Resources Restricted Aggressive Scheduling

Ping F. Yeung, David J. Rees
Department of Computer Science,
University of Edinburgh

Abstract
We describe a scheduling methodology for high-
level synthesis of designs with a significant amount
of control structure. The objective is to utilize all the
available resources while scheduling with respect to re-
source restriction. To do so, a vector/matrix structure
is built which provides a global view of resource usage
at each node. It supports the migration of operations
across basic blocks to wherever idle resources are avail-
able. With it, we formulate a list scheduling algorithm
in which the dispatching priority changes dynamically
with respect to resource availability.

1. Introduction
In high-level synthesis, in order to fit a design within
an allowable silicon area, a significant degree of re-
source re-usage is necessary. On the other hand, to
take full advantage of the available area, within the
search for an optimal area-time trade-off, resources
should be utilized as fully as possible. These chal-
lenges call for a scheduling methodology which is in-
telligent enough to extract the most timing advantage
out of the available resources.

Force Directed Scheduling introduced in [3] and
conditional resource sharing in [1] are able to ex-
plode mutually exclusive conditions for resource shar-
ing. However, when an operation is placed inside a
control block (fork-join pair), the scheduler cannot
move it out even though there are abundant resources
elsewhere.

The percolation scheduling algorithm described in
[5] can handle this better. It performs parallelisation
across the basic block boundaries. Starting with an
initial schedule, semantics preserving transformations
are applied repeatedly to convert the program flow
graph into a more parallel one. Also in [6], condition
vectors are used to represent the mutually exclusive
conditions. However, only one nested condition block
can be dealt with at a time. These two techniques are
closely related to our work. Their shortcoming is: the
effect of the resource restriction is not taken into ac-
count at the beginning.

2. Aggressive Code Migration
Let us consider the effect of aggressive code migra-
tion first with the example, sofaq , in figure 1a. The
control-data flow graph (CDFG) shown in
figure 1b representing the detailed control and data
dependencies of each node.

To extract all the parallelism in a description, we
can separate the computation part and the assignment
part of a statement by introducing a temporary vari-
able. Then, by moving all the computation parts out
of the conditional statements, a description can be re-
structured and divided into two sections: a computa-
tion section consisting of all the operation parts; and
an assignment section consisting of all the assignments
subject to given conditions, figure 2.

By storing the results in temporary variables, com-
putations can be performed without regard for the
guarding conditions. Then, later, when the conditions
are valid, they can be put back to where they belong.
In this way, the computation section will be saturated
with parallelism.

It is very beneficial to obtain all these potential
parallel-executable computations. However, we have
sacrificed the advantage offered by mutually exclusive
conditions - resource sharing. It will require more
resources than it should need and as a result, cause
a waste of resources. The key issue is the handling
of the condition expressions. A method is needed to
represent the guarding conditions so that it is possi-
ble to release parallelism when there is abundant re-
source, and to support resource sharing when there is
not enough.

3. Resource Restricted Scheduling
To overcome these problems, we have developed the
Resource Restricted Scheduling, R2Sch . Given the
input description and the resource constraints, the op-
Operations are scheduled into a minimum number of control steps. In order to achieve this, we need to extract as much fine-grain parallelism from the description as possible. This is done by means of an aggressive migration of operations across basic block boundaries.

```
x := Xi;
y := Yi;
a := y > 0;
case a is
  when "1" =>
p := y + x;
b := x > y;
m := y + p;
n := x + p;
end case;
case b is
  when "0" =>
  end case;
end case;
```

Figure 1: The CDFG of `sollon`.

Figure 2: Global-Data Transformation.

The main criteria of the scheduling algorithm are:

- Operations should not be restricted by basic blocks or fork-join pairs. They should be allowed to migrate anywhere within the schedule space.
- Specification of resource restriction should be allowed for fast area-time trade-off investigation and can be specified either by human designer or by expert system.

The scheduling algorithm, $R^2Sch$, is divided into two phases. The first phase scans the flow graph to gather information. Various static parameters for the priority function are computed. In the second phase, a list of candidates ready for scheduling is established. The most urgent one is chosen according to the priority function. After the node is assigned to cycles, the candidate list and the resources-used will be updated.

In the following sections, we introduce the priority function. The concept of resource vector ($RV$), propagated condition vector ($PCV$), and resource condition matrix ($RCM$) are used. Their detailed formulation has been presented in [7]. Basically, $RV$ represents the requirement of resources at each node. Each entity of it represents one kind of functional unit. Multiple resources requirement can be represented. For a simple operation, besides the functional unit, data transfer/buses and background memory access can be logged. For a sub-graph, the vector represents all the resource requirements of that graph. Differing from the path-based CV [6] which represents the path the node belongs, $PCV$ records the conditions under which the node is executed. The vector is derived from the global data dependencies across basic blocks. Finally, by crossing these two vectors, we get the resource condition matrix ($RCM$) which
shows the requirement of each resource under every condition. The RCM view of solon is constructed in figure 3.

The resource restricted scheduling is described in section 5, 6 and 7. Other detailed aspects of R2Sch can be found in [8].

4. Priority Functions

Two functions are used for determining the priority for scheduling.

Resource Priority and Path Delay

![Resource Priority and Path Delay](image)

Figure 4: Effect of different priority functions.

Why is a composition of priority functions needed? In figure 4 for instance, the current node has two different paths, \{a b\} to the output. If only the path delay or depth function is considered, path b will have a higher priority. However, it will only be true if there are enough resources to realize path(a) in less than four levels. Particularly, if only one adder is available, the above priority is clearly defective. A consideration of the accumulated resources against the resource constraints together with the accumulated delay can reflect a better priority for the whole situation.

The Path Delay Function of a node is defined as the longest path delay from the current node to the output. It shows how urgent each node is with respect to the timing constraints. It does not consider the effect of conditional branches, however. We therefore multiply it with the condition vector to take into account the delay of different branches.

The Resource Priority Function is derived by comparing the accumulated resource condition matrix (ARCM) with the remaining available resources. The ARCM of a node is the accumulation of its own RCM and the RCM's from all the nodes in its descendant sub-graph. This sub-graph contains the nodes which form the paths from the current node to the output. The ARCM of solon is shown in figure 5.

The ARCM provides a view of the foreseeable resource requirement at each node with the requirements on different kinds of functional unit and conditions. By projecting the ARCM onto the resource axis, the accumulated resource vector (ARV) is formed. For instance, the ARV of node4 is [4,1,15]. ARV shows the resource usage in the sub-graph of current node. When under resource constraint, it is the very information we needed to determine the priority of a node.

\[
\text{Available} = \text{Constraint} - \text{Used(sc)}
\]
\[
\text{Differential} = \text{ARV (nodes)} - \text{Available}
\]

The Constraint vector is the initial resource constraints input into the scheduling algorithm. The Used (sc) vector represents the amount of resource used so far in the current scheduling cycle, sc. Then, the difference, the Available vector, gives the amount of resource still available in the current cycle. The Differential vector, represents the resource shortage situation with respect to each type of functional unit. These differences from the differential vector can be weighted with the resource costs to determine the priority of a node.

This function gives a global view of the flow graph. The distribution of different kinds of operations and hence the requirement on different kinds of functional units.
5. Cycle Condition/Resource Vector

Since not all conditions in the conditional statements can be evaluated at the very beginning, to record the evaluated conditions during the scheduling process, for each cycle in the schedule space, there is a cycle condition vector (cycle-CV). When the select(case) operation of a conditional statement, CSi, is assigned to a cycle, the conditions for its branches will be valid. Assuming the current schedule cycle is sc, this is recorded by

\[ \forall \text{cycle-CV}_n \text{ with } c > sc + \text{delay} (\text{controller}) \]

record \((CV(CSi), \text{cycle-CV})\)

Analogous to cycle-CV, there is also cycle resource vector (cycle-RV). The cycle-RV is to record the resources committed so far in the current schedule cycle, sc. It is used in formulating the resource priority function with

\[ \text{Used}(sc) = \text{cycle-RV}_n \]

As cycle-RV is updated continuously as scheduling proceeds, the resource priority function will also be updated and hence, be able to reflect the priority dynamically.

6. Execution Condition/Resource

During the scheduling process, operations subjected to data dependence constraints can migrate to wherever resources are available. After a node ready to be scheduled is put into the candidate list, the order of migration is determined by the priority function. However, when an operation migrates outside its basic block, the control condition will no longer be the same. This leads to changes in the conditions under which it executes and also the resources it requires. In order to keep track of these changes, the execution condition vector (ECV) and the execution resource vector (ERV) are introduced.

While PCV assumes that all the conditions are available before any execution, ECV represents the actual conditions under which the node is executed. ECV is derived from the PCV of the operation and the cycle-CV of the current schedule cycle, sc.

\[ \text{ECV}(\text{node}_i) = \text{PCV}(\text{node}_i) \text{ masked by cycle-CV}_{sc} \]

\[ [10 \ldots] = [10 \ldots 01] \text{ masked by [11 11 \ldots]} \]

Due to resource sharing in mutually exclusive conditions, the resource requirement of a node to be assigned to the current schedule cycle may be less than what it requires in a stand-alone situation. Thus, the ERV reflects the actual requirement by taking away resources in the RV which could be shared exclusively with the nodes already assigned.

7. Scheduling

The second phase of \(R^2 Sch\) is to perform the scheduling. Nodes are scheduled according to their data flow dependence. For the nodes in \(\text{node}_\text{list}_{se}\) ready to be scheduled in cycle, sc,

\[ \forall n \in \text{node}_\text{list}_{se} \]

compute-ECV(n, cycle-CV_{se})
compute-ERV(n, ECV_{n})
if no_resource(ERV_{n}, Constraints, cycle-RV_{se})
    defer(n)
else compute_priority(n)

node = get_priority_node(node_list_{se})
if operation(node)
    settle(node), add(ERV_{node}, cycle-RV_{se})
distribute(data_flow_succ(node))
else if fork(node)
    record cycle-CV(CV(node))

The ECV and ERV of each node are computed first. The ERV is then compared against the constraints and the committed resources recorded in cycle-CV_{se}. If not enough resources are available, the node will be deferred to later cycles. Otherwise, the priority is computed using the static information from the resource priority and the path delay functions. A node with a null ERV implies perfect conditional resource sharing and will be given a very high priority. The highest priority node is selected and scheduled. For multicycling, the node will be settled into several consecutive cycles. At the same time, the cycle-CV_{se} is incremented. In the case of operation chaining, distribute adds the data dependent successors of node to the node_lists of the current cycle. Otherwise, they are added to some later cycles.

Listing 1

```
RV   PCV
[1] [10] 1. b := x + y
[0] [11] 2. case c
[1] [10] 3. <1> ⇒ p := x + b
[1] [01] 4. <0> ⇒ p := y + y
```

For the description in listing 1, listing 2 shows the schedule under normal situation. The PCV of node₁ reflects that b is only used in one of the branches.

Listing 2

```
Sch
RV, ERV, PCV, ECV
1 1 1 10 \ldots 1. b := x + y
2 1 11 \ldots 2. case c
3 1 10 10 \ldots 3. p := x + b
4 1 01 01 \ldots 4. p := y + y
```

504
On the other hand, when the conditional expression is evaluated at the very beginning, we have listing 3.

Listing 3

<table>
<thead>
<tr>
<th>Sch</th>
<th>RV, ERV, PCV, ECV</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>2. case c</td>
</tr>
<tr>
<td>c2</td>
<td>&lt; 1 &gt; [1 1 1 10 0 10]</td>
</tr>
<tr>
<td>c2</td>
<td>&lt; 0 &gt; [1 1 1 10 0 01]</td>
</tr>
<tr>
<td>c3</td>
<td>&lt; 1 &gt; [1 1 1 10 10]</td>
</tr>
</tbody>
</table>

As mentioned, by introducing temporary variables, statements can be decomposed into computation and assignment parts. With speculative execution, the computation parts can be migrated freely outside their basic block with a change of their ECVs. In listing 3, node3 and node4 can be split into computation and assignment parts, {3a, 3b}, {4a, 4b}. If two adders are available, statements 3a and 4a can be migrated across node3 and results listing 4.

Listing 4

<table>
<thead>
<tr>
<th>Sch</th>
<th>RV, ERV, PCV, ECV</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>1 1 1 10 0 10</td>
</tr>
<tr>
<td>c2</td>
<td>1 1 1 10 0 01</td>
</tr>
<tr>
<td>c2</td>
<td>1 1 1 10 10 01</td>
</tr>
<tr>
<td>c3</td>
<td>2. case c</td>
</tr>
<tr>
<td>c4</td>
<td>&lt; 1 &gt; [1 1 1 10 10]</td>
</tr>
<tr>
<td>c4</td>
<td>&lt; 0 &gt; [0 0 0 10 0 01]</td>
</tr>
</tbody>
</table>

This can only be carried out if data dependencies are preserved and abundant resources are available. By using this speculative scheduling technique, the computation can be speeded up globally. Significant improvement in resource utility can be achieved and conditional operations within basic block could be reduced and balanced.

As ECVs and ERVs are updated as operations are moved across the basic block boundaries, they record the execution condition and the resources requirement respectively. With them, the condition and the resource requirement of an operation can be represented dynamically. Scheduling of operations thus become more flexible and the use of migration allows operations to be distributed more evenly.

8. An Example

To illustrate the benefit of code migration, we compare \( R^2Sch \) with a conventional scheduling style i.e. where the mobility of an operation is confined to its basic block. Consider the example in listing 5. It consists of two disjoint conditional statements.

Listing 5

```plaintext
a := A; b := B; d := D; p := P; q := Q;
if (p > q) then c := p + q;
else c := p * q;
end if;
case c is
when 1 => y := ((a + b) + (c + d));
when 2 => y := ((a + b) + (c * d));
when 3 => y := ((a + c) * (b + d));
when 4 => y := ((a + c) + (b * d));
end case;
y := y;
```

Assuming the delay \{ Add - 1 cycle, Mul - 2 cycles, Cmp - 1 cycle, control - 1 cycle \}, resource unrestricted scheduling of both styles are performed. The result is 10 cycles for the conventional style and 7 cycles for \( R^2Sch \). Since resources are not restricted and operations are allowed to move outside their basic blocks, significantly more resources are required. Starting from the resource requirement of the maximum parallel schedule, the number of resources are gradually restricted. The results are summarized in table 1. The performance advantage of \( R^2Sch \) is mainly due to the concurrent evaluation of condition expressions and operations.

9. Results

To study the efficiency of the algorithm, several widely used examples have been tested.

a. An example from [2] is chosen first because it contains a substantial number of nested and disjoint condition branches. This helps to establish the effectiveness of \( R^2Sch \) in sharing resources under mutually exclusive conditions. \( R^2Sch \) is able to achieve the fastest schedule of 4 cycles with \{ 2 Add, 1 Sub \} which is known to be the optimum resource-delay combination.

b. Fifth-Order Elliptic Filter

Scheduling with various resource restrictions was performed. The results are summarized in table 2. The results of Force Directed Scheduling (FDS) [3], Force Directed List Scheduling (FDLS) [4] and Percolation...
Based Scheduling (PBS) [5] are summarized for comparison. Each schedule takes less than 1 second to compute on a SparcStation 1.

As a whole, $R'Sch$ is able to equal the best results published hitherto. These are believed to be the best possible for this heavily studied example. Although $R'Sch$ performs very well on this example, it is not a good yardstick. The description is a straight line segment of code consisting solely of arithmetic computations with no conditional statements. Nevertheless, it confirms that the underlying algorithm of $R'Sch$ is sound enough for basic block scheduling.

c. Group 1 Instruction of MC6502
From the original description, we have taken the group 1 instruction decode, address generation, and instruction execution as an example. Most subroutines in that group are expanded. The input description consists of two parts: address generation and instruction execution in two select statements. Each of them consists of 8 branches. Both conventional style and $R'Sch$ scheduling were carried out. Results show that schedules produced by $R'Sch$ are 16% to 18% faster. After studying the schedule graph, it is clear that the performance gain comes from the migration of computations.

10. Conclusions
In this paper, we have presented a new representation and scheduling methodology to handle dynamic code motion with respect to resource constraints. Conditional and different cycles resource sharing are considered concurrently. From the investigations conducted, it is observed that when operations are allowed to migrate to wherever resources are available, better scheduling results can be achieved. At discussed, the algorithm can perform scheduling with respect to resource restriction. With the resource vectors, bus transfers, background/foreground memory access can also be represented and restricted.

$R'Sch$ is nevertheless still far from meeting the ideals of high-level synthesis. The main reason is that it considers scheduling as an individual task, while tasks in high-level synthesis are highly interdependent. As a result of its good performance, it is being used for front-end coarse scheduling. It helps to evaluate the effect of allocation with different combinations of functional units. The flexible allocations and the scheduling results are passed onto the next stage where cycle and module binding are performed.

Table 2: Result of the Elliptic Filter.

<table>
<thead>
<tr>
<th>Resource</th>
<th>PBS</th>
<th>FDLS</th>
<th>FDS</th>
<th>Min</th>
<th>Back</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>3+, 3+</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3+, 2*</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2+, 2*</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2+, 1*</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1+, 1*</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3+, 2+*</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3+, 1+*</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2+, 1+*</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1+, 1+*</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Delay (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>POT</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>28</td>
</tr>
</tbody>
</table>

References