slow for large graphs (e.g., EXE). Then all *cuts* on all paths are computed, which corresponds to finding the cliques on the interval graphs. These cuts are then overlapped, and the *cliques* are computed using exhaustive search. This is also quite fast, because the graphs tend to have few edges and thus a large number of trivial cliques containing only one or a few nodes (Fig. 4). Finally, the finite state machine for *control* is built. This tends to be one of the slowest steps due to the necessary bookkeeping and the construction of conditions. Some Boolean optimization capabilities were built into the algorithm, to keep expressions for conditions small.

Runtimes are reasonably short, even for large examples. Implementing the algorithm in a compiled, lower level language such as C would result in a significant speed-up.

The results in Table II again show the different steps of the algorithm. The *nodes* and *edges* columns give the size of the problem, giving both the control-flow and the data-flow graphs (control-flow/data-flow). The amount of *loops* usually includes one external infinite loop. The number of constraints (*Constr.*) does not include external constraints; the examples were not limited to a certain size or cycle delay. The total number of *paths* is given next. *Cuts* include the obvious cut at the first operation (see Section III). The number of *cliques* does not include trivial cliques with only one operation.

The column *states* gives the number of states followed by the number of state transitions of the finite state machine constructed for control. The examples KALMAN, TX8251, and EXE are hierarchical and contain calls to other modules which may generate the need for additional states. This paper does not discuss hierarchical design issues, the reader is referred to [19], [30].

The last column labeled *Exec.* gives the number of cycles to execute the behavior without repeating loops. The first number gives the shortest sequence that returns to the initial state. The second number gives the longest sequence that returns to the initial state, without repeating states. This gives an idea of the performance, e.g., the number of cycles for the longest and shortest instructions in a microprocessor. Notice that the shortest sequence is often of length one, due to the fact that loop bodies are scheduled in the same state as the following operations, so that they can be conditionally executed in the same state if the exit condition is met.

We also compared these results with the YSC heuristic mentioned at the end of the previous section. In many cases the same results were reached. The exceptions were KALMAN scheduled in 30 states, TX8251 scheduled in 29 states, and HUNT scheduled in 8 states. Execution times for heuristic 1 and AFAP scheduling cannot be compared in a meaningful way, since they were implemented in different environments.

V. COMPARISONS

The comparison of AFAP scheduling to other scheduling techniques, such as force directed scheduling [18] or critical path first scheduling [14], is difficult because the objectives of these techniques are different. As already stated, AFAP scheduling deals mainly with applications with many conditional branches and loops that emphasize fast schedules. Such applications are common in processor design (as our experience tells us), in control dominated applications [34] and whenever short schedules are important. "Classical" scheduling in high-level synthesis, however, emphasizes much more applications where potential parallelism is high and the resulting schedules are relatively long to obtain a reasonably sized data path. Some comparisons summarized in Table III may clarify these points further. In Table III, *Adds* is the number of adders, *Subs* is the

TABLE III
COMPARISONS WITH FORCE DIRECTED AND CRITICAL PATH FIRST SCHEDULING

Design	*Method*	*Adds*	*Subs*	*Muls*	*States*	*Paths*	*Chain*
Filter	Path	2	—	1	13	13/13	3
	Force	2	—	1	19	19	1
Filter	Path	3	—	1	9	9/9	3
	Force	3	—	1	18	18	1
Diffeq	Path	1	1	2	4	4/1	1
	Force	1	1	2	4	4	1
mahal	Path	2	3	—	4	3/1	5
	Crit	2	3	—	4	4	3
mahal	Path	1	1	—	9	5/2	2
	Crit	1	1	—	8	8	2

number of subtracters, *Muls* is the number of multipliers, *States* is the number of states, *Paths* is the longest/shortest path (in number of states) to execute the behavior without repeating loops, and *Chain* is the maximum number of operations chained.

The *filter* example is the well-known filter from [27]. The problem in this case degenerates to just one path (no conditional branches). Path directed scheduling uses less states than force directed scheduling [18] because it chains operations. Notice that arithmetic operations can be chained with very little time penalty, i.e., usually the delay necessary to compute just a 1-b operation in carry chain implementations [35]. Force-directed scheduling uses pipelined multipliers that take 2 cycles with a latency of 1, while path-directed scheduling uses multipliers that take one cycle.

The *Diffeq* example is the differential equation from [36]. Force-directed scheduling and path-directed scheduling essentially yield the same result. There is, however, a path that takes only one state to complete: if the exit condition of the loop is true at the start, the computation takes one cycle.

The third example is used in MAHA (scheduling critical path first) [14]. The description used for path-directed scheduling assumes that all the forks are conditional branches. In this case it can be seen that the path-directed schedule needs less states to complete, even though an extra state was needed for the case with one adder and one multiplier. Again, the chained operations are all arithmetic so that the cycle time is only slightly affected. The CPU times for all examples is negligible (below 1 s on an IBM PC/RT).

Unfortunately, little results have been reported on the benchmarks given in Tables I and II. Among those, the results for the i8251 benchmark as reported in [20] are summarized in Table IV. The description is given in three modules, the main, the receiver, and the transmitter. *Tran* means number of transistors, in the case of path-directed scheduling excluding registers. *Regs* gives the number of registers (in bits). *States* is the total number of states and *Paths* gives the longest and shortest paths. The combinational logic in this example is rather simple containing only a few functional units of the complexity of a 4-b subtracter and an 8-b parity generator. It was minimized using the YLE [37]. The execution times (*CPU* on a 3090/200) exclude logic synthesis for the case of path-directed scheduling. Naturally it is difficult to compare these results directly since the design environments are different.

A comparison with Wakabayashi's and Yoshimura's example [21] is given in Table V. Only one adder, one subtracter, and one comparator are used. The column labeled "Paths" gives

TABLE IV
COMPARISON WITH BRIDGE

Design	*Method*	*Tran*	*Regs*	*States*	*Paths*	*CPU*
main	Path	488	48	10	8/1	8.0
	Bridge	3382		27		236.3
rcvr	Path	1176	38	28	20/2	22.4
	Bridge	3840		19		270.3
xmtr	Path	1236	18	22	18/2	357.6
	Bridge	3600		37		300.2

TABLE V
COMPARISON WITH WAKABAYASHI'S METHOD

Design	*Method*	*Adds*	*Subs*	*Comp*	*States*	*Paths*	*Chain*
Waka	Path	1	1	1	8	7/3/4.75	2
	Waka	1	1	1	7	7/5/5.75	2
	Path	ALU	ALU	1	6	6/3/4.25	2

the number of states necessary to execute the longest path, the shortest path, and the average over all paths assuming equal probabilities of taking each branch. Path-directed scheduling uses one extra state, but it executes the shortest path in only 3 states rather than in 5 states, and it uses one cycle less on average. An interesting case arises if, instead of the adder and the subtracter, two ALU's capable of both addition and subtraction are used. Now also two additions can be chained, reducing the length of the critical path to only 6 states and the average number of states for the execution to 4.25. In addition, overlapping of states allows to reduce the total number to only 6 states.

VI. CONCLUSIONS AND OUTLOOK

This paper presented a new scheduling method that combines several unique capabilities. AFAP scheduling is path-based and obtains the minimum number of control states for all execution paths. To allow the optimal scheduling of all execution paths, operations may be scheduled into several states (states overlap). Arbitrary constraints can be taken into account. Loops and conditional branches are handled as an integral part of the method. The output is the exact specification of a control finite state machine that implements the AFAP schedule.

Although exact methods are used, including the computation of all paths in a graph and the solution of a minimum clique covering problem, designs of several thousand nodes (the equivalent of a complete microprocessor) could be run. Since the representation level for a design is arbitrarily high (operations may be Boolean or arithmetic or of any complexity), large problems can be handled. Results and comparisons with other scheduling methods are encouraging.

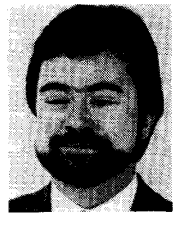
The main limitations at present are the lack of pipeline scheduling capabilities and the fact that the order of operations must be chosen in advance (although ordered operations may still be scheduled in the same control state in parallel). These are obvious topics for further research.

REFERENCES

[1] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallet, "Some experiments in local microcode compaction for horizontal machines," *IEEE Trans. Comput.*, vol. C-30, July 1981.

[2] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, July 1981.
[3] A. Nicolau, "Percolation scheduling: A parallel compilation technique," Ithaca, NY: Dept. of Computer Science, Cornell University, TR 85-678, May 1985.
[4] A. K. Uht, "Requirements for optimal execution of loops with tests," in *Proc. ACM Int. Conf. Supercomputing*, July 1988.
[5] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proc. IEEE*, vol. 78, pp. 301-318, Feb. 1990.
[6] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, "Recent developments in deterministic and stochastic scheduling: A survey," in M. A. H. Dempster, J. K. Lenstra, A. H. G. Rinnooy Kan, ed., *Deterministic and Stochastic Scheduling*. Dordrecht, Germany: D Reidel, 1982.
[7] A. H. G. Rinnooy Kan, *Machine Scheduling Problems—Classification, Complexity and Computations* The Hague, The Netherlands: Martinus Mijhoff, 1976.
[8] M. J. Gonzalez, "Deterministic processor scheduling," *ACM Comput. Surveys*, vol. 9, no. 3, Sept. 1977.
[9] R. Bellman, A. Esogbue, and I. Nabeshima, *Mathematical Aspects of Scheduling and Applications*. New York: Pergamon Press, 1982.
[10] S. French, *Sequencing and Scheduling*. Chichester, U.K.: Ellis Horwood Ltd., 1982.
[11] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proc. MICRO-20*, Dec. 1987.
[12] M. C. McFarland, "Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions," in *Proc. 23rd Design Automation Conf.* Las Vegas, June 1986, pp. 474-480.
[13] E. F. Girczyc and J. P. Knight, "An ADA to standard cell hardware compiler based on graph grammers and scheduling," in *Proc. ICCD'84*, Oct. 1984.
[14] A. C. Parker, J. Pizarro, and M. Mlinar, "MAHA: A program for datapath synthesis," in *Proc. 23rd Design Automation Conf.*, Las Vegas, June 1986, pp. 461-466.
[15] R. Camposano and W. Rosenstiel, "Synthesizing circuits from behavioral descriptions," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 171-180, Feb. 1989.
[16] V. Berstis, "The V Compiler: Automatic hardware design," *IEEE Design & Test Comput.*, pp. 8-17, Apr. 1989.
[17] C. E. Leiserson, F. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in R. Bryant, ed., *Third Caltech Conference on VLSI*. Rockville, MD: Computer Science, 1983.
[18] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 661-679, June 1989.
[19] R. Camposano, "Structural synthesis in the Yorktown silicon compiler," in C. H. Sequin, ed., *VLSI'87, VLSI Design of Digital Systems*. Vancouver: North-Holland, 1988, pp. 61-72.
[20] C. J. Tseng, R. W. Wei, S. G. Rothweiler, M. Tong, and A. K. Bose, "Bridge: A versatile behavioral synthesis system," in *Proc. 25th ACM/IEEE Design Automation Conf.*, Anaheim, CA, June 1988, pp. 415-420.
[21] K. Wakabayashi and T. Yoshimura, "A resource sharing control synthesis method for conditional branches," in *Proc. ICCAD'89*, Santa Clara, CA, Nov. 1989, pp. 62-65.
[22] G. Goossens *et al.*, "Loop optimization in register-transfer scheduling for DSP systems," in *Proc. 26th Design Automation Conf.*, June 1989.
[23] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM J. Comput.*, vol. 4, no. 1, pp. 77-84, Mar. 1975.
[24] Y. E. Ionnidis and R. Ramakrishnan, "Efficient transitive closure algorithms," Computer Science Tech. Rep. #765, Univ. of Wisconsin-Madison, Apr. 1988.
[25] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1986.
[26] L. Stok, "Interconnect optimization during data path allocation," in *Proc. EDAC'90*, Glasgow, Scotland, Mar. 1990, pp. 141-145.
[27] P. Dewilde, E. Deprettere and R. Nouta, "Parallel and pipelined VLSI implementation of signal processing algorithms," in S. Y. Kung. H. J. Whitehouse, T. Kailath, ed., *VLSI and Modern Signal Processing*, Englewood Cliffs, NJ: Prentice Hall, 1985, pp. 258-264.
[28] S. Even, *Graph Algorithms*. Rockville, MD: Computer Science, 1979.
[29] C.-J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 379-395, July 1986.
[30] R. Camposano, "Behavior-preserving transformations for high-level synthesis," in *Proc. Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*. New York: Springer Verlag, 1989.
[31] Benchmarks for the Fourth International Workshop on High-Level Synthesis, 1989.
[32] *Standard VHDL Language Reference Manual*. New York: The Institute of Electrical and Electronics Engineers, Mar. 1988.
[33] M. Barbacci, G. Barnes, R. Catell, and D. Siewiorek, "The ISPS computer description language," Rep. CMU-CS-79-137, Dep. Comput. Sci., Carnegie Mellon University, 1979.
[34] W. Wolf, "A catalog of optimizations for the behavioral synthesis of control-dominated machines," in *Proc. ACM/IEEE Workshop on High-Level Synthesis*, Kennebunkport, MA, Oct. 1989.
[35] H. DeMan, "Tutorial on high-level synthesis," in *Proc. EDAC'90*, Glasgow, Scotland, Mar. 1990.
[36] P. G. Paulin, J. P. Knight and E. F., Girzyc, "HAL: A multi-paradigm approach to automatic data-path synthesis," in *Proc. 23rd Design Automation Conf.*, June, 1986, pp. 263-270.
[37] R. Brayton, "Algorithms for multi-level synthesis and optimization," in *Proc. NATO ASI*, L'Aquila, Italy: Martinus Nijhoff, 1986.

*



Raul Camposano received the diploma from the University of Chile in 1978 and the Ph.D. degree in computer science from the University of Karlsruhe, in 1981.

In 1982 and 1983, respectively, he was with the University of Siegen and Professor of Computer Science at the Universidad del Norte in Antofagasta. From 1984 to 1986, he was a researcher in the Computer Science Research Laboratory at the University of Karlsruhe. Since 1986, he has been at the IBM T.J. Watson Research Center in Yorktown Heights, NY. His research interests include design automation for digital systems and computer design methodology.