MASTER

High-level synthesis scheduling for conditional statements

van Besouw, P.W.P.M.

Award date:
1995

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
High-Level Synthesis
Scheduling for Conditional Statements

By P.W.P.M. van Besouw

Master Thesis
High-Level Synthesis
Scheduling for Conditional Statements

By P.W.P.M. van Besouw

Master Thesis

June 1, 1995
by order of Prof. Dr. Ing. J.A.G. Jess
supervised by Ir. M.J.M. Heijligers
Abstract

This thesis outlines several scheduling methods that emphasize conditional branching. The main idea of these algorithms is to schedule mutual exclusive operations to allow sharing of resources and possibly create a faster schedule for some paths in the control flow graph.

The representations of conditional branches can broadly be classified into two types: control select and data select. In order for a scheduler to make a tradeoff between resource sharing and fast execution of mutually exclusive branches, the scheduler must have the possibility to select either of the two forms of representation. Many systems do not consider conditional resource sharing in their basic algorithms. Some systems, however, consider mutual exclusion of some operations in conditional branches. Some well known mutual exclusion testing methods are the node coloring algorithm and the condition vector technique. These algorithms, however cannot handle certain types of conditional branches correctly, and give incorrect results in some cases. These problems are overcome by the branch numbering algorithm.

As-Fast-As-Possible (AFAP) is a path-based scheduling algorithm that ensures the minimum number of control steps for all possible sequences of operations in the control flow graph, under given resource constraints. This technique requires scheduling one operation into different states depending on the path. Although the worst case computational complexity is non-polynomial, there are no execution time problems in practice. The Condition Vector List Scheduling (CVLS) algorithm exploits a more "global parallelism". That is, it can parallelize multiple nests of conditional branches and optimize across the boundaries of basic blocks. Furthermore, it can optimize all possible execution paths. Also some control sequence improvement techniques as operation node reassignment and operation node dividing are introduced. Finally, the Hierarchical Reduction Approach algorithm transforms a data flow graph with conditional branches into an "equivalent" one that has no conditional branches. A schedule is then obtained for the latter, using a conventional scheduling algorithm, from which a schedule for the former is derived. Time complexity of this algorithm is low in comparison with the other algorithms, but it does not exploit potential parallelism to the fullest.

It is difficult to compare the different scheduling algorithms because the objectives of the techniques used are different. The AFAP algorithm minimizes the number of cycle steps, but the fundamental order of operations for a given path has to be chosen in advance and consequently potential parallelism of operations can easily be overlooked. Although the complexity of the Hierarchical Reduction Approach algorithm is low in comparison with the other algorithms the possibility of global resource sharing depends on the order in which the transformations are performed. The CVLS algorithm has the limitation that certain types of conditional branches cannot be handled correctly. However, these limitation are easily
overcome and furthermore the CVLS algorithm is best suited to exploit the possibilities to make a trade off between the complexity of the control path and the scheduling results. For these reasons the CVLS has been implemented on the NEAT++ system.
## Contents

1 Introduction ............................................. 1

2 Definitions and terminology ............................ 3
   2.1 The data flow graph ............................... 3
   2.2 General definitions ............................... 4
   2.3 Scheduling constraints and costs .................. 5

3 Conditional statements and branch representation ...... 7
   3.1 Conditional statements ............................ 7
   3.2 Representation of conditional branches .......... 8

4 Mutual exclusion testing ............................... 10
   4.1 Node coloring .................................... 10
   4.2 Condition vectors ................................ 11
   4.3 Node tagging .................................... 12
   4.4 Branch numbering ................................ 13

5 Path-based scheduling .................................. 17
   5.1 As-Fast-As-Possible scheduling Algorithm ........ 17
      5.1.1 Execution paths ............................. 17
      5.1.2 Hardware constraints ......................... 18
      5.1.3 AFAP scheduling of a single path ............ 19
      5.1.4 Overlapping the schedules for all paths .... 20
      5.1.5 Construction of the Finite-State Machine ... 21
   5.2 Conclusions ...................................... 22

6 Global scheduling based on Condition Vectors ........ 23
   6.1 Extended and actual condition vector ............ 23
      6.1.1 Extended condition vector .................. 23
## Contents

6.1.2 Actual condition vector .................................. 24

6.2 Condition Vector List Scheduling .......................... 27
  6.2.1 Module utilization vector ........................... 27
  6.2.2 Priority function .................................. 27
  6.2.3 CVLS algorithm for a single nested branch .......... 28
  6.2.4 Multiple conditional branches ..................... 29
  6.2.5 Control structure transformation .................. 29

6.3 Control synthesis ......................................... 30
  6.3.1 Control sequence synthesis method .................. 30
  6.3.2 Operation node dividing ........................... 30
  6.3.3 As-fast-as-possible scheduling ................... 31
  6.3.4 Pre-execution of branch operations ................ 32
  6.3.5 Operation node re-assignment ..................... 33

6.4 A scheduling example .................................... 33

6.5 Conclusions ................................................. 35

7 Hierarchical reduction approach ............................. 37
  7.1 Transformation of the data flow graph .................. 38
    7.1.1 Conditional resource sharing ..................... 38
    7.1.2 A transformation procedure ...................... 39
  7.2 Derivation of a schedule for the original data flow graph .. 41
  7.3 Generation of the Finite State Machine ............... 43
  7.4 Conclusions ............................................. 43

8 Conclusions .................................................. 45
  8.1 Comparison ............................................... 45
  8.2 Conclusions ............................................. 46
  8.3 Future work ............................................... 46

A Results of the CVLS algorithm ............................... 49
In high-level synthesis a behavioral description of a chip is converted into a structural design. The behavioral description is usually specified by means of an algorithm, similar to programming language descriptions. The structural description is a register-transfer level implementation that includes a data-path portion and a control portion. The data-path contains a network of functional units, registers, and their interconnection. The control part activates components of the data-path to realize the required behavior. One of the objectives is to find a structural description that satisfies the constraints, such as requirements on area, latency or cycle-time, while minimizing other costs. For example, the goal might be to minimize the area while meeting some timing requirements.

Due to its complexity, high level synthesis is divided into a number of distinct yet inter-dependent tasks:

- **Selection**: What kind of resources are required?
- **Allocation**: How many resources are necessary?
- **Binding**: Which operations have to be performed by a specific resource?
- **Scheduling**: When should specific operations be activated?

A lot of research is done in finding algorithms that solve these tasks satisfactory. The algorithms used, and the order in which they solve the tasks depend on the constraints and objectives given.

Scheduling and allocation are the most important tasks in order to synthesize circuits that are efficient in terms of area and performance. They are strongly related and inter-dependent. For example, scheduling attempts to minimize the number of required control steps subject to the amount of available hardware which depends on the results of allocation. Likewise, allocation exploits concurrency among operations to allow sharing of hardware resources, where the degree of concurrency is determined by scheduling.

Most scheduling algorithms minimize the number of control steps and/or the amount of hardware needed by exploiting the potential parallelism among the operations. Parallelism is extracted from the input description based on the data-dependencies between the operations. This technique, however, ignores the cases when operations are executed conditionally, which if taken into account can yield a better sharing of hardware resources.
Many systems do not consider conditional execution in their basic algorithms. Mutual exclusiveness between operations based on the topology can be derived of the control and data flow graphs. Operations in different branches of a conditional block are mutually exclusive and hence can share the same hardware (even if scheduled in the same control step). Another way of deriving mutual exclusiveness is to analyze all possible execution paths separately. This thesis gives a survey of known mutual exclusion testing methods and outlines several scheduling algorithms that exploit the freedom imposed by the presence of conditional branches.
Definitions and terminology

This chapter presents the definitions and terminology necessary for the remainder of the thesis.

2.1 The data flow graph

Def 2.1 (Graph) A Graph G is defined by a tuple \((V, E)\), in which:

1. \(V\) is a finite set of nodes, and
2. \(E \subseteq V \times V\) is a set of directed edges.

A data flow graph is a graph, where each node represents an operation, while the edges represent the transfer of values between nodes. The execution of the graph is defined by a token-flow mechanism. A behavior can be defined using nodes and edges with different properties [Eijn91]. According the properties the nodes and edges can be classified. Using these classified nodes, it is possible to describe certain constructs. Special constructs and the use of complex data-types results in an extension on the basic classification of the node types.

Classification of Nodes and Edges Nodes are distinguished by so called node types, each describing a particular semantic. Some nodes are:

- **Operation Nodes**
  These nodes represent operations like arithmetic operations (+, −, ∗, /), boolean operations (and, or, <, >) or more complex operations. These complex operation nodes provide hierarchy within the graph semantics as used in description languages.

- **Input and Output Nodes**
  Every graph requires at least one input node and at least one output node to define its interface. Output nodes are the only nodes without output edges, input nodes are the only nodes without input edges.

- **Constant Nodes**
  Nodes of type constant are nodes which generate a constant data value at their output. Nodes of this type can only execute when they are enabled.
• Delay Nodes

Nodes of this type pass incoming (data) tokens to their output after a delay. When
the node is initialized, it contains an initial token which becomes available at the first
execution of it.

2.2 General definitions

The process that determines when operations of a data flow graph \( G = (V, E) \) start and
finish their execution is called scheduling. The start time of an operation \( v \in V \) is denoted
by \( \text{begin}(v) \) and the end time by \( \text{end}(v) \). The pair \([\text{begin}(v), \text{end}(v)]\), \( \text{begin}(v) \leq \text{end}(v) \) is called
the schedule of operation \( v \). This pair is denoted by \( \varphi(v) \). In order to make a schedule
the need arises to know which operations can be performed by which functional modules
and the area costs and delays of these modules. For this purpose libraries containing these
functional modules and their information exist. The following definitions deal with how to
obtain the information needed to make a schedule from the libraries.

**Def 2.2 (Type of Operation)** Let \( G = (V, E) \) be a data flow graph, \( v \in V \) and \( \text{Otype} \) be a set
of operation types. \( \tau : V \mapsto \text{Otype} \) is a function. \( \tau(v) \) returns the operation type of operation
\( v \).

**Def 2.3 (Type of a Module)** Let \( M \) be a set of modules, \( m \in M \) and \( \text{Modtype} \) be a set of
module types (often called a library). \( \tau : M \mapsto \text{ModType} \) is a function. \( \tau(m) \) returns the module type of module \( m \).

**Def 2.4 (Operation Type Mapping)** Let \( \text{Otype} \) be a set of operation types, \( t \in \text{Otype} \) and
\( \text{Modtype} \) be a set of module types (often called a library). \( \mu : \text{Otype} \mapsto \text{Modtype} \) is a function.
\( \mu(t) \) returns the module types that can implement the operation type \( t \).

**Def 2.5 (Delay)** Let \( \text{Modtype} \) be a set of module types (a library), \( \text{Otype} \) be a set of operation
types, \( l \in \text{Modtype}, t \in \text{Otype} \) and \( l \in \mu(t) \). \( d : \text{Otype} \times \text{Modtype} \mapsto \mathbb{R} \) is a function; \( d(t, l) \)
returns the number of cycles the operation type \( t \) needs when it is executed on a module
type \( l \). This number of cycles is called the delay of the operation.

**Def 2.6 (Cost)** Let \( \text{ModType} \) be a set of module types (a library) and \( l \in \text{ModType} \). \( c : \text{ModType} \mapsto \mathbb{R} \)
is a function; \( c(l) \) describes the cost of a functional unit type.

**Def 2.7 (Operation Mapping)** Let \( G = (V, E) \) be a data flow graph, \( v \in V \) and \( \text{Modtype} \) be a
set of module types (a library). \( \xi : V \mapsto \text{Modtype} \) is a mapping from an operation to a module
type. \( \xi(v) \) returns the module type upon which the operation \( v \) will be implemented.

Once for all nodes \( v \in V \) the operation mappings are known, the delay \( \delta(v) = d(\tau(v), \xi(v)) \)
of each node \( v \) can be determined. For each each scheduled operation \( v \in V \) the following
property holds:
\( \text{begin}(v) + \delta(v) = \text{end}(v) \).
2.3 Scheduling constraints and costs

The aim of scheduling is to find the best point in the design space, i.e. the point with minimal costs. The goal of the designer is reflected by a cost function. Minimization of the total area is a possible objective. The number of modules needed to implement a scheduled data flow graph can be determined by counting the number of operations that are executed in the same cycle on the same module type. A distribution function is needed to define module cost.

Def 2.8 (Distribution Function) Let \( T \) be a set of cycles, \( t \in T \), \( \text{Modtype} \) be a library, \( l \in \text{Modtype} \), \( S \) be the set of schedules and \( \varphi \in S \). \( DF : S \times \text{Modtype} \times T \rightarrow \mathbb{R} \) is a function given by:

\[
DF(\varphi, l, t) = | \{ v \in V \mid \xi(v) = l \land t \in [\text{begin}(v), \text{end}(v)] \} |
\]

The cost of a schedule can now be defined in terms of distribution functions: Let \( S \) be the set of schedules and \( \varphi \in S \). \( c : S \rightarrow \mathbb{R} \) is a cost function given by \( \sum_{l \in \text{Modtype}} c(l) \cdot \max_{t \in T} DF(\varphi, l, t) \).

Precedence constraints A data flow node can only start its execution after all its predecessors have finished their execution and all its input is available. This partial order imposed on the start times of the nodes in the data flow graph is given by the precedence relation of the graph.

Def 2.9 (Precedence Relation) Let \( G = (V, E) \) be a data flow graph and \( v_i, v_j \in V \). \( \prec : V \times V \) is a relation between two operations of \( G \). \( v_i \prec v_j \) if and only if there is a data flow from \( v_i \) to \( v_j \).

The precedence constraints for each pair of nodes for which \( v_i \prec v_j \) can be given by:

\[
\text{begin}(v_i) + \delta(v_i) \leq \text{begin}(v_j), \text{ or } \text{end}(v_i) \leq \text{begin}(v_j).
\]

Time Constraints The data flow graph has to be scheduled within a limited number of cycles, denoted by \( T_{\text{max}} \). This means that \( \text{end}(v) \leq T_{\text{max}} \) for all operations \( v \in V \).

Resource Constraints The number of resources that may be used are determined beforehand. Often resources are modules, but one may also constrain the number of registers, multiplexers and interconnect.

Asap and Alap schedules An ASAP (as soon as possible) scheduler schedules each operation at the earliest possible time while retaining the precedence constraints. Let \( G = (V, E) \) be a data flow graph. For each \( v \in V \) ASAP schedule \( \varphi(v) \) is given (recursively) by:

\[
asap : V \rightarrow \mathbb{R}
\]

\[
asap(v) = \begin{cases} 
0 & \text{if } \text{pred}(v) = \emptyset \\
\max_{v_i \in \text{pred}(v)} (\text{asap}(v_i) + \delta(v_i)) & \text{otherwise}
\end{cases}
\]

\[
\varphi(v) = [\text{asap}(v), \text{asap}(v) + \delta(v)]
\]
An ALAP (as late as possible) scheduler schedules each operation at the latest possible time while retaining the precedence constraints. Let $G = (V, E)$ be a data flow graph. For each $v \in V$ ALAP schedule $\varphi(v)$ is given (recursively) by:

$$\text{alap} : V \mapsto \mathbb{R}$$

$$\text{alap}(v) = \begin{cases} T_{\text{max}} & \text{if succ}(v) = \emptyset \\ \min_{v_i \in \text{succ}(v)} (\text{alap}(v_i) - \delta(v_i)) & \text{otherwise} \end{cases}$$

$$\varphi(v) = [\text{alap}(v) - \delta(v), \text{alap}(v)]$$
3.1 Conditional statements

There are two types of conditional statements:

1. **if-then-else**: according to a test-condition either the *then* or the *else* part is executed (see figure 3.1(a));

2. **case**: according to a test-condition the appropriate case-body is executed, identified by a label. The test-condition must be covered by one (and at most one) label, and the total set of labels determine the possible test-conditions (see figure 3.1(b)).

For each statement holds that only one of the alternatives can be executed.

```
if test then
  then-body
else
  else-body
fi
```

```
switch test
  case 0: case-0-body
  case 1: case-1-body
  ...
  default: default-body
switchend
```

(a) If-then-else statement  (b) Case-statement

Figure 3.1: Example of conditional statements

For the implementation of conditional statements *control-nodes* are needed. When a test-signal is evaluated, and a body is selected, tokens must be fed to the inputs of that body. All remaining bodies do not get tokens. When bodies are executed, they deliver tokens to their output; these must be merged and be made available for the next operations. So, the control nodes either deliver tokens to the inputs of a body (*branch-node*), or merge tokens received from the the outputs of the body (*merge-node*). These nodes are of the class *control*, and are shown in figure 3.2.
3.2 Representation of conditional branches

Representation of conditional branches can broadly be classified in two types: control-select (C-select) and data-select (D-select). In the C-select representation (figure 3.3(a)) a conditional branch is selected before any conditional operation is executed. Since alternate branches of a conditional are mutually exclusive, there is no resource conflict between operations in different branches and all mutually exclusive operations of the same type can share resources. In figure 3.3(a), one module can perform the two operations without any resource conflict. In C-select representation the decision as to which operations share resources is postponed until scheduling or binding is done. Due to the absence of information on sharing resources a priori, the precedence relationships between condition selection and execution path must exist in the representation.

In D-select representation (figure 3.3(b)) all conditional branches are executed separately in parallel and correct data values are selected at the end of the conditional branch. This representation may result in fast execution since conditional operations can be executed before the condition selection is made. One drawback with this representation is that mutually exclusive operations cannot share resources, resulting in expensive designs. The use of D-select representations, however, may be preferable to C-select in certain cases, for example, when conditional branches do not contain any mutually exclusive operations due
3.2 Representation of conditional branches

to type conflict. Another drawback with the D-select representation is that some conditional actions cannot be represented. This is because the correct conditional path is chosen by the selection of output values and some conditional actions do not have an output, for example:

```
P;
if test then
  exit(1)
fi
Q;
```

In this example `exit(1)` is always executed and hence Q is never reached. Optimally, the scheduler must have the possibility to make a tradeoff between the two forms of representations.
Mutual exclusion testing

Many systems do not consider conditional resource sharing in their basic algorithms. Some systems, however, consider mutual exclusion of some operations in conditional branches [Paul89, Park86, Tsen88]. Some well-known algorithms for mutual exclusion testing are the node coloring algorithm by Park ([Park88]) and the condition vector technique by Wakabayashi ([Waka89, Waka92]). In the following sections several algorithms will be described that deal with mutual exclusion testing.

4.1 Node coloring

The node coloring algorithm assigns a color code consisting of a sequence of one or more integers to each node such that testing of mutual exclusion between and two nodes can be done by simply comparing the color codes of the nodes in a constant number of steps. The length of the color code of a node represents how many levels of branch-merge blocks the node is nested in. A single digit color code represents that the node is not in any branch-merge block. The rules for node coloring are:

1. Every unconditional operation, including the outermost branch and merge nodes, has a unique single element code sequence.

2. In an outermost branch-merge block, the first elements in the code sequences of all the nodes are the same.

3. For any branch node with a color code of length $n$, every successor node except the matching merge node has a color code of length $n + 1$ where the first $n$ elements are the same as the branch node and the $n + 1$ element is a unique integer among the successors.

4. A merge node has the same color as its matching branch node.

5. Any two connected nodes have the same color code if neither of them is a branch or a merge node.

The mutual exclusion testing procedure for any two nodes in a data flow graph is as follows (for an example see figure 4.2). Two nodes with color codes $M$ and $N$, with lengths $m$, respectively $n$ are mutually exclusive if:
4.2 Condition vectors

A new vector form expression, called Condition Vector (CV) is introduced [Waka89] to deal with conditional branches systematically. The CV shows conditions to identify when operations are activated. The conditions of operations execution can be expressed in Boolean expressions or in some other expressions. An expression suitable for exploiting mutual exclusiveness of conditions is the basic Condition Vector. The basic CV is defined to be the bitwise encoding of distinct leaf branches for the nested conditionals, such that every leaf has a one-hot encoding. CVs for non-leaf nodes are obtained by logical or operation of the successor nodes' CVs. That is, basic CV \( v_i \) is defined as follows:

\[
v_i = \begin{cases} 
\text{leaf vector for } n_i & (n_i \text{ is a leaf node}) \\
v_k \text{ or } v_l \text{ or } \cdots \text{ or } v_m & (n_i \text{ is not a leaf node})
\end{cases}
\]

In this way, all possible conditions are represented by the basic CV systematically.

---

**Figure 4.1:** Basic CV for nested if construct

Figure 4.1 shows an example of the basic CV. The second bit of the basic CV corresponds to node \( n_o \), and it shows the nodes execution condition, namely, \( c1 \rightarrow c2 \rightarrow c3 \). Mutual exclusiveness is derived by means of a CV associated with each operation. When the CVs for the two
operator nodes have no common bits asserted, those nodes are mutually exclusive and can share the same hardware resources. The basic CVs for each operation express explicit mutual exclusiveness. This encoding is not expensive. The dimension of the basic CV is the number of leaf branches. Therefore, the dimension is equal to

\[(\#\text{if statements} + 1) + (\#\text{case branches}) - (\#\text{case statements} - 1)\]

### 4.3 Node tagging

A drawback of the algorithms as described in sections 4.1 and 4.2 is that they cannot handle certain types of conditional branches correctly, and give incorrect results in some cases (see [Rim92]). In figure 4.2, node \(n_a\) is mutually exclusive to node \(n_5\) but is not mutually exclusive to node \(n_c\). The algorithm of Park and Wakabayashi incorrectly indicate that nodes \(n_a\) and \(n_c\) are mutually exclusive.

<table>
<thead>
<tr>
<th>color</th>
<th>cond. vector</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Park88</td>
<td>Waka89</td>
<td>Rim92</td>
</tr>
</tbody>
</table>

```
if c1 then
  if c2 then
    na 111 100000 1T 2T
  else
    nb 112 010000 1T 2F
  fi
if c3 then
  nc 121 001000 1T 3T
else
  nd 122 000100 1T 3F
fi
else
  if c4 then
    ne 131 000010 1F 4T
  else
    nf 132 000001 1F 4F
  fi
fi
```

Figure 4.2: Example of node coloring, condition vectors and node tagging

These problems arise when there are multiple branches within a branch and they are overcome by the node tagging algorithm as described in [Rim92]. A tag is assigned to each node. The tag of a node represents the condition selections which activates the node. Each element contains two fields: the condition field and the selection field. The condition field represents the boolean values of the conditional branch. The selection field represents either true or false decision which activates the node. Two nodes are mutually exclusive if a node is activated with the true selection of a certain condition and another node is activated with the false condition. Thus two nodes with tags \((1T\ 2F)\) and \((2T\ 3F)\) are mutually exclusive.
However, this tagging handles only one selection condition, so multi-way branches are not allowed. Furthermore, if two nodes are tagged with (1T) and (2T) respectively, the two tags assigned to the two nodes do not have enough information to determine if they are mutually exclusive or not.

4.4 Branch numbering

The most recent technique for mutual exclusion testing is the branch numbering algorithm [Amel94]. The branch numbering algorithm has none of the drawbacks mentioned in the previous sections. First a few new concepts are introduced and they are defined as follows:

- The whole control and data flow graph is represented by a connected graph, called *conditional graph*, made up of the conditional and non-conditional branches. This conditional graph is the skeleton of the control and data flow graph. An example is given in figure 4.3.

- A *global block* groups an unconditional branch and all the subsequent conditional branches.

- A *branch* groups all the nodes executed under the same conditions, if it is conditional. Otherwise, it groups all the nodes that exist before the next conditional branch.

- The *level* of a conditional branch is the number of branch nodes to be executed before reaching the branch incremented by 1, because a non-conditional branch is of level 1.

![Figure 4.3: Conditional graph example](image)

This mutual exclusion testing procedure uses a branch numbering that is similar to the node coloring technique, but implemented and used in different ways. Only branches of the conditional graph are numbered, and each node has the reference of the branch to which
it belongs. The numbering of a branch in the graph is a dynamic size vector having the following form:

\[ B = (n, N_1, N_2, \ldots, N_n) \]

where:

- \( n \) is the branch level,
- \( N_1 \) is the global block number, and
- \( N_i \) for \( 2 \leq i \leq n \)

An example of branch numbering is given in figure 4.4. Using this branch numbering, the testing of the mutual exclusion of nodes is easily performed. Having two nodes, one from a branch \( i \) whose number is \( B_i = (n, N_1, N_2, \ldots, N_n) \) and the second from a branch \( j \) whose number is \( B_j = (m, M_1, M_2, \ldots, M_m) \), they are mutually exclusive if:

1. \( B_i \) and \( B_j \) are in the same global block \( (N_1 = M_1) \), and
2. \( B_i \) and \( B_j \) are in a conditional block \( (n \) and \( m > 1) \), and
3. there exist two different conditional branches of same level \( k \) \( (N_k = M_k) \) having the same root \( (N_t = M_t \) for \( t < k) \).

In figure 4.4 a node of branch \( (3,1,2,3) \) is not mutually exclusive with a node from branch \( (2,2,2) \), because they are not in the same global block. But \( (3,1,2,3) \) is mutually exclusive with a node of branch \( (2,1,1) \) because:
4.4 Branch numbering

- they are in the same global block (1),
- are in a conditional block,
- and there exist two different conditional branches (2,1,1) and (2,1,2) of the same level (2) having the same root.

The algorithm in figure 4.5 calculates the branch numbers for all nodes in data flow graph G. Because the branch level n is equal to the length of B, it can be omitted. So n = len(B), where $B = (N_1, N_2, \ldots, N_n)$.

```plaintext
procedure calculate_branch_number ( G(V, E) )
    level := 1; block := 1; L := \emptyset;
    foreach \{ v_i \mid \tau(v_i) = input \land v_i \in V \} do
        b(v_i) = (1);
        L := L \cup \{v_i\};
    endforeach;
    while L \neq \emptyset do
        NL := \emptyset;
        foreach v_i \in L do
            if (\tau(v_i) = branch) then
                level += 1;
                branch = 0;
            fi
            if (\tau(v_i) = merge) then
                level -= 1;
                if (level == 1) then
                    b_i = (block + 1);
                fi
            fi
            foreach (v_i, v_j) \in E do
                if (\tau(v_i) = branch) then
                    b_{v_j} = b_{v_i} \oplus (branch); /* \oplus : concatenation */
                    branch += 1;
                else
                    b_{v_j} = b_{v_i};
                fi
                NL := NL \cup \{v_j\};
            endforeach
        endfor
        L := NL;
    endwhile
end
```

Figure 4.5: Calculate Branch Number
Chapter 5
Path-based scheduling

A behavioral description of the problem to schedule is given by the directed control flow graph $G_c = (V, E)$. The objective of path-based scheduling ([Berg91, Berg92, Camp91a, Camp91b]) is to yield solutions with a minimum number of states, under given constraints such as timing and area. The graph may contain conditional branches and loops. In fact in this approach conditional branching is emphasized. The scheduling algorithm ensures that each of the possible paths generated by conditional branches is scheduled optimally (in the minimum number of states) for the given constraints. This requires scheduling one operation into different states depending on the path. This path-based scheduling algorithm is termed As-Fast-As-Possible (AFAP) scheduling.

5.1 As-Fast-As-Possible scheduling Algorithm

The main problem addressed by the AFAP algorithm is the minimization of the number of control steps for all possible sequences of operations in the control flow graph, thus AFAP is a path-based scheduling algorithm. Scheduling is performed on each path independently, taking into account only the set of constraints pertinent to the operations in the path. As a result the minimum number of control steps is found for all paths and not only for the critical path. The algorithm consists of the following steps:

1. Compute all execution paths in the control flow graph.
2. Derive all constraint intervals applicable to each path.
3. Schedule each path independently using exact clique covering techniques.
4. Overlap the schedules for all paths, minimizing the total number of states.
5. Construct the Finite State Machine for the resulting schedule.

5.1.1 Execution paths

A path is a sequence of operations which is executed if all conditions defining the path evaluate to true. Paths are computed by performing a depth-first traversal of the acyclic control flow graph (which is made acyclic by removing the feedback edges in loops). Any sequence of operations starting at a given first operation and ending at operations with no
successors constitutes a path. Since loops represent repetitions of sequences of operations, the first operation in a loop also starts a new path.

As an example we use the control flow graph depicted in figure 5.1(a). All execution paths are computed, resulting in the paths \{1, 2, 3, 4, 8, 9, 10, 11, 12, 13\}, \{1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13\} and \{10, 11, 12, 13\}.

5.1.2 Hardware constraints

Constraints are restrictions imposed on the final implementation which are used to guide the scheduling. Constraints are the following:

- Variables can be assigned only once in one state.
- IO ports can be read or written only once in one control state.
5.1 As-Fast-As-Possible scheduling Algorithm

- Functional units can be used only once in a control state. This constraint is only relevant if the amount of hardware is restricted. 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.

Constraints are represented as intervals in the control flow graph. This type of representation allows constraints to be applied on a path basis. A constraint interval involves a sequence of operations and it implies that these operations cannot all be executed in the same cycle step. In other words a new state must start at some point within the interval, for the constraint to be met.

Several constraints are imposed on the graph (see figure 5.1(b)); Constraint C10 is present between nodes 1 and 9 because Data Ready is an output port and therefore operations 1 and 9 should not be executed in the same control step. Assuming that only one adder should be used, constraint C11 prevents the two additions on operations 3 and 4 from being scheduled in the same clock cycle. The other constraints are derived in a similar way.

5.1.3 AFAP scheduling of a single path

A constraint interval in a path implies that the operations contained in the interval cannot all be scheduled in the same control step. Hence, the scheduler must cut the path inside the interval, so that the operations before the cutting point are scheduled in one control step and the operations after it are scheduled in the following control step. A cut through multiple overlapping constraints clearly satisfies all of them simultaneously. Thus the problem of scheduling a single path in the minimum number of control steps is reduced to the problem of finding the minimum set of cuts which satisfies all constraints. This problem can be formulated in terms of finding the minimum clique cover of the constraint graph, which is
constructed as follows: nodes correspond to constraint intervals and an edge between two nodes indicates that the two intervals overlap. A clique in this graph corresponds to a cut, which is given by the overlapping of the intervals in each clique. Hence in order to find the minimum set of cuts required, it is enough to find a minimum clique cover of the constraint graph. Since this graph is an interval graph, a minimum cover can be found in linear time using the left-edge algorithm.

![Diagram of paths and cuts](image)

Figure 5.3: Paths and cuts

Figure 5.2 gives the constraint graph (constructed from the paths and constraints in figures 5.1(a) and 5.1(b)) and a minimum clique covering. Figure 5.3 shows the cuts generated from each clique. At this point each cut may still comprise a range of nodes.

### 5.1.4 Overlapping the schedules for all paths

The schedule of the complete design is obtained by combining the schedules for all paths. Cuts in different paths may overlap at common nodes. The effect of merging these overlapping cuts is that the resulting states (in the different paths) are also merged, reducing the final number of states. Thus the problem of finding the minimum number of states for all paths (and still fulfill the AFAP schedule for each path) is equivalent to find the minimum
set of non-overlapping cuts. The graph formulation of this problem is: find the minimum clique cover of the cut graph, whose nodes represent the cuts in all paths and the edges join overlapping cuts. The overlapping cuts are redefined to contain only the overlapping nodes. Since the cut graph may be a general graph, the computation of the minimum clique cover is an NP problem. In practice, however, this is reasonably sparse and exact algorithms can be used.

Figure 5.4: Cut graph and final cutting points

After this step all necessary cuts have been determined. Some cuts may still contain more than one operation, in which case, a single operation within each cut must be selected as the final cutting point for the construction of the control automaton. Although this selection makes no difference in the number of control steps, it may affect the final number of registers and the performance of loops.

Figure 5.4(a) shows the cut graph and a minimum clique cover for the paths and cuts in figure 5.3. Figure 5.4(b) gives the final cutting points selected in each path.

5.1.5 Construction of the Finite-State Machine

The final states of the controller are formed in the following way. A path is divided by sets of cuts into a number of path segments, which are placed in successive control steps. Path segments from different paths which share the same first operation are merged in the same state. The conditions on the state transitions are derived from the conditions upon which each path segment succeeds another path segment. These conditions are computed by ANDing the conditions on the control flow edges leading from the first operation in a path segment to the first operation in the succeeding path segment.

Figure 5.5 shows the finite-state machine built from the paths and cuts in figure 5.4(b). The AFAP scheduling of a description containing conditional operations and loops may require that some operations (e.g. operations 8, 9 and 10) are scheduled in more than one state. As a result the overall number of states may not be minimal, but a schedule with less states
would not be AFAP for all paths. The notation 5(a) and 13(\(\overline{b}\)) indicates that the operations are executed if and only if \(a\) respectively \(\overline{b}\) are true.

5.2 Conclusions

AFAP scheduling is a path-based method and obtains the minimum number of cycle steps for all execution paths. To allow the optimal scheduling for all execution paths, operations may be scheduled into several cycle steps for the different execution instances. The main limitation is the fact that the order in which the operations are to be executed must be chosen in advance and consequently operations may be needlessly scheduled into different cycle steps which attributes to the complexity of the control path in a negative way.
Global scheduling based on Condition Vectors

Although the path-based scheduling algorithm finds all possible execution paths and optimizes each of them, the fundamental order of operations for a given path is chosen in advance, so that the scheduling result may not be sufficiently optimized across the basic limit boundaries. The main aim of scheduling based on condition vectors is to accomplish the following three objectives at the same time:

- Resource sharing for alternation.
  Hardware resources, e.g. function units, registers and connections, could be shared when they are executed alternatively. Hence, the exploiting mutual exclusiveness is important in order to reduce required hardware resources.

- Different scheduling for each path.
  There are many paths in a behavior description containing conditional branches. The number of necessary steps to accomplish them differs according to conditionals, because each branch needs a different number of control steps.

- Scheduling independent of control dependencies.
  The condition vectors can express dynamic mutual exclusiveness and consequently the scheduler can neglect the control dependencies, while the semantic of the given behavior is preserved.

6.1 Extended and actual condition vector

6.1.1 Extended condition vector

The basic condition vector (see section 4.2) shows the execution condition in HDL description and it expresses explicit mutual exclusiveness for each operation. The basic CV is now extended to data flow graph nodes. The extended CV $e_i$ shows the conditions wherein a node $n_i$ needs to be executed and is defined as follows:

$$
e_i = \begin{cases} 
[1] & (n_i \text{ is a predecessor of an output node}) \\
\text{basic CV } v_i & (n_i \text{ is a predecessor of a merge node}) \\
\text{or } e_k \text{ or } e_l \text{ or } \cdots \text{ or } e_m & (\text{in other cases})
\end{cases}$$
where, nodes \(n_k, n_i, \ldots, n_m\) are successors for node \(n_t\). The expression \([x]\) denotes a vector all of whose components are ‘1’ (i.e. \([1, 1, \ldots, 1]\)). Similarly \([0]\) denotes \([0, 0, \ldots, 0]\).

The extended CV basically has the same value as its basic CV. However, the value of extended CV is sometimes different from that of the basic CV, as shown in 6.1. The value of variable \(x\) is not used in all branches. In this description operations in statement (1) need not be executed when the condition is \(c_1 \cdot c_2\). This fact is expressed by extended CVs. Then, basic CV of operations in (1) is \([1, 1, 1]\), while the extended CV is \([1, 0, 1] = [1, 0, 0](2)\) or \([0, 0, 1](4)\). The ‘0’ component shows the conditions that operations need not to be carried out: \(c_1 \cdot c_2\). Therefore, the statements (1) and (3) are mutually exclusive and can share the same resources. In this way, operations which are not mutually exclusive in behavioral description are sometimes implicit mutual exclusive.

\[
\begin{array}{|c|c|c|}
\hline
\text{BCV} & \text{ECV} & \text{ACV} \\
\hline
x = a + b; & [1, 1, 1] & [1, 1, 1] & [1, 1, 1] \\
\text{if } (c1)\text{ then} & [1, 0, 0] & [1, 1, 1] & [1, 0, 0] \\
y = x + 1; & [0, 1, 0] & [1, 1, 1] & [0, 1, 1] \\
\text{else} & [0, 0, 1] & [0, 1, 1] & [0, 1, 1] \\
\text{fi} & /\star x \text{ is not used anymore }/ \\
\hline
\end{array}
\]

\(\text{Resolved Condition: } c1\)

\(\text{Unresolved Condition: } c1 \cdot c2\)

(a) Behavior description

(b) data flow graph

Figure 6.1: Condition vector example

The extended CV values are calculated by traversing the nodes in the data flow graph from the end node to the start node. They are determined independently for each top level IF construction. The algorithm in figure 6.2 calculates the extended condition vector for all nodes in data flow graph \(G(V, E)\). The procedure does a breadth-first search starting from the output nodes and updates all extended condition vectors. When a node is a predecessor of a merge node the extended condition vector is set equal the the basic condition vector.

6.1.2 Actual condition vector

Finally, the actual CV is introduced. The actual CVs are used in scheduling, allocation and control sequence synthesis. When exploiting both potential parallelism and alternatives, the conditionals are difficult to handle. Consider the following simple description:
procedure calculate_extended_condition_vector(G(V, E))
  \text{L} := \emptyset
  \text{foreach} \{ v_i \mid \tau(v_i) = \text{input} \land v_i \in V \} \text{ do}
    b(v_i) = [1];
    L := L \cup \{v_i\};
  \text{endforeach}
  \text{while} L \neq \emptyset \text{ do}
    NL := \emptyset
    \text{foreach} v_i \in L \text{ do}
      \text{foreach} (v_j, v_i) \in E
        \text{if} (\tau(v_i) = \text{merge}) \text{ then}
          e_{v_j} := b_{v_j};
        \text{else}
          e_{v_j} := e_{v_j} \lor e_{v_i};
        \fi
        NL := NL \cup \{v_j\};
    \text{endforeach}
  \text{endforeach}
  L := NL;
end

Figure 6.2: Algorithm to calculate extended condition vectors

\begin{verbatim}
if (a > b) then
  x = c - d;
else
  x = c - e;
fi
\end{verbatim}

In order to share an adder, the conditional operation \(a > b\) must be completed before either branch can be calculated (control-select). On the other hand, if the alternatives are not used, both branches should be calculated and the value of \(x\) is selected from the results of both branches according to the conditional \(a > b\) (data-select). In the latter case, the adder cannot be shared, but the conditionals and branches could be executed concurrently. Furthermore, branches could be calculated before the pertinent conditional operations are executed or branches and conditional operations could be executed in parallel. Operations in different branches are mutually exclusive after the conditionals are resolved, and are not mutually exclusive until conditionals are resolved. Such dynamic mutual exclusiveness can be handled by actual CVs. The value of the ACVs are changed whether conditionals are resolved or not.

Extended CVs express mutual exclusiveness when all conditionals are resolved. Then, the ACV value of a operation can be obtained by setting its ECV components corresponding to unresolved conditionals to '1'. An ACV shows the actual condition under which operations are executed. The ACV changes it values, as conditional operations are scheduled. After all conditional operations are scheduled, the actual CVs are equal to the extended CVs.
The actual CV $a_i$ for operation node $n_i$ in cycle step $k$ is calculated as follows:

$$a_i = e_i \text{ or } (e_i \text{ or } \cdots \text{ or } e_m)$$

where, nodes $n_i, \ldots, n_m \in C_i$. $C_i$ is a set of conditional nodes defined as follows:

$$C_i = \left\{ n_j \left| \begin{array}{l} n_j \text{ is a conditional node related to node } n_i, \\ n_j \text{ is unresolved until cycle step } k, \\ e_i \text{ and } e_j \neq [0] \end{array} \right. \right\}$$

In the last equation, 'and' denotes the bitwise logical and operation and the equation means that node $n_i$ and conditional node $n_j$ are not mutually exclusive. An example of actual CVs are shown in figure 6.1. When conditional $c_2$ is unresolved, the ACV value of (3) becomes $[0, 1, 1]$ ($= e_3[0, 1, 0] \text{ or } e_2[0, 1, 1]$).

When the conditional operation delay is short (e.g. Boolean operation, read data), the control signal for a conditional branch can be available in the same cycle step as the control operation. On the other hand, when the delay of a control operation is long (e.g. comparator), the control signal cannot be used until the next cycle step.

For example, when description 1 in figure 6.3 has to be carried out in a single cycle step and control operation ($a > b$) requires a delay of one cycle step, then the conditional signal is not available until the next cycle step. Therefore, adders are not mutually exclusive and two adders and one comparator are needed. On the other hand, when description 2 has to be carried out in a single cycle step, only one adder is needed. This is because the conditional ($i && j$) is resolved within the cycle step and the two add operations then become mutually exclusive. They are shown in figures 6.3(a) and (b).
6.2 Condition Vector List Scheduling

The List Scheduling method [McFa86] can be extended to deal with nested conditional branches through use of the CV concept. Such extension is called CV based list scheduling (CVLS). The CVLS can achieve both conditional sharing and optimization of all possible paths. Scheduling and control synthesis techniques are both based on CVs.

6.2.1 Module utilization vector

In order to keep track of the usage for various types of modules, the Module Utilization Vector [Waka89] is introduced: MUV $muv(l, t)$, each component of which represents the number of modules $l \in Modtype$ used in the cycle step $t \in T$ under each actual CV condition. $muv(l, t)$ is the vector sum of all actual CVs for all operations which are assigned to a module $l$ in cycle step $t$.

$$muv(l, t) = \sum_{v \in V} \{ a_v | \xi(v) = l \land t \in [begin(v), end(v)] \}$$

The largest component of MUV shows the necessary number of corresponding function units. In addition, MUV is used for optimizing all possible paths. The number of possible paths for a conditional tree is equal to the dimension of its CVs. Here, MUVALL $muvall(t)$ is the vector sum of the actual CVs of all operations scheduled in cycle step $t$, i.e.:

$$muvall(t) = \sum_{l \in Modtype} muv(l, t)$$

If any components of a MUVALL is '0', the cycle step can be skipped under '0' component conditions, because the '0' components show that no operation is performed under those conditions. The control sequence for all possible paths are generated by using such characteristics of MUVALL.

6.2.2 Priority function

A conventional priority function for list scheduling is calculated based on the path length from the operation node to the output node. In the proposed algorithm, however, the longest path in the conditional branches is not a good priority function, because shorter paths in the conditional branches are achieved in shorter steps than the longer paths. Moreover, the effect of shortening high occurrence probability branches is higher than that of shortening low occurrence probability ones.

The priority function $p_f$ [Waka89] is defined as an inner product of probability vector and distance vector, which are defined in the following. If no conditional branches exist, the $p_f(v)$ is equal to the longest path length from node $v$ to an output node. The priority function is defined as:

$$p_f(v) = (p_v, d_v) = \sum_j p_v[j] \cdot d_v[j],$$

where $p$ is the probability vector whose components denote the occurrence probabilities for each leaf condition. In the example of 4.1, if all conditions $c1$, $c2$, $c3$ are satisfied by 50%,
the probability vector becomes \([1/4, 1/8, 1/8, 1/2]\). The distance vector \(d\) is the sum of extended CVs for all operation nodes in the path from successor of node \(v\) to the output node. Distance vector \(d_v\) values are calculated in sequence from the output node to node \(v\). The largest component of \(d_v\) is adopted at the node where several paths merge. If there are no conditional branches, the distance vector is equal to the longest path length. Figure 6.4 shows an example of the calculation of the priority function. The conditions \(c1\) and \(c2\) are satisfied by 50%, thus the probability vector becomes \([1/2, 1/4, 1/4]\).

6.2.3 CVLS algorithm for a single nested branch

The scheduling algorithm is for a behavior description containing single top level nested conditional branches. Let the ready list be a list of nodes whose data and control dependencies have been satisfied. Thus, they are ready for cycle step assignment. The scheduling algorithm can be summarized as follows.

1. Calculate ECVs and priority functions \(pf\) for all nodes and let the current cycles step \(k\) be the first cycle step 0.

2. Compile the ready list for cycle step \(k\).
   The remainder nodes are contained in the rest list.

3. Take \(u\) in the ready list with largest \(pf(u)\).

4. If \(u\) is a successor to a join node, the invoke the operation node dividing procedure (see section 6.3.2).
6.2 Condition Vector List Scheduling

5. If the largest component for \( m_{uv}(l, k) \) does not exceed the number of the available function units, then assign \( u \) to cycle step \( k \), otherwise move \( u \) to the rest list \( l \in ModType \land \tau(l) = \tau(u) \).

6. If the ready list is not empty, go to (3).

7. If the rest list is not empty, then let the current cycle step be the next cycle step and go to (2).

8. Invoke the operation node re-assignment procedure (see section 6.3.5).

6.2.4 Multiple conditional branches

Operations in different branches are not mutually exclusive and they cannot share hardware. Therefore, the CVLS algorithm first exploits alternation within a branch, and then exploits parallelism among branches. Individual top level branches are scheduled in turn. If the first branch didn’t use all the modules in all cycle steps, then the remaining modules can be available when scheduling the next branch.

6.2.5 Control structure transformation

The basic CV can’t be determined for multiple branches within a branch (figure 6.5(a)). Operations in branch (c1) and branch (c2) are not mutually exclusive. In this case, the basic CVs for these operations can’t be determined by the former definition. Therefore,

\[
\begin{align*}
  \text{if } (a > b) & \text{ then } A = (a > b); \\
  a++; & \\
  \text{if } (c1) & \text{ then } x = a + b; \\
  \text{else} & x = a - b; \\
  \text{fi} & \\
  \text{if } (c2) & \text{ then } y = a + 1; \\
  \text{else} & \Rightarrow \\
  \text{fi} & \\
\end{align*}
\]

\[
\begin{align*}
  \text{if } (A) & \text{ then } \\
  a++; & \\
  \text{if } (c1) & \text{ then } x = a + b; \\
  \text{else} & x = a - b; \\
  \text{fi} & \\
  \text{if } (c2) & \text{ then } y = a + 1; \\
  \text{else} & y = b + 1; \\
  \text{fi} & \\
\end{align*}
\]

(a) Behavior description (b) Modified behavior description from (a)

Figure 6.5: Transformation of control structure

such kind of branch is converted into multiple basic branches, in which operations can have its basic CVs. The converted description is the description (b) in figure 6.5. These descriptions produce the same outputs and the modified description can be scheduled as multiple conditional branches.
6.3 Control synthesis

6.3.1 Control sequence synthesis method

Some of the conventional scheduling methods which analyze data flow graphs and schedule all branches at the same time for exploiting potential parallelism, can share resources among alternative branches [Park86, Paul87]. However, most of them generate a single sequence for all branches, in other words, operations assigned to cycle steps are processed in sequence from the earliest cycle step to the latest.

However, each branch of conditional branches should have its individual sequence suitable for itself. This could be done by scheduling each path separately. Nevertheless, such approaches are weak in exploiting global potential parallelism. In addition, they cannot search for the possibilities that branches are executed before the conditionals are resolved.

A more efficient control sequence can be generated for control data flow graphs containing conditional branches by making use of the module utilization vector muvall. The muvall(t) indicates the number of modules of all kinds of operations to be executed in cycle step t. Therefore if some components of muvall(t) are zero, no operations will be carried out in cycle step t under the conditions corresponding to the '0' components of the MUVs. Consequently, cycle step t can be skipped under such conditions. If for example muvall(t) = [0, 0, 1] then cycle step t can be skipped under the conditions corresponding to the first and second components for MUVA.

6.3.2 Operation node dividing

The node dividing technique minimizes the number of cycle steps by transforming the data flow graph while scheduling. When an operation node is a successor of a merge node or a predecessor of a branch node, the operation can be divided into two nodes. Figure 6.6 shows how to divide a node succeeding a merge node, where the extended CV is also divided into two extended CVs. It should be noted that the required number of modules does not increase by node dividing, because the divided nodes are mutually exclusive. The divided operation nodes become more mobile than the original node. Therefore, the
control sequence can sometimes be made more efficient by re-assignment of either node into a former cycle step. If either of the divided nodes can be assigned to a former cycle step, and the number of '0' components of MUVALL \( \text{muva} \) increases by the re-assignment, node dividing is adopted. After a node is divided, the nodes succeeding the divided node becomes the successor of the merge node, then these nodes can be divided, and so on.

Another way of minimizing the number of cycle steps is shown in figure 6.7. This example shows that an unconditional operation and an operation inside conditional branches can be mutually exclusive, even if they have no relation with each other. Note that unconditional operations can be moved into any conditional tree by node dividing, while operations inside a conditional tree can only be moved into inner branches of the tree. In addition, one operation will be executed in different cycle steps for each execution instance when the node is divided.

![Figure 6.7: Non-relational operation node dividing example](image)

6.3.3 As-fast-as-possible scheduling

In order to minimize the overall number of cycle steps, operation node division is only employed if the division decreases the number of '0' components of MUVALL. Nevertheless it is possible to employ operation node division whenever a divided node can be assigned to a former cycle step regardless of the number '0' components of MUVALL. Such application of node division potentially creates faster execution paths without increasing the required number of modules. Figure 6.8 shows an example of AFAP scheduling. As the division of node \( a \) does not increase the number of '0' components of MUVALL, node division will normally not be carried out. In the case of AFAP scheduling the node division is applied anyway and consequently node \( b \) becomes a candidate for division. The division of node \( b \) will increase the number of '0' components of MUVALL resulting in a faster schedule for some execution paths. Although AFAP scheduling possible creates faster schedules for some execution paths, node division will increase the number of control signals accordingly, resulting in a more complicated control path.
6.3.4 Pre-execution of branch operations

Pre-execution of branch operations is a technique which allows operations inside a conditional branch to be executed before the conditional test operation itself, in order to save resources and minimize the number of cycle steps. Figure 6.9 shows an example of pre-execution. If the scheduler didn’t allocate operations in a branch before the conditional operation, five cycle steps are required, while in this case only 3 cycle steps are required to schedule the operations. The actual CVs express dynamic mutual exclusiveness and consequently the scheduler can neglect the control dependencies. The semantic of the given behavior is preserved by means of actual CVs. For example, the ‘+’ operation in cycles 0 and 1 are always executed. On the other hand, the ‘+’ operations in cycle 2 are only executed when the branching condition is true.

Note that an operation executed before a conditional test, can be handled as an unconditional operation. Hence, the operation moved outside the conditional tree can be divided and moved into another conditional tree by node dividing. This means that operations in different conditional trees can be mutually exclusive, and can share resources.
6.3.5 Operation node re-assignment

The control sequence is sometimes made more efficient by moving an operation node into another cycle step. If the movement of a node that has mobility into another cycle step increases the number of '0' components for $m_{wall}$, then node re-assignment is adopted. This improvement is accomplished from the end node to the start node.

6.4 A scheduling example

As an example we use the control data flow graph as shown in figure 6.10. It is assumed that available modules are an adder, a subtracter and a comparator. Nodes are sorted by $G_0$ value. We assume that $n_p$ is available in cycle step 0 and that $n_p$ and $n_q$ are true by 50% and thus the probability vector $p$ becomes $[1/2, 1/4, 1/4]$. The values of the condition vectors are shown in table 6.1. The first ready list contains $n_q$, $n_a$, $n_b$, $n_c$ and $n_k$ and according to the $pf$ value, $n_q$ is assigned first to cycle step 0, followed by $n_b$, $n_c$ and $n_q$. As $n_q$ is resolved the actual condition vectors are recalculated. Assignment of $n_k$ to cycle step 0 would exceed the number subtracters, so the ready list is recompiled for cycle step 1. The results of the CVLS algorithm are shown in table 6.2. The needed number of cycle steps without node division and operation node re-assignment for each leave condition is [5, 5, 7] and the final

![Figure 6.10: A control data flow graph scheduled by the CVLS algorithm](image)
result for each is leave condition is \([4,4,7]\). The control sequence for the scheduling result is shown in figure 6.11. Condition vectors 100, 010 and 001 correspond with conditions \(p\), \(\overline{p} \cdot q\) and \(\overline{p} \cdot \overline{q}\) respectively.

<table>
<thead>
<tr>
<th>Node</th>
<th>Ext. CV</th>
<th>Act. CV</th>
<th>Dist. V</th>
<th>pf</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>101</td>
<td>101</td>
<td>427</td>
<td>4.25</td>
</tr>
<tr>
<td>b</td>
<td>011</td>
<td>011</td>
<td>146</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>010</td>
<td>111</td>
<td>142</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>101</td>
<td>101</td>
<td>326</td>
<td>3.5</td>
</tr>
<tr>
<td>e</td>
<td>001</td>
<td>011</td>
<td>125</td>
<td>2.25</td>
</tr>
<tr>
<td>f</td>
<td>100</td>
<td>111</td>
<td>211</td>
<td>1.5</td>
</tr>
<tr>
<td>g</td>
<td>001</td>
<td>011</td>
<td>125</td>
<td>2.25</td>
</tr>
<tr>
<td>h</td>
<td>010</td>
<td>011</td>
<td>132</td>
<td>1.75</td>
</tr>
<tr>
<td>i</td>
<td>001</td>
<td>011</td>
<td>124</td>
<td>2</td>
</tr>
<tr>
<td>j</td>
<td>001</td>
<td>011</td>
<td>123</td>
<td>1.75</td>
</tr>
<tr>
<td>k</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>l</td>
<td>011</td>
<td>011</td>
<td>011</td>
<td>0.5</td>
</tr>
<tr>
<td>m</td>
<td>011</td>
<td>011</td>
<td>122</td>
<td>1.5</td>
</tr>
<tr>
<td>n</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>q</td>
<td>011</td>
<td>111</td>
<td>022</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.1: Values of condition vectors and priorityfunction for the scheduling example

Figure 6.11: Control sequence for the scheduling result

The control sequence is generated by making use of the module utilization vector MUVA. For example the resulting value of \(nuval(1)\) is 102 (see table 6.2 so cycle step 1 can be skipped under the condition corresponding to the '0' component. The same applies for cycle step 2, so cycle steps 1 and 2 can been skipped. In figure 6.11 cycle steps 1 and 2 are skipped
### 6.5 Conclusions

The CVLS algorithm can schedule operations independent of the control dependencies, such as control node dividing and pre-execution of branch operations. Furthermore, potential parallelism and alternation are exploited at the same time without increasing computational complexity. The ability for exploiting potential parallelism is based on the original scheduling algorithm. The list scheduling algorithm is not so strong in such ability, however the Condition Vector concept can be applied to other methods powerful for exploiting potential concurrency.

<table>
<thead>
<tr>
<th>Node</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>+101+</td>
</tr>
<tr>
<td>b</td>
<td>-011-</td>
</tr>
<tr>
<td>c</td>
<td>#011#</td>
</tr>
<tr>
<td>q</td>
<td>+101+</td>
</tr>
<tr>
<td>d</td>
<td>-001-</td>
</tr>
<tr>
<td>e</td>
<td>+010+</td>
</tr>
<tr>
<td>h</td>
<td>+001+</td>
</tr>
<tr>
<td>f</td>
<td>+001+</td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>+001+</td>
</tr>
<tr>
<td>j</td>
<td>+001+</td>
</tr>
<tr>
<td>k</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Scheduling results with operation node division and re-assignment under condition $p \cdot q$ which corresponds with the '0' component of the module utilization vector.
Hierarchical reduction approach

The hierarchical reduction approach [Kim91, Kim94] is a new scheduling approach for data flow graphs with nested conditional branches. The algorithm employs a bottom-up approach to transform a data flow graph with conditional branches into an "equivalent" one that has no conditional branches. A schedule is then obtained for the latter, using a conventional scheduling algorithm, from which a schedule for the former is derived. This approach is particularly effective when there is a large number of nested conditional branches in a data flow graph. The main features of the algorithm are the following:

- Conditional resources sharing is emphasized. In other words, the amount of conditional resources sharing is maximized first and then the amount of unconditional resource sharing is maximized.
- Because of the hierarchical bottom-up approach, the algorithm can handle conditional resource sharing for data flow graphs with deeply nested conditional constructs effectively.
- The hierarchical bottom-up approach makes it relatively simple to construct the finite state machine incrementally at each level of the hierarchy.
- An efficient scheduling algorithm that is suited for particular design goals and constraints can be chosen to exploit unconditional resource sharing optimally.

The algorithm consists of three steps:

1. Transform a data flow graph with conditional branches into a data flow graph without conditional branches.
   In this step, an algorithm explores the possibility of conditional resource sharing while transforming the graph. Nested conditional branches are handled in a bottom-up fashion, i.e. the innermost conditional branches are handled first, and the outermost are handled last.

2. Use a conventional scheduling algorithm to obtain a schedule for the data flow graph that contains no conditional branches.
   The resulting graph contains no conditional branches, hence any scheduling algorithm, of which the selection is based on the design goals and constraints, can be employed. The number of control steps obtained by a conventional scheduling method is an upper bound on the number of control steps in the schedule for the data flow graph with conditional branches.
3. Transform the obtained schedule into a schedule for the original data flow graph. This can be accomplished, by simply reversing the transformation performed in step 1. Furthermore, at each level of the reverse transformation, the algorithm tries to improve the schedule of the corresponding conditional block before the reverse transformation at the next level of the hierarchy is carried out.

7.1 Transformation of the data flow graph

The objective of the transformation step is to replace a conditional block by an "equivalent" non-conditional block. The non-conditional block can be viewed as a common representative of the conditional branches in the block and is constructed by exploring conditional resource sharing among the operations in the conditional branches.

7.1.1 Conditional resource sharing

Since executions of conditional branches are mutually exclusive, hardware resources can be shared between the operations in different branches. The basic transformation procedure is to select two operations of the same type, one from each of the conditional branches, and merge them together to form one node. When pairing these nodes, the length of the critical path of the block might increase. For example, in figure 7.1 if operation 1 and 11 are to share the same module, the length of the critical path of the conditional path will increase from 5 to 7 (e.g. 6, 7, 9, (1, 11), 4, 5, 12).

![Figure 7.1: An example of a data flow graph with conditional branches](image)

In order to prevent the critical path length to grow beyond a certain limit, the sharing of resources is controlled. Let $L_0$ denote the critical path length in the original conditional block. For a user defined constant $b$, $b$ cases will be examined in which the critical path...
length does not exceed $L_0 + \Delta L$ for $\Delta L = 0, 1, \ldots, b - 1$ when operations in the conditional branches are paired off. In each case, a transformation maximizes the conditional resource sharing of the conditional block. In order to select the best transformation among the $b$ cases, a cost function is defined. For a chosen value of $\Delta L$ the procedure described in the next paragraph, is used to determine a transformation of the conditional block. Let $P_t$, $t \in OpType$, denote the pairs of type $t$ operations that are chosen for conditional resource sharing in the transformation. Let $\alpha_t, t \in OpType$, be a set of user chosen weights. Next, the quantity

$$R = \sum_{t \in OpType} \alpha_t \cdot P_t$$

is computed as a weighted measure of the total amount of conditional resource sharing in the transformation. For each of the $b$ cases, the ratio

$$\frac{R}{L_0 + \Delta L}$$

is a measure of the effectiveness in increasing conditional resource sharing by increasing the critical path length. Of the $b$ cases, the transformation with the largest ratio is chosen.

### 7.1.2 A transformation procedure

For a chosen value of $\Delta L$ an algorithm iteratively pairs off operations in the conditional branches while limiting the critical path length to no longer than $L_0 + \Delta L$. Operations can be paired off only if their execution intervals overlap. All possible pairings are examined and the most 'promising' one is selected. For each possible pairing of operations an estimate of the quantity $R$ is computed. Let $\hat{R}(v_i, v_j)$ be the estimate of $R$ when operations $v_i$ and $v_j$ are to be paired off. To compute this estimate, the probability that an operation will be scheduled in a particular cycle step inside its execution interval is used (similar to [Paul89, Stok91]). For example, suppose operations 1 and 10 in figure 7.1 are to be paired off, figure 7.2(a) shows the initial execution intervals, figure 7.2(b) shows the resultant execution intervals and figure 7.3 shows the density function for each operation-type.

![Figure 7.2: Execution intervals](attachment:figure72.png)

The density function $D(t, k)$ is the sum of the probabilities of all operations of type $t$ at cycle step $k$. Let $D_n(t, k)$ denote the densities of operation-type $t$ of the $n^{th}$ conditional branch,
0 \leq n < N, at cycle step \( k, k \in T \). Then the estimate can be defined as:

\[
\hat{R}(v_i, v_j) = \sum_{t \in O} \alpha_t \sum_{k \in T} \min_{0 \leq n < N} \{D_n(t, k)\}
\]

Figure 7.3: Density functions

Let \( \alpha_+ = 1 \) and \( \alpha_- = 1 \). Among all possible pairs of operations in figure 7.1 the pair (2,9) has the largest estimate value \( \hat{R} \). Thus operations 2 and 9 will be paired off and they are merged to become one operation node (2,9), which inherits all the data dependencies of the

Figure 7.4: A sequence of pairings of operations for conditional resource sharing
two merging nodes (7.4(a)). The execution interval of the node (2,9) is the intersection of the execution intervals of operation nodes 2 and 9. In the next iteration, operations 3 and 10 are selected and paired off (figure 7.4(b)). The iteration continues until there is no operation pair with overlapping execution intervals. Figure 7.4(c), (d) and (e) show the data flow graph after the third, fourth and final iteration, respectively.

7.2 Derivation of a schedule for the original data flow graph

In the next step an arbitrary scheduling algorithm can be applied to the data flow graph obtained in the first step. The choice of the algorithm will be based on the design goals and constraints, such as execution time constraints, hardware constraints, chaining operations, multi-cycle operations and pipelining. The schedule for the data flow graph without conditional branches obtained in this step is then transformed into a schedule for the original data flow graph. This step is carried out by reversing the order in which the data flow graph without conditional branches was constructed (i.e. in a top-down fashion). Operations that were merged to form a new operation node in the nonconditional block in step 1 will share one functional unit conditionally. As an illustrative example, figure 7.5 show the data flow graph obtained from the data flow graph in figure 7.1 after all conditional blocks are transformed into nonconditional blocks. Figure 7.6(a) shows two schedules for

```
Figure 7.5: A schedule for the transformed dataflow graph
```

the two execution instances corresponding to the two possible outcomes when condition c1 is evaluated. Note, for example, the operation node 3,10,18 in figure 7.5 is split into two operation nodes 3,10 and 18, because they belong to two different conditional branches.
Operations 20, 21, 22 and 23 appear in both execution instances because they are global to the conditional branches. Now the schedule for the right conditional branch can be modified to reduce the number of cycle steps by 2, as shown in figure 7.6(b). The same procedure is repeated for the left conditional branch, as shown in figure 7.6(c). Notice that operation 22 is now scheduled in control step 4 when the left conditional branch is executed and in control step 3 when the right conditional branch is executed. Consequently, an operation might be executed in different control steps for different execution instances which leads to a reduction in the number of cycle steps. However, the number of control signals will increase accordingly, resulting in a more complicated control path. Figure 7.7 shows the final schedule for the original data flow graph. The number of needed cycle steps for the different execution paths are 6, 6, 5.
7.3 Generation of the Finite State Machine

A finite state machine which controls the order of execution of the operations is constructed incrementally during the reverse transformation step. Since the data flow graph to be scheduled in step 2 contains no conditional branches, the state transitions of the corresponding finite state machine are simply straight sequences of transitions, beginning at the first control step and ending at the last one, with no conditional transitions. When a non-conditional block is transformed into a conditional block, the state transitions are modified according to the corresponding execution schedules in the conditional block.

7.4 Conclusions

Although the bottom-up hierarchical approach is computationally more efficient when compared with global non-hierarchical ones, resource sharing is, for the same reason, not exploited to the fullest. The transformation sequence of a data flow graph with conditional branches into a data flow graph without conditional branches is not unique. Different sequences of transformation will result in different data flow graphs without conditional branches and consequently possibilities for resource sharing may be overlooked. Because of the incremental synthesis procedure for finite state machines a way is provided to control trade off between scheduling results and the cost of the control path.
8.1 Comparison

The comparison of the different scheduling methods is difficult because the objectives of the techniques used are different. The algorithm of TASS [Ame94] performs a global optimization using area costs of hardware resources and it optimizes conditional resource sharing using the new mutual exclusion testing method as described in section 4.4. This scheduling method is not capable to schedule an operation into different control steps for different execution instances. The path-based As-Fast-As-Possible (AFAP) scheduling algorithm deals mainly with applications with many conditional branches and loops that emphasize fast schedules. The scheduling method based on condition vectors exploits a more "global parallelism". That is, it parallelizes multiple nests of conditional branches and optimizes across the boundaries of basic blocks. The hierarchical approach handles a dual problem: optimization of hardware cost under a given execution time constraint. Critical

<table>
<thead>
<tr>
<th>Data</th>
<th>Chain</th>
<th>Method</th>
<th>Adds</th>
<th>Subs</th>
<th>States</th>
<th>max. cycles/min. cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>TASS</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Park86</td>
<td></td>
<td>Condition vector</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5/2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hierarchical</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8/3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>TASS</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Critical</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AFAP</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>5/2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Condition vector</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>4/1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hierarchical</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>5/2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Critical</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Condition vector</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3/1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hierarchical</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3/2</td>
</tr>
<tr>
<td>Waka89</td>
<td>1</td>
<td>Condition vector</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7/4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hierarchical</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7/4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>AFAP</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>7/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Condition vector</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7/3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hierarchical</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7/3</td>
</tr>
</tbody>
</table>

Table 8.1: Scheduling results for different scheduling algorithms
path first scheduling [Park86] is a "classical" non-conditional scheduling algorithm. The results of the different algorithms are listed in table 8.1.

8.2 Conclusions

The objective of path-based scheduling is to yield solutions with a minimum number of cycle steps for all execution paths, under given constraints such as timing and area. In order to yield such solutions operations may be scheduled into different control steps for different execution instances. However, the computational complexity of the algorithm depends on the number of the execution instances, which could be exponential. Furthermore, for each execution instance, the order in which the operations are to be executed must be chosen in advance. Thus execution instances are straight line sequences of operations; consequently implicit concurrencies in the data flow graph are often overlooked and the scheduling result strongly depend on the chosen order.

The Condition Vector List Scheduling algorithm uses a condition vector to identify mutual exclusive operations. The algorithm can exploit "global parallelism", i.e. can deal with top level conditional branches. Furthermore the algorithm can schedule operations independent of the control dependencies, such as operation node dividing and pre-execution of branch operations. These techniques minimize the number of cycle steps, by allowing operations to be scheduled in different control steps for different execution instances and by allowing operations inside a conditional branch to be executed before the conditional test operation itself, respectively. Limitations of the algorithm are that it cannot handle certain types of conditional branches correctly and it doesn't minimize the path length globally.

The hierarchical reduction approach has the advantage, that an efficient scheduling algorithm that is suited for particular design goals and constraints can be chosen to exploit unconditional resource sharing optimally. Furthermore, the computational complexity is low in comparison with the other methods. Because of the bottom-up transformation method, mutual exclusiveness of operations is not exploited to the fullest. Also, the algorithm cannot exploit sufficient potential parallelism, because it cannot move an operation inside a conditional block to a cycle step outside of it, even if there are available resources left.

Based on these conclusions and the results listed in table 8.1 the CVLS algorithm has been implemented on the NEAT++ [Heij94] system. The limitation that certain types of conditional branches cannot be handled correctly can easily be overcome by transforming the control structure. Furthermore the CVLS algorithm is best suited to make a trade off between different scheduling results and the complexity of the control path. Some results of the CVLS algorithms are listed in table A.1.

8.3 Future work

Some topics of interest that are related with scheduling of conditional branches are the following.
8.3 Future work

- In order to obtain high quality designs, trade offs between the scheduling result and the cost of the control path should be investigated.
- Register sharing among mutually exclusive variables.
- Sharing of interconnections among mutually exclusive data transfers in conditional branches.

At this stage the most important aspect to look at next is how the different scheduling methods affect the complexity of the control path and to find a measurement that can attribute to make a trade off between different scheduling results and the cost of the control path. The path-based AFAP scheduling algorithms is not well suited to handle this problem because it schedules each execution path independently.
Appendix A

Results of the CVLS algorithm

Table A.1 shows some results of the CVLS algorithm for some arbitrary data flow graphs ex1, ex2, ex3 and ex4 and are shown in figures A.1, A.4, A.7 and A.10 respectively.

<table>
<thead>
<tr>
<th>Data</th>
<th>Constraint</th>
<th>Method</th>
<th>Cycles</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>control seq.</td>
<td>straight</td>
</tr>
<tr>
<td>ex1</td>
<td>#adder: 1</td>
<td>-</td>
<td>[5,5,7]</td>
<td>[7,7,7]</td>
</tr>
<tr>
<td></td>
<td>#sub: 1</td>
<td>nodediv</td>
<td>[4,5,7]</td>
<td>[4,7,7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full</td>
<td>[4,4,7]</td>
<td>[4,7,7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>afap</td>
<td>[4,4,7]</td>
<td>[4,4,7]</td>
</tr>
<tr>
<td>ex2</td>
<td>#Add/Sub: 1</td>
<td>-</td>
<td>[7,7]</td>
<td>[7,7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nodediv</td>
<td>[5,7]</td>
<td>[7,7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full</td>
<td>[5,7]</td>
<td>[7,7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>afap</td>
<td>[5,7]</td>
<td>[5,7]</td>
</tr>
<tr>
<td>ex3</td>
<td>#Add/Sub: 2</td>
<td>-</td>
<td>[7,8,9]</td>
<td>[9,9,9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nodediv</td>
<td>[6,8,9]</td>
<td>[9,9,9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full</td>
<td>[6,6,6]</td>
<td>[6,6,6]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>afap</td>
<td>[6,7,9]</td>
<td>[6,7,9]</td>
</tr>
<tr>
<td>ex4</td>
<td>#Add/Sub: 4</td>
<td>-</td>
<td>(16/12)</td>
<td>(16/16)</td>
</tr>
<tr>
<td></td>
<td>#Mult: 3</td>
<td>nodediv</td>
<td>(16/11)</td>
<td>(16/16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full</td>
<td>(12/10)</td>
<td>(12/10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>afap</td>
<td>(16/8)</td>
<td>(16/13)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>afap+</td>
<td>(12/8)</td>
<td>(12/8)</td>
</tr>
</tbody>
</table>

Table A.1: Scheduling results for the CVLS algorithms

Methods:
- "nodediv": operation node division and re-assignment,
- "full": operation node division, pre-execution and re-assignment,
- "afap": As-Fast-As-Possible,
- "afap+": As-Fast-As-Possible and pre-execution.

Cycles:
- control seq: number of needed cycle steps with control sequence synthesis,
- straight: number of needed single sequence cycle steps.
Notation:

\[ [x,x,\ldots] \]: number of needed cycle steps for each execution, instance,
\( (x,x) \): maximum and minimum number of needed cycle steps.

The resulting data flow graphs for example 1, example 2 and example 3 for the methods "nodediv" and "afap" are shown in figures A.2, A.3, A.5, A.6, A.8 and A.9 respectively.

The execution times (CPU on a HP 700/RX) vary from 0.7 seconds for example 1 and 27 seconds for example 4.
Figure A.1: Data flow graph of example 1
Figure A.2: Resulting data flow graph for example 1 with operation node dividing

Figure A.3: Resulting data flow graph for example 1 with operation node dividing and afap scheduling
Figure A.4: Data flow graph of example 2
Figure A.5: Resulting data flow graph for example 2 with operation node dividing

Figure A.6: Resulting data flow graph for example 2 with operation node dividing and afap scheduling
Figure A.7: Data flow graph of example 3
Figure A.8: Resulting data flow graph for example 3 with operation node dividing

Figure A.9: Resulting data flow graph for example 3 with operation node dividing and afap scheduling
List of Figures

3.1 Example of conditional statements ........................................ 7
3.2 Representation of control nodes ........................................... 8
3.3 Conditional representations ................................................ 8

4.1 Basic CV for nested if construct .......................................... 11
4.2 Example of node coloring, condition vectors and node tagging ........ 12
4.3 Conditional graph example .................................................. 13
4.4 Example of branch numbering ............................................... 14
4.5 Calculate Branch Number ...................................................... 15

5.1 Application of As-Fast-As-Possible scheduling algorithm ............... 18
5.2 Constraint graphs and cliques .............................................. 19
5.3 Paths and cuts ....................................................................... 20
5.4 Cut graph and final cutting points ......................................... 21
5.5 Finite State Machine ............................................................. 22

6.1 Condition vector example ...................................................... 24
6.2 Algorithm to calculate extended condition vectors ....................... 25
6.3 Datapath for sample description ............................................ 26
6.4 Example of the priority function ........................................... 28
6.5 Transformation of control structure ........................................ 29
6.6 Relational operation node dividing ........................................ 30
6.7 Non-relational operation node dividing example ........................ 31
6.8 As-Fast-As-Possible scheduling ............................................ 32
6.9 Pre-execution of an operation in a branch ................................ 32
6.10 A control data flow graph scheduled by the CVLS algorithm ......... 33
6.11 Control sequence fot the scheduling result ................................ 34

7.1 An example of a data flow graph with conditional branches ........... 38

59
List of figures

7.2 Execution intervals .................................. 39
7.3 Density functions ................................... 40
7.4 A sequence of pairings of operations for conditional resource sharing .... 40
7.5 A schedule for the transformed dataflow graph ............................ 41
7.6 A sequence of steps for deriving a schedule for the original dfg .......... 42
7.7 A final schedule for the original data flow graph ......................... 43

A.1 Data flow graph of example 1 ................................ 51
A.2 Resulting data flow graph for example 1 with operation node dividing ... 52
A.3 Resulting data flow graph for example 1 with operation node dividing and
    afap scheduling ........................................ 52
A.4 Data flow graph of example 2 ................................ 53
A.5 Resulting data flow graph for example 2 with operation node dividing ... 54
A.6 Resulting data flow graph for example 2 with operation node dividing and
    afap scheduling ........................................ 54
A.7 Data flow graph of example 3 ................................ 55
A.8 Resulting data flow graph for example 3 with operation node dividing ... 56
A.9 Resulting data flow graph for example 3 with operation node dividing and
    afap scheduling ........................................ 56
A.10 Data flow graph of example 4 ................................ 57
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Values of condition vectors and priority function for the scheduling example</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>Scheduling results with operation node division and re-assignment</td>
<td>35</td>
</tr>
<tr>
<td>8.1</td>
<td>Scheduling results for different scheduling algorithms</td>
<td>45</td>
</tr>
<tr>
<td>A.1</td>
<td>Scheduling results for the CVLS algorithms</td>
<td>49</td>
</tr>
</tbody>
</table>
Bibliography


