Global Scheduling Independent of Control Dependencies Based on Condition Vectors

Kazutoshi WAKABAYASHI
C&C Systems Research Laboratories, NEC Corp.
4-1-1 Miyazaki Miyamae-ku Kawasaki, Japan
wakaba%btl.cl.nec.co.jp@sj.nec.com

Hirohito TANAKA
NEC Scientific Information System Development Ltd.
R&D 637, KSP, 3-2-1 Sakado, Takatsu-ku
Kawasaki, Japan

Abstract

This paper presents a global scheduling method based on *condition vectors*. The proposed method exploits a more "global parallelism". That is, it can parallelize multiple nests of conditional branches and optimize across the boundaries of basic blocks thoroughly, and so on. Moreover, it can optimize all possible execution paths. Also proposed is an algorithm which generates a single finite state machine controller from parallel individual control sequences derived in the global parallelization process. Experimental results proves that the global parallelization is markedly effective.

1 Introduction

In order to minimize the number of states in the final schedule for a given behavior, it is very important to exploit both parallelism and alternation among operations. Some methods such as the Force-directed [2], the Sehwa [3], the Bridge [4] and Kim's methods [5], are able to accomplish conditional resource sharing for mutually exclusive operations. However, they cannot exploit sufficient potential parallelism, because they cannot move an operation inside a fork-join pair to a control step outside of it, even if there are available resources left.

On the other hand, path-based scheduling [6] and percolation scheduling[7] can somewhat exploit parallelism across basic blocks boundaries. Though path-based scheduling finds all possible execution paths and optimizes each of them, the fundamental order of operations for a given path is chosen in advance, then the scheduling result may not be sufficiently optimized across the basic limit boundaries. "Percolation" scheduling starts with optimal schedule and applies semantics-preserving transformations. It can extract parallelism beyond the basic block limits. However, it does not consider conditional resource sharing well, and it optimizes only the total number of control steps, not all possible paths.

Moreover, no conventional method can exploit "global parallelism"; no method deals with parallelization of top level conditional branches.

We previously proposed a List Scheduling method based on Condition Vectors (CVLS) [9]. This method can deal with single nested conditional branches systematically. It can accomplish conditional resource sharing thoroughly, and can optimize all possible execution paths. However, this method did not overcome all of the above problems.

In order to overcome all of them, we have developed some essential new improvements to the CVLS method. These new techniques can schedule operations independent of control dependencies. In other words, they transform control structure of the given behavior drastically, while preserving semantics, in order to minimize the number of states in final schedule.

2 Preliminaries

2.1 Condition Vectors and Varieties of Conditional Mutual Exclusiveness

We introduced the **Condition Vector (CV)** concept in [9]. CVs show the execution conditions of an operation in a vector form, which is a 1-hot encoding for all leaf branches in the behavior description, the value of CV for a particular operation is the OR-ed result of all leaf branches reachable from that operation. Mutual exclusiveness is derived by means of a CV associated with each operation. The CVs are defined for each top level nest of conditional branches, called a "**conditional tree**". Examples of CVs for a conditional tree are shown in Fig.1. The "basic" column shows basic CVs for each operation, which expresses explicit mutual exclusiveness. If CVs have no common '1' bit, they are mutually exclusive, such as (3) and (4).

The scheduling method proposed in this paper allows operations inside a conditional branch to be executed before the condition test operation itself, in order to save resources and states. In that case, the operations are *always* executed and afterward the result of operations will be used or abandoned according to whether the conditional is true or false. Therefore, the execution condition of operations in a branch changes according to whether it is executed before the conditional operation or after. CVs can express such a dynamic execution condition. In the Figure, the execution condition of operations in a branch will change on the basis of whether or not condition test operations in (2) and (4) are previously executed. Columns (a) (b) and (c) show three possible CV values for each operation. Such a *dynamic* CV can be obtained as the bitwise OR-ed result between a operation and the "not done" condition test operation. In column (c), if conditional (2) is "not done", the values of CV for all operation becomes [111], regardless of whether conditional (4) is 'done' or 'not'.

Note that the value of operation of (1) in column (a) is [1,0,1], while that in 'basic' column is [1,1,1]. The variable 'x' is not used under the condition for executing branch (5)(CV=[0,1,0]), therefore it is not necessary to calculate the value of 'x'. Thus, the actual execution condition is expressed as [1,0,1]. This shows the operations in (1) and (5) are actually mutually exclusive when conditionals (2) and (4) are "done", which is called *data-dependent mutual exclusiveness*.

Similar solutions for detecting mutual exclusiveness in vector form are adopted in Sehwa[3] and R^2S[8]. However, they can detect neither data-dependent mutual exclusiveness nor dynamic mutual exclusiveness. Moreover, for a more significant difference, CVs can be used for optimizing individual execution paths, as well as resource sharing as in the following section, while vectors in Sehwa and R^2S are used only for resource sharing.

		basic	(a)	(b)	(c)
$x = a+b-c+d;$	-(1)	[111]	$[1\underline{0}1]$	[111]	[111]
if $(a \neq 0)$	-(2)	$[\underline{111}]$	[111]	[111]	[111]
$y = x + c;$	-(3)	[100]	[100]	[100]	[111]
else if $(a + b < c)$	-(4)	$[\underline{011}]$	[011]	[011]	[111]
$y = c + d;$	-(5)	[010]	[010]	$[01\underline{1}]$	[111]
else $y = x + d;$	-(6)	[001]	[001]	$[0\underline{1}1]$	[111]
Conditionals:		$a \neq 0$ -(2)	done	done	not
		$a+b<c$ -(4)	done	not	*don't care*

Figure 1: Example description and its CVs

2.2 Scheduling and control synthesis for a conditional tree

The List Scheduling 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.

In order to to keep track of the usage for various types of FUs, we introduced Function unit Utilization Vector : $\mathbf{FUV}_{\mathrm{FU}\,f}$(c-step k), each component of which represents the number of FU f used in the c-step k under each CV condition. $\mathbf{FUV}_f(k)$ is the vector sum of CVs for all operations in the c-step k which are assigned to the FU f in the scheduling. The value of the largest component of $\mathbf{FUV}_f(k)$ shows the necessary FU f number in c-step k.

In addition, FUV is used for optimizing all possible paths. The number of possible paths for a conditional tree is equal to the dimensions of its CVs. Here, **FUVALL**(c-step k) is the vector sum of the CVs of any kinds of operations scheduled in a c-step k; i.e. $\mathbf{FUVALL}(k) = \sum_{f \in allFUs} \mathbf{FUV}_f(k)$. If any components of a FUVALL is '0', the c-step k can be skipped under '0' component conditions, because the '0' components show that no operation is performed under those conditions. The control sequences for all possible paths are generated by using such characteristics of FUVALL. (See [9] for a more detailed discussion.)

Fig.4 demonstrates the above discussions. There are two conditional trees CT1 and CT2, and an unconditional operation (3) in (a). The scheduling result for the CT1 and the unconditional operation is shown in (CT1) of Fig.4 (b). The maximum components of FUV+ and FUV- can be seen to lie within the constraint (two adder and one subtractor) shown in "R0" columns. Any c-step whose FUVALL has a '0' component can be skipped under the condition for '0' components. Thus, the generated control sequence becomes "FSM for CT1" in Fig.4(c).

3 Scheduling independent of control dependencies

3.1 Operation Node Dividing Technique

This section presents a "node dividing technique", by which an unconditional operation can be duplicated and moved inside conditional branches, if extra resource sharing

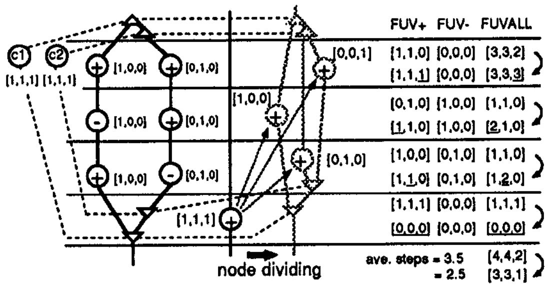
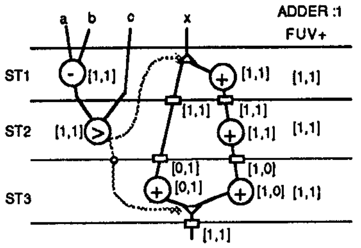


Figure 2: Node dividing technique



```
if (a - b > c)
    t = x + y + z + u ;
else
    t = x + 2 ;
```

Figure 3: Pre-execution of an operation in a branch

is possible. This technique minimizes the number of c-steps.

Fig.2 shows a typical example. The scheduling result without node dividing, requires [4,4,2] c-step for each path. If the unconditional node, whose CV is [111], is divided into tree nodes whose CVs are [100], [010] and [001], the scheduling can be improved into [3,3,1] c-steps. 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.

Such node dividing can be applied for any operation node whose CV has plural '1' components. If a node dividing makes length of some paths shorter, the dividing will be adopted. Node dividing is applied when a node becomes a candidate for allocation in List Scheduling as in the Section 4. Note that unconditional operation can be moved into any conditional trees by node dividing, while operations inside a conditional tree can be moved into only inner branches of the tree. In addition, one operation will be executed in different c-steps for each execution instance, when the node is divided.

3.2 Execution of operations inside a branch before the condition test

This section presents a scheduling technique which allows operations inside a conditional branch to be executed before the conditional test operation itself, in order to save resources and the number of c-steps. Fig.3 is an example showing scheduling under one adder and one subtractor. CVs of operators and registers are shown beside them. The data lines for y, z, u and 2 are omitted. Only three c-steps are required to execute the entire behavior, while, if the scheduler didn't allocate operations in a branch before the conditional operation, five c-steps are required.

As explained in Section 2, CVs express dynamic mutual exclusiveness. Consequently, the scheduler can neglect the control dependencies. The semantic of the given behavior is preserved by means of dynamic CVs. For example, the '+'[1,1] operations in ST1 and ST2 are always executed. On the other hand, the result of '+' in ST1 is always registered (which CV is [1,1]), but the result of '+' in ST2 is registered only when the branching condition is true because the registers CV is [1,0].

Note that an operation executed before a conditional test, can be handled as 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.*

3.3 Conditional tree duplication

At the first stage of the scheduling, "conditional tree duplication" is optionally performed, which duplicates a tree and move them into the previous tree as long as the number of leaves of the combined tree is less than 32, because a CV is implemented as one word. This could parallelize conditional trees. The following shows an example of the duplication: CT1 and CT2 are combined into CT1-2.

$$\begin{array}{lll}
\text{CT1: if (c1)} & [10] & \text{CT1-2: if (c1) if (c2)} \quad [1000] \\
\text{else} & [01] \rightarrow & \text{else} \quad [0100] \\
\text{CT2: if (c2)} & [10] & \text{else} \quad \text{if (c2)} \quad [0010] \\
\text{else} & [01] & \text{else} \quad [0001]
\end{array}$$

3.4 Parallelization of multiple conditional trees

Previous sections presented how to exploit mutual exclusiveness among conditional trees and unconditional operations. This section presents how to exploit potential parallelism among multiple conditional trees. The proposed scheduling method parallelizes conditional trees, independent of control dependencies, while preserving data dependencies and the semantics of the original design. For instance, in Fig.4(a), some plus operations in CT1 and CT2 can be executed concurrently without violating the data dependencies between them; 'x' in this case. ([1] denotes a vector whose components are all '1'.)

Conditional trees are scheduled in the order of the control flow of the given behavior. Operations in the first conditional tree are scheduled first, under the given resources constraints. When the first tree didn't use all FUs in any c-steps, some FUs remain in the c-steps. Thus, the remaining FUs can be available for the next conditional tree. Then, operations of the next tree are scheduled under the remaining FUs. Namely, operations of each conditional tree are scheduled under various resource constraints in each c-step.

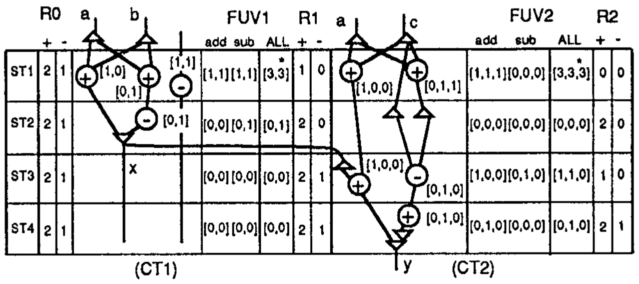
Fig.4(b) shows a scheduling result for (a) under two adders and one subtractor. (In the figure, FUVALL in ST1 contains CVs for $c1, c2, c3$.) First, operations in the conditional tree CT1 are scheduled. The number of remaining FUs in each c-step are shown in column 'R1'. Next, operations in the second tree CT2 are scheduled, using the remaining FUs: R1. Unconditional operations are schedules with any conditional trees. They are scheduled according to the priority function, and when all operations in a conditional tree are scheduled, the remaining unconditional operations will be scheduled with the next branch. (See Section 4.)

3.5 Control sequence synthesis for globally parallelized branches

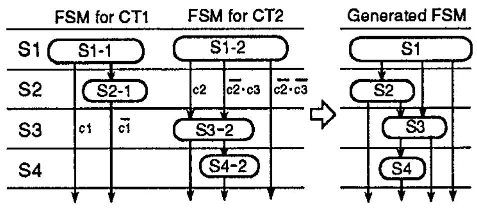
This section presents how to synthesize and how to optimize the control sequence for the parallelized conditional trees as in the previous section. Each conditional tree has an active execution path, therefore, parallel conditional

```
CT1: if (c1)      x = a + b;              -(1) [1,0]
     else         x = a + b - c;          -(2) [0,1]
     z = a - b;                           -(3) [1]
CT2: if (c2)      y = a + c + x;          -(4) [1,0,0]
     else {       y = a + c;              -(5) [0,1,1]
            if (c3) y = y - c - d; }-(6) [0,1,0]
```

(a) Multiple conditional trees



(b) Parallelization of multiple conditional trees in (a)



(c) Control sequence for the scheduling result shown in (b)

Figure 4: Scheduling of multiple conditional trees

trees requires the simultaneous activation of as many paths as the number of them. "FSM for CT1" and "CT2" in Fig.4 (c) shows control sequences for the scheduling of CT1 and CT2 in (b). Note that these two control sequences are dependent, since there are some data dependencies and some resources conflicts between them. Hence, the control for the parallelized conditional trees cannot be realized in two independent FSMs. A FSM shown in right-hand in (c) shows the control sequence for the parallelized trees. The FSM is obtained by combining the two FSMs expressing sequences of individual conditional trees so that they could move synchronously. That is, states in the same c-step for the two FSMs should be executed at the same time.

Such combined FSM can be generated by determining the transition destination and its condition from each c-step. They are simply determined by enumerating all possible combinations of paths, but the number of the combination is equal to the product of the number of each path and optimizing them. Then, we propose better generation method. The method determines the nearest transition destination and its condition first. In the figure example, they are 'S2 ' and $\overline{c1}$. Under *other* conditions, it can transit to a farer state. Second nearest destination is 'S3', and the condition is '$c2 + \overline{c2} \cdot c3$'. By this way, the state transition from ST1 (*DS1*) becomes the left-hand in the following:

$$\begin{array}{lll}
DS1 = & \text{if } (\overline{c1}) \qquad \rightarrow S2; & \text{if } (\overline{c1}) \rightarrow S2; \\
& \underline{\text{else}} \text{ if } (c2 + \overline{c2} \cdot c3) \rightarrow S3; \quad \rightarrow & \text{else} \quad \rightarrow S5; \\
& \underline{\text{else}} \qquad \rightarrow S5; &
\end{array}$$

The destinations for the other states can be determined in the same way. If there are some relations between conditional test operation $c1, c2, c3$, the FSM can be optimized more. For example, if a relation such as '$c1 \cdot c2 = 0$' and '$c1 \cdot \overline{c3} = 0$'holds, then state transition *DS1* is optimized as the right-hand in the previous example. In this way, the proposed method can deal with the "false path" problem described in [6] by using such boolean relations among conditional test operations.

4 Entire Scheduling Algorithm

```
conditional tree duplication; /* option */
$R_{FU_f}$ = number of FU $f$;
for $CT$ = the first conditional tree to the last do {
  Irest = operations inside $CT$;
  Orest = unconditional operations;
  $Cstep = 0$;
  while (Irest $\neq \phi$) {
    $Cstep = Cstep + 1$;
    compile ReadyList for $Cstep$ from Irest and Orest;
    Irest, Orest = the remainder operations;
    while (ReadyList $\neq \phi$) {
      $n$ = operation with the largest priority;
      calculate dynamic CV of $n$ in $Cstep$;
      if $n$ is dividable (CV has plural '1') then
        divide $n$;
        if the schedule becomes better then
          allocate the divided nodes in the former c-step
          goto L1;
      if $max(FUV_f) \leq R_f$ then  allocate $n$ to $Cstep$
      else Irest or Orest = $n$;
    L1:
      /* for operation chaining */
      if $n$ is allocated then renew Readylist, Irest,Orest;
    }
  }
  $R_{FU_f} = R_{FU_f}$ - number of used FU $f$ for $CT$;
}
/* node re-allocation */
for $Cstep$ = the last c-step -1 downto the first do
  re-allocate $n$ to the later c-step if '0' of FUVALLL increases;
Control sequence synthesis for the scheduling result;
```

5 Experimental results and remarks

The proposed method is implemented in *Cyber* system [10]. Table.1 shows comparisons with "MAHA" in [1], "PATH" in [6], "KIM" in [5], "R^2S" in [8] and our method "CVLS". In the table, "maha", "kim" are examples in [1] and [5], and "parker' is parker1986 in the HLS benchmark1991. "Wmaha" and "25maha" are obtained by duplicating "maha" twice and twenty five times respectively, and connecting them in series (we put one data dependency among each "maha" data). "Paths" shows the length of longest and shortest path and the average where all conditionals have equal probabilities. "Full" shows results for the proposed method. "Not par" shows results without parallelization of multiple conditional trees and node dividing, and "not cnd" shows results also without branch operation pre-execution.

Examples "maha" and "parker" have only two conditional trees, but our method gives significantly better results than other methods. Note that our method produce good results without chaining, this shows our method has high ability to exploit potential parallelism. Our method gives a better result also for "Kim". "Kim" has only one conditional trees, therefor the result shows the ability of node dividing and pre-execution of branch operations are effective. The results for "Wmaha" and "25maha" show the powerful ability to exploit potential parallelism of CVLS. The "full" results are much better than "not par" and "not cnd". In addition, the path length for "25maha" is far less than 25 times of maha. Moreover, the path length in "full" drastically decreases according to the increase of FU numbers, on the other hand, that in "not cnd" decrease slightly. These results prove that the CVLS scheduling can utilize FUs sufficiently.

The CVLS algorithm is so simple that it runs fast. The behavior containing more than one thousand operations can be scheduled in about a minute on 28.5 MIPS work station.

Table 1: Scheduling Results

Data	Method	Constraint			Results: Paths max/min(ave.)			
		add	sub	Ch	St	full	not par	not cnd
maha	KIM	1	1	1	8	8/3		
	CVLS	1	1	**1**	**5**	**5/2**(3.31)	6/3(4.12)	8/3(4.62)
	MAHA	1	1	2	8	8		
	PATH	1	1	2	9	5/2		
	KIM	1	1	2	6	5/2		
	MAHA	2	3	3	4	4		
	KIM	2	3	3	3	3/2		
	CVLS	2	3	3	3	3/1(1.56)	3/3(3.00)	4/3(3.25)
	CVLS	2	3	**1**	4	**4/1**(2.38)	5/3(3.75)	8/3(4.62)
	R^2S	2	1	2	4	4/4		
	CVLS	2	1	2	4	4/1(2.38)	4/3(3.38)	5/3(4.00)
parker	CVLS	1	1	1	5	5/2(3.31)	6/3(4.12)	8/3(4.62)
	CVLS	2	3	1	4	4/1(2.38)	5/3(3.75)	8/3(4.62)
kim	KIM	2	1	1	7	7/6(6.25)		
	CVLS	2	1	1	**6**	**6/5**(5.75)	6/6(6.00)	8/7(7.25)
*w*maha	CVLS	2	3	1	7	7/2(4.80)	10/5(6.38)	14/5(7.75)
	CVLS	2	3	3	5	5/2(3.75)	6/5(5.38)	6/5(5.38)
	CVLS	1	1	1	10	10/3(6.77)	13/6(8.25)	16/6(9.25)
25maha	CVLS	5	5	1	32	**32**(31.96)	102(66.25)	153(83.00)
	CVLS	2	3	1	65	**65**(63.95)	107(71.75)	157(87.12)
	CVLS	1	1	1	125	**125**(94.88)	156(88.5)	180(96.38)

add/sub: No. of adder/subtractor; Ch: No. of chaining;
St: No. of state;

6 Conclusions

This paper proposed a novel scheduling method based on condition vectors which can schedule operations independent of the control dependencies, such as node dividing, pre-execution of branch operation and parallelization of multiple conditional trees. Also, presented is a FSM synthesis algorithm for the parallelized multiple conditional trees. The experimental results show that our method can produce remarkably better scheduling and it runs efficiently.

References

[1] A.C.Parker, et al, "MAHA:A Program for Datapath Synthesis", 23rd DAC, pp416-424, 1986.

[2] P.G.Paulin, J.P.Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis", 24th DAC, pp195-202, 1987.

[3] N. Park and A.C.Parker, "SEHWA:A Software Package for Synthesis of Pipelines from Behavioral Specifications", IEEE trans. on CAD, vol.7, no.3, March 1988.

[4] C.J.Tseng, et al, "Bridge:A Versatile Behavioral Synthesis System", Proc. 25th DAC, pp415-420, 1988.

[5] T.Kim, et al., "A Scheduling Algorithm For Conditional Resource Sharing", ICCAD'91, pp84-87. 1991.

[6] R.Camposano, "Path-Based Scheduling for Synthesis",IEEE trans. on CAD, Vol.10,No.1, pp85-93,1991.

[7] R. Potasman, et al., "Percolation Based Synthesis", 27th DAC, pp444-449, 1990.

[8] P.F. Yeung and D.J.Rees, "Resources Restricted Global Scheduling", VLSI'91, 7.2, 1991.

[9] K.Wakabayashi and T.Yoshimura, "A Resource sharing and Control Synthesis Method for Conditional Branches", ICCAD'89,62-65,Nov 1989

[10] K.Wakabayashi, "Cyber: High Level Synthesis System from Software into ASIC", High-level VLSI Synthesis, Kulwer Academic Publishers, June 1991.