PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 48, NO. 3–4, PP. 133–149 (2004)

TIME-CONSTRAINED DESIGN OF PIPELINED CONTROL-INTENSIVE SYSTEMS

András ORBÁN, Zoltán Ádám MANN and Péter ARATÓ

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
H–1117 Budapest, Magyar tudósok körútja 2, Hungary
e-mail: arato@iit.bme.hu, {zoltan.mann, andras.orban}@cs.bme.hu

Received: April 2, 2004

Abstract

Although there are widely known solutions for dataflow-dominated resource constrained high-level synthesis (HLS) problems, optimization of hardware resources under time-constraints in control-intensive systems is still a challenge. This paper examines the case when functional pipelining is used to increase the throughput of the system. The traditional concept of mutually exclusive conditional branches must be dropped and new methods are needed to exploit the resource sharing possibilities of conditional branches. We developed new methodologies able to exploit the resource sharing possibilities under these circumstances and extended the two schedulers and the allocation module of the HLS tool PIPE to handle arbitrarily nested conditional structures and demonstrated the improved resource utilization on control-intensive benchmarks.

Keywords: conditional branches, functional pipelining, time-constrained scheduling, allocation, HLS.

1. Introduction

As hardware design becomes even more complex and time-to-market constraints grow, there is a strong motivation for high-level-synthesis methodologies. HLS should be able to equally handle dataflow-dominated and control-intensive designs. There are efficient algorithms and tools (also covering a wide-range of special cases like multicycle operations, operation chaining, functional pipelining etc.) in [2, 9, 5, 16], but there is still a need for new solutions in the latter area.

Mostly the resource-constrained design problem is considered. Although less frequently analysed, the time-constrained model is very useful in the early stage of the system design where behavioral timing requirements are already known but few information of the resources are available. The HLS tool PIPE [2] developed at our department is one of the few tools working in *time-constrained* model.

The ever-growing speed requirements to special purpose hardware circuits increased the significance of pipeline architectures. Pipeline operation is useful if a large amount of data has to be processed with the same algorithm and inter-iteration dependencies are not present. Examples include image processing, cryptographic applications, medical devices. This paper deals with the control-intensive HLS problem in the case of functional pipelining.

Related articles in the field of control-intensive HLS can be categorized as exact methods [6, 23] and heuristics [24, 3, 18]. There are several optimization techniques mainly in the scheduling phase of HLS. A good survey can be found in [15]. We mention some of them without the intention of an exhaustive list: code-motion techniques moving operations in and out of conditionals [11], speculative execution [10, 24], conditional resource sharing detection [14, 23, 25], node duplication [24], lazy execution transformation [7], false-path elimination [15], optimization according to the probabilistic distribution of the input values [4]. Another research direction aims at optimizing the underlying control structure or, vice versa, the control structure implies further constraints on the scheduling [7, 13].

Another approach beside HLS dealing with parallelization techniques is represented by the ILP (Instruction Level Parallelism) community. In most cases a concrete target architecture is selected (e.g. superscalar or VLIW processors), hence the majority of papers deal again with the resource-constrained scheduling problem. Similar techniques have been developed in this field as well, like code–motion [8], loop–pipelining [20, 12] or branch probability prediction [26]. These techniques aim at executing many operations as soon as possible (respecting the resource constraints) to reduce the latency of the system. In time–constrained model, on the other hand, one should rather distribute the operations uniformly in every control step to get minimal resource usage.

There are several representations of the control-intensive design problem, the most commonly used are the variants of the CDFG (Control and Data Flow Graph) [17, 4], however, new guard-based representations are emerging [15, 23] claiming that this representation can avoid the drawbacks of syntactic variance in the input description.

The basic concept of resource sharing in a control-dominated problem instance is based on the mutual exclusiveness of the two branches of a conditional [6, 25]. However, if the system is used in pipeline mode to achieve higher throughput, this principle is hurt, hence traditional approaches fail. It is possible that with the first piece of data the left branch of a conditional, while with the second piece of data the right branch of the same conditional is active *at the same time*, hence they are not mutual exclusive. There are only few papers handling pipeline processing in control-intensive problems. [23] is one of the exceptions; it presents a theoretical calculation of mutual exclusivity in loop-intensive problems using a complicated guard mechanism for the detection. However, it also examines the resource-constrained problem. Our model, on the contrary, is time-constrained and is much simpler and it is not only theoretical but a tool that has proven its applicability.

PIPE automates the HLS process from C to VHDL. It operates on a CDFG-like data structure detailed in Section 2. In the design of PIPE the support of pipeline processing from the beginning of the design was one of the most important goals. Now we extend it to manage conditional branches in the same philosophy. The former version handles conditional branches in the most pessimistic way: it produces a circuit that would be able to execute both branches at any time and no resource-sharing is applied.

Although the two branches are not *completely* mutually exclusive, but depending on the value of the restart time (i.e. the frequency of giving new input to the system) there are *pairs* of operations that are mutual exclusive and can be allocated to the same processor. (An example follows in Section 2.) These pairs can be detected without assuming anything about the branch probabilities. Beside the detection mechanism, our main contribution is to improve the two schedulers and the allocation module of PIPE to take advantage of these resource-sharing possibilities.

Note that the aim of this paper is neither to compete with advanced code-motion techniques available in the literature nor to use a special representation. We only focus on the issues following from the pipeline processing; the above techniques are orthogonal to that and might be combined with ours.

The paper is organized as follows. Section 2 introduces the basic concepts of our model, the structure of the HLS tool PIPE. Section 3 describes the new scheduling and allocation algorithms and the extension of PIPE in detail. The experimental results comparing the former PIPE with the updated one follow in Section 4, while Section 5 concludes the paper.

2. Background

To represent conditional information we use a CDFG variant. Besides the usual nodes representing data-transformation operations (elementary operations, EO) and usual edges representing data-dependencies some special control nodes and control edges are introduced to identify the nested structure of conditional branches. The control nodes have no real operation on data. The beginning of a conditional branch is marked with a special *fork* node. For the sake of simplicity of our algorithms we divided this node into two, namely *forkR* and *forkL* identifying the right and the left branch of the conditional, respectively. As input they receive the output of the evaluated condition corresponding to this conditional and they decide whether to be active or not. The fork nodes are connected to the first operations of their conditional branch through special control edges. An active fork sends a start signal to its successors through these control edges. The fork operation has zero execution time, and it will not be allocated into physical processors.

The end of the conditional branch is represented by a *join* node. If e.g. variable x can be modified in a conditional branch, the value of x will depend on that conditional. The task of the join operation is to select the appropriate value depending on the branch that has been really executed. If there are k variables that can be modified in either of the branches of the conditional, the join operation will have $2k + 1$ input signals: the data signals for each variable (one for the left branch and another for the right branch) and the control information of the executed branch through a control edge. *Fig. 1* shows an example.

In the time-constrained model the goal of the design is to minimize the total cost of utilized resources. Beside the CDFG two other parameters are given as input describing the time requirements of the system. The latency of the system, denoted by L, is the time the system requires to execute an iteration. The more important time property is the restart time (or initiation interval) of the system, denoted by R, which specifies how frequently a new input is fed into the system. If $R < L$, we talk about *functional pipelining*, i.e. the system is working on more than one input data at the same time. There is a minimal valid latency determined by the sum of the execution times on the longest path of the CDFG. The actual throughput of the system is determined by R, hence a decrease in R might be compensated by a small increase in L.

PIPE automates the whole HLS process starting from the C until structural VHDL code. Separate modules are responsible for the different tasks of HLS, therefore it is easy to improve a subtask by replacing the corresponding module. The most important modules of PIPE are the following:

Restart. This module makes transformations on the CDFG to make the desired restart time possible. It inserts buffers and replicates operations in an optimal way. See [2] for more details.

Scheduling. This module aims at finding the starting time of each operation. The quality of the final output is highly dependent on the used scheduling algorithm. Two different scheduler modules can be plugged into PIPE, the first one is a force-directed scheduler [21, 22] the second is a genetic scheduler [1]. Both schedulers have been extended to handle conditional branches, see Section 3 for details.

Allocation.[1] The scheduled operations are allocated to physical processors. There is a mapping between operations and available processor types. An unlimited number of processors are assumed in each type, and the goal is to use a minimal-cost set of them. The allocation unit uses a first-fit heuristic. This module has also been extended to take advantage of the conditional branches (Section 3).

Let us consider the example in *Fig. 1* which demonstrates the used notions and the problem of conditional branches in pipeline mode. The name of the operations contains their type and a number, e.g. `-2` means the second subtraction. Assuming that every real operation except multiplication takes one time step, and multiplication takes three[2], the minimal latency of the system equals $L = 7$ time steps. For the sake of simplicity we use ALUs capable of executing all the operations in the example.

Without pipeline operation ($R = 7$) the operations `*1` and `+1` are compatible, i.e. they can share the same resource—irrespective of the scheduling, since they are in separate conditional branches. Now consider the pipeline case with $R = 4$ and $L = 8$. A possible scheduling with allocation can be seen in *Fig. 2*. A vertical track corresponds to an ALU. The operations `-1`,`-2` and `+1` conditionally

[1]Note that scheduling and allocation are separate subsequent tasks in our model. For further details see [2].

[2]Multicycle operations are allowed in our model.

```
int a,b,c,d;

if (a<1){
  b=c*d;
  if(a+b>0){
    c=a-3;
  }
  else{
    c=b-3;
  }
}
else{
  b=a+d;
}
```

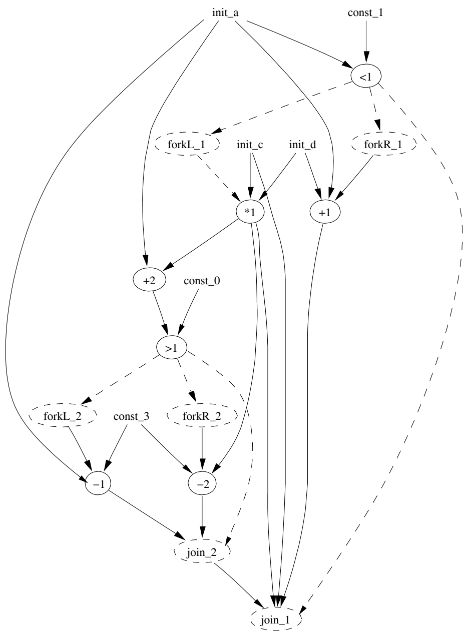


Fig. 1. An example C code and the generated CDFG

share the same ALU. The figure shows a second iteration shadowed. Although operation `+1` and `*1` are in different branches of the same conditional, they are concurrent in time-cycle 7 due to pipelining. (Of course they are still mutually exclusive within an iteration.)

3. Improved Algorithms

This section presents our new algorithmic results. First an effective method is described to extract the conditional structure from the input description, then the so-called CONCHECK algorithm is introduced, which can decide whether two operations can share the same processor or not. Finally two existing scheduling algorithms and the allocation procedure will be modified according to the new concepts.

3.1. Extracting Conditional Information

In order to determine the compatibility of node-pairs with the CONCHECK algorithm (Section 3.2), first the information whether two nodes are executed under exclusive conditions, i.e. whether they are mutually exclusive in the traditional

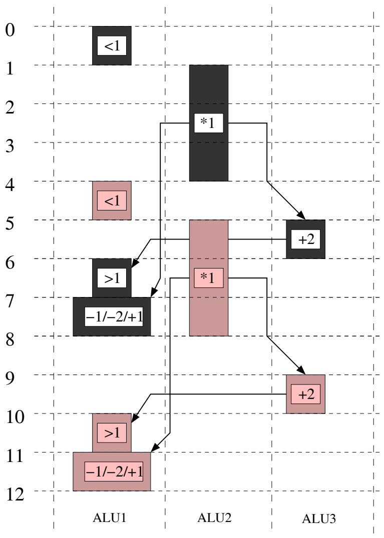


Fig. 2. A possible scheduling for the example with $R = 4$ and $L = 8$.

sense, has to be extracted. Formally:

DEFINITION 1 *Let $\bowtie \subseteq V \times V$ be the following symmetric relation between two operations. $EO_i \bowtie EO_j$ if and only if they are in separate branches of the same conditional, i.e. they are mutually exclusive within an iteration.*

Therefore our aim is to determine the $\bowtie$ relation. Similar techniques exist based on conditional vectors [24] or BDDs [15]. In our method the detection is achieved by parsing the description file of the CDFG. We are aware of the fact that structural variance in the input file can cause a different conditional structure of the same behavior and that there are more advanced representations like HCDG [15] to avoid this, but this is not the focus of our research.

From a CDFG with n nodes and m conditional branches we create an $n \times m$ table (A) each row of which corresponds to the role of a node in the conditional branches. The role types are as follows: NOT_MEMBER, LEFT_BRANCH, RIGHT_BRANCH, FORK, JOIN. More precisely: the element of the jth column of the ith row (a_{ij}) determines the role of the ith operation (EO_i) in the jth conditional. (In the following we assume that the conditionals are somehow numbered from 1 to m and there is a one-to-one mapping between the numbers and the corresponding *fork* and *join* nodes. So we can refer to the *forkR_j*, *forkL_j* or *join_j* node of the jth conditional branch.) This table encapsulates the whole nested conditional structure and the $\bowtie$ relation as the following obvious proposition shows.

PROPOSITION 1 *Two nodes EO_i and EO_j are exactly in the same conditional* *branches if and only if their rows are exactly the same.* $EO_i \bowtie EO_j$ *if and only if there is at least one column where their roles are LEFT_BRANCH and RIGHT_BRANCH, or vice versa.*

Table 1. The table encapsulating the conditional structure of the example

Operation	**Conditional1**	**Conditional2**
`<1`	NOT_MEMBER	NOT_MEMBER
forkR_1	FORK	NOT_MEMBER
`+1`	RIGHT_BRANCH	NOT_MEMBER
forkL_1	FORK	NOT_MEMBER
`*1`	LEFT_BRANCH	NOT_MEMBER
`+2`	LEFT_BRANCH	NOT_MEMBER
`>1`	LEFT_BRANCH	NOT_MEMBER
forkR_2	LEFT_BRANCH	FORK
`-2`	LEFT_BRANCH	RIGHT_BRANCH
forkL_2	LEFT_BRANCH	FORK
`-1`	LEFT_BRANCH	LEFT_BRANCH
join_2	LEFT_BRANCH	JOIN
join_1	JOIN	NOT_MEMBER

Table 1 contains the conditional information of the previous example. However, it is not straightforward to fill in this table, since the information in each row may depend on other ones.

Important to note that the row of an operation can only depend on the rows of its *predecessors*, but not on the successors' rows. Therefore one should first sort the nodes of the CDFG in topological order, i.e. $EO_1 \prec EO_2 \prec \ldots \prec EO_n$, so that for each edge (EO_i, EO_j) $i < j$ holds[3]. Parsing the nodes in this order all the information needed to fill in the row of the next node is already available. Based on the following proposition an effective algorithm can be given for the extraction of conditional information.

PROPOSITION 2 *The ordinary node (neither fork nor join) x is member of the j-th conditional branch if and only if one of its predecessors is member (RIGHT_ BRANCH, LEFT_BRANCH or FORK) of the j-th conditional branch.*

REMARK 1 *It is possible for node x to have predecessors that are members of the jth conditional as well as predecessors that are not. The previous proposition states that in this case x is a member of the j-th conditional. See e.g. the* `-2` *node of*

[3]More precisely we should write that there is a permutation π of the set $\{1,2,\ldots,n\}$ so that $EO_{\pi(1)}, EO_{\pi(2)}, \ldots, EO_{\pi(n)}$ is a topological order. For the sake of simplicity we assume that the original numbering is a topological order.

the example of Fig. 1: this node has three predecessors, one of which (`forkR_2`) is member of the second conditional branch, others are not. Node `-2` should be declared as a member of the second conditional.

Proof of the proposition. Since x needs the result of an operation in the jth conditional branch, this result will only be produced provided the condition for this conditional branch is evaluated as true. Hence the execution of x also depends on the condition of the jth branch, so x is a member of this conditional branch. The other direction of the proof is obvious. □

This proposition implies an algorithm to fill in the table. The input nodes are not members in any of the conditional branches. The following recursive definition of a_{ij} can be obtained.

$$a_{ij} := \begin{cases} \text{LEFT\_MEMBER}, & \text{if } \exists k : (EO_k, EO_i) \in E \text{ and} \\ & (a_{k,j} = \text{LEFT\_MEMBER or } a_{k,j} = forkL\_j) \\ \text{RIGHT\_MEMBER}, & \text{if } \exists k : (EO_k, EO_i) \in E \text{ and} \\ & (a_{k,j} = \text{RIGHT\_MEMBER or } a_{k,j} = forkR\_j) \\ \text{JOIN}, & \text{if } EO_i = join\_j \\ \text{FORK}, & \text{if } EO_i = forkL\_j \text{ or } EO_i = forkR\_j \\ \text{NOT\_MEMBER}, & \text{otherwise.} \end{cases} \tag{1}$$

Parsing the input graph in topological order, table A can be easily filled in using the above rule.

The complexity of the algorithm is linear in the size of the CDFG, i.e. $\mathcal{O}(n + |E|)$, since sorting the nodes in topological order of a DAG takes linear time and determining the value of a_{ij} according to *Eq.* (1) takes in-degree of EO_i time, that is altogether also linear time,[4].

The hierarchy of the conditional branches can also be detected easily using this table.

3.2. *The CONCHECK Algorithm*

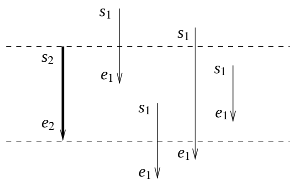
To exploit the resource sharing possibilities of conditional branches even in pipeline processing an algorithm should be given that decides whether two scheduled operations are compatible or concurrent, i.e. they can be allocated to the same processor or not. (We examine only operation pairs mapped to the same processor type.) This algorithm is called CONCHECK. This is the improved and simplified version of the CONCHECK algorithm presented in [2]. It is important that CONCHECK works without assuming anything about the branch probabilities. In the following we outline the basic concepts of CONCHECK.

[4]A CDFG generally has only few edges, $|E| = \mathcal{O}(n)$, hence $\mathcal{O}(n + |E|) = \mathcal{O}(n)$.

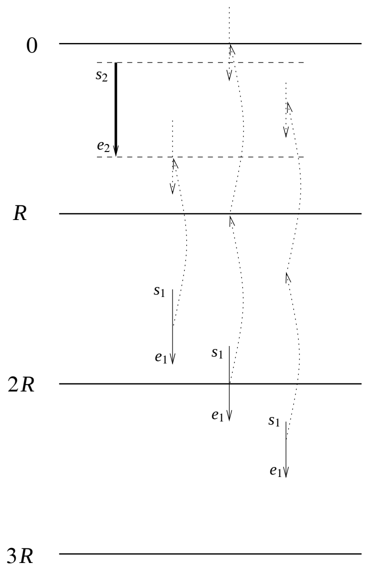
3.2.1. *Without Conditional Branches*

Assume two operations EO_1 and EO_2 that occupy their processing units in the closed intervals $[s_1, e_1]$, $[s_2, e_2]$, respectively. If there is no pipelining ($R = L$), the two operations are concurrent if and only if the intervals $[s_1, e_1]$ and $[s_2, e_2]$ intersect. The possible arrangement of the intervals can be seen in *Fig. 3(a)*. It is easy to see that the necessary and sufficient condition for the intersection is the following:

$$s_1 < e_2 \text{ and } s_2 < e_1 \tag{2}$$



(a) The intersection of two intervals within the same iteration



(b) The intersection of two intervals between iterations

Fig. 3. Some possible positions of intersecting intervals (left) and intersecting intervals modulo R (right)

If pipeline processing is allowed, these intervals should intersect modulo R. This means that e.g. interval $[s_1, e_1]$ can be shifted by a multiple of R so that the two intervals intersect. *Fig. 3(b)* illustrates such situations. This can be expressed as:

$$\exists k \in \mathbb{Z} : \quad s_1 - k \cdot R < e_2 \text{ and } s_2 < e_1 - k \cdot R \tag{3}$$

The previous equation simplifies as (since R is a positive integer):

$$\exists k \in \mathbb{Z} : \quad \frac{e_1 - s_2}{R} > k > \frac{s_1 - e_2}{R} \tag{4}$$

Our goal is to eliminate k. *Eq.* (4) holds if and only if the largest integer *strictly* smaller than $\frac{e_1-s_2}{R}$ still greater than $\frac{s_1-e_2}{R}$ is. The use of the $\lfloor\cdot\rfloor$ operation almost solves the problem with one exception: if $\frac{e_1-s_2}{R}$ is by chance an integer, $\lfloor\frac{e_1-s_2}{R}\rfloor$ returns the *same* number and not the largest integer strictly *smaller* than that. The desired behaviour can be achieved with the use of the floor operation and a small trick: we subtract a sufficiently small number, $\frac{1}{R+1}$ from $\frac{e_1-s_2}{R}$. If $\frac{e_1-s_2}{R}$ is non-integer, it will have the same integer part after subtraction as before. If it is integer, the subtraction will decrease the integer part by one. To sum up, the operations are concurrent if and only if the following equation holds:

$$\left\lfloor \frac{e_1 - s_2}{R} - \frac{1}{R+1} \right\rfloor > \frac{s_1 - e_2}{R} \tag{5}$$

It is possible that more than one k values satisfy (3) (e.g. if one of the operations lasts at least R, then surely), but this will not cause any problem, this case need not to be handled specially[5]; for us it is only relevant that at least one such k exists.

3.2.2. *With Conditional Branches*

The situation is slightly different if the two operations are in separate branches of the same conditional, i.e. if $EO_1 \bowtie EO_2$. If there is no pipeline processing, then the two operations are always compatible; this is the traditional mutual exclusivity principle. However, with pipeline processing they are concurrent if the two intervals intersect modulo R but they do not intersect within the same iteration. This is because the two operations cannot be executed at the same time in the same iteration; the only way to operate simultaneously is between iterations, hence k, the difference of the iterations of EO_1 and EO_2, cannot be zero. To be precise, the previous condition of *Eq.* (4) is applicable with a small exception:

$$\exists k \in \mathbb{Z}, k \neq 0 : \quad \frac{e_1 - s_2}{R} > k > \frac{s_1 - e_2}{R} \tag{6}$$

[5]If e.g. $e_1 \geq s_1 + R$ then the two operations are surely concurrent. It can be derived from (4) as well: $\frac{e_1-s_2}{R} \geq \frac{s_1-s_2}{R} + 1 > \left\lceil\frac{s_1-s_2}{R}\right\rceil \geq \frac{s_1-s_2}{R} > \frac{s_1-e_2}{R}$, hence $k := \left\lceil\frac{s_1-s_2}{R}\right\rceil$ proves that they are indeed concurrent.

We eliminate k the same way as before, but the case when $k = 0$ is the only solution that should be forbidden.

$$\left\lfloor \frac{e_1 - s_2}{R} - \frac{1}{R+1} \right\rfloor > \frac{s_1 - e_2}{R} \tag{7}$$

and

$$\left\lfloor \frac{e_1 - s_2}{R} - \frac{1}{R+1} \right\rfloor = 0 \Rightarrow \frac{s_1 - e_2}{R} < -1 \tag{8}$$

Eq. (8) ensures that if $k = 0$, then *Eq.* (6) does not have a solution.

The flowchart of the new CONCHECK algorithm incorporating both the conditional and the non-conditional cases is illustrated in *Fig.* 4.

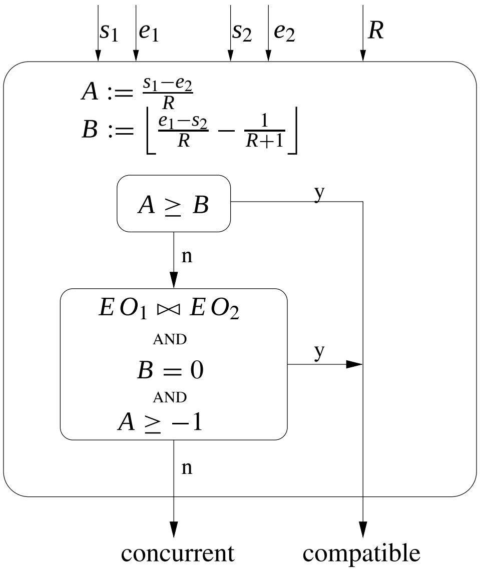

s_1
e_1
s_2
e_2
R
$A := \frac{s_1-e_2}{R}$
$B := \left\lfloor \frac{e_1-s_2}{R} - \frac{1}{R+1} \right\rfloor$
$A \geq B$
y
n
$EO_1 \bowtie EO_2$ AND $B = 0$ AND $A \geq -1$
y
n
concurrent
compatible

Fig. 4. The CONCHECK algorithm

3.2.3. *Multiplied Operations*

To achieve the desired throughput given by R, some operations need to be multiplied in several copies (see the restart algorithm in Section 2 and in [2]). CONCHECK is able to handle multiple copies of operations as well. The original CONCHECK algorithm given in [2] also handles multiplied operations, but needs additional examination to achieve this. Our CONCHECK algorithm, on the other hand, already incorporates the case of multiple operations and needs no further questions to decide the compatibility in this special case.

Table 2. The benchmark problems

Name (size)	Conditions		Description
	Nr.	Depth	
example1 (23)	7	2	Simple benchmark v1.
jian (28)	4	2	The benchmark of [15]
example2 (29)	9	2	Simple benchmark v2.
quad (36)	3	2	Quadratic equation solver
concheck1 (38)	6	2	The CONCHECK alg. v1.
concheck2 (51)	8	4	The CONCHECK alg. v2.
det (80)	4	2	Calculates a determinant
idea (230)	2	1	IDEA cryptographic alg.

3.3. New Scheduling Algorithms

PIPE contains two scheduling heuristics: a genetic scheduler and a force-directed scheduler. The user can decide which scheduler to use.

The aim of scheduling is to find a good candidate for the allocation, that is to select a schedule for which allocation finds a low-resource solution. Since allocation itself is an $\mathcal{NP}$-hard problem [19], it is too expensive to evaluate each schedule with that, hence another objective function for scheduling has to be chosen.

The genetic scheduler tries to find a schedule with a maximum number of compatible pairs.

To extend the genetic scheduler to effectively handle conditional branches is simply to calculate the number of compatible node pairs according to the CONCHECK algorithm. (Previously the information of conditional branches was completely ignored.)

The force-directed scheduler aims at finding a schedule for which the resource load at each time step is the most uniform possible. The resource load in a time step is summed up by the resource needs of each operation. This is calculated in a probabilistic way: each operation occupies every time step of its mobility domain with the same probability and the resource usage is this probability multiplied with the full resource-need of this operation.

Due to conditional branches this calculation should be modified in such a way, that instead of the *sum* of the resource-need of compatible node pairs scheduled to the same time the *maximum* should be considered. The resource-need of an operation can be further on evaluated in a probabilistic way. To build this maximum properly, the tree representing the conditional hierarchy of the CDFG should be traversed with a depth-first search.

3.4. Allocation

The allocation module uses a first-fit allocation heuristics. The nodes are considered in some heuristic order and each node is allocated into the first free processor (note that the starting time of the node is already known) able to execute this operation. If none of the processors is free, a new processor instance is created.

Note that without conditional branches two operations scheduled to the same time must always use two separate processors. Compatible operations in separate branches of the same conditional that are scheduled to the same time are allowed to share the same processor. The calculation of free processors has been modified accordingly. As a result, allocation employs this information to find a lower resource utilization.

4. Experimental Results

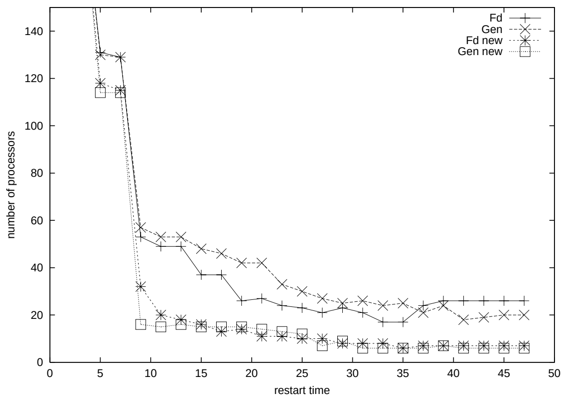


Fig. 5. The detailed results of the 'det 48,[3-48]' example

Since we could not find any paper reporting test results on exactly the same problem as ours (time-constrained model, control-intensive behavior, functional pipelining), we compared our improved algorithms to the previous version of PIPE. Altogether four versions of PIPE were tested, the previous versions are called *Fd* and *Gen*, corresponding to the type of the scheduler (force-directed or genetic) and the improved versions are called *Fd-new* and *Gen-new*, respectively.

The list of benchmark problems can be found in *Table* 2. Note that the last benchmark is significantly larger than the usual benchmarks in the literature, what justifies our methods on big examples. Although it has only one conditional, this conditional reaches across the whole problem, hence the resource-sharing becomes vital in this example. On each benchmark we ran a complete scan of tests: we fixed

Table 3. The reduction with the new algorithms in the number of required ALUs

Name L,[R range]	Reduction Fd new vs. Fd	Reduction Gen new vs. Gen
example1 18,[3-18]	32.9%	38.6%
jian 24,[3-24]	9.3%	15.6%
example2 18,[3-18]	30.9%	37.5%
quad 72,[3-30]	20.6%	29.6%
quad 72,[30-70]	33.7%	35.6%
concheck1 33,[3-33]	17.7%	13.4%
concheck2 61,[3-30]	9.4%	17.6%
concheck2 61,[30-61]	15.7%	24.5%
det 48,[3-48]	54.5%	61.3%
idea 300,[11-30]	19.8%	35.3%
idea 300,[30-100]	28.6%	34.0%
idea 300,[100-300]	34.9%	40.9%

the latency of the benchmark to a concrete value and varied the restart time between 3 clock cycles and the latency. We refer to a test case in the form: 'test_name L_value,[R range]', for example 'det 48,[3–48]' means, that the 'det' benchmark has latency 48 and the restart time varied between 3 and 48. If this range was very large, we divided the range into several pieces, since we expected different behavior with smaller and with larger R values. To make the evaluation easier we assumed that only one resource type, ALU was available, that is able to execute all the operations. In all benchmarks the comparator operations (<,>,==, etc.) require 2 clock cycles, the addition and subtraction 4 clock cycles and multiplication and division 8 clock cycles. The tests ran on 7 different PCs equipped with SuSE Linux and 500MHz-1GHz processors.

The detailed results for a selected benchmark can be seen in *Fig.* 5. The four functions indicate the required number of processing units of each algorithm for various restart time values. One can clearly see, that the improved algorithms perform significantly better than the previous ones, especially at higher R values.

It would be infeasible to show all the details for all test cases, hence a summary is given in *Table* 3. For each benchmark the average reduction to the previous version of the algorithm in the number of required ALUs is presented. The average is taken over the different restart time values of the given range. As we expected the new algorithms could reduce significantly the required number of processors. If R is not much lower than L (see e.g. the *idea* or the *quad* benchmark), the new algorithms work especially effectively. The reason is that if R is very low, it is nearly impossible to share a processing unit regardless of the conditional branches, but if R is high enough, our resource-sharing mechanism starts working.

The average execution times of the algorithms can be studied in *Table* 4.

Table 4. The average execution times of the different algorithms in each benchmark

Name L,[R range]	Fd	Gen	Fd new	Gen new
example4 18,[3-18]	2.8 s	5.8 s	13.4 s	12.6 s
jian 24 [3-24]	2.5 s	5.6 s	11.7 s	13.0 s
example5 18,[3-18]	5.8 s	14.8 s	24.5 s	25.5 s
quad 72,[3-30]	55.1 s	73.9 s	43.2 s	48.8 s
quad 72,[30-70]	83.1 s	140.8 s	42.4 s	51.0 s
concheck1 33,[3-33]	12.2 s	27.6 s	48.6 s	50.8 s
concheck2 61,[3-30]	35.4 s	62.6 s	43.4 s	48.1 s
concheck2 61,[30-61]	43.6 s	171.0 s	40.8 s	45.1 s
det 48,[3-48]	41.0 s	89.2 s	283.6 s	249.6 s
idea 300,[11-30]	45.5 m	31.1 m	21.4 m	26.7 m
idea 300,[30-100]	26.9 m	56.9 m	44.8 m	57.1 m
idea 300,[100-300]	41.0 m	36.3 m	38.0 m	48.2 m

One can see that the new algorithms usually generate some overhead according to the more complex calculation but it is not very significant since every algorithm finishes within a couple of minutes in the small tests and within an hour in the large *idea* benchmark.

Unfortunately the execution times are very rhapsodic in every version of the algorithm, which is worth some explanation: The execution time of the force-directed scheduler heavily depends both on the number of operations and the size of the mobility domain of the operations. The mobility of an operation is continuously changing during scheduling, which also depends on the algorithm itself. The old and the new force-directed scheduling algorithm might result in different mobility domains, which can cause both shorter and longer execution times. The running time of the genetic scheduler is determined by its termination condition: on the one hand, the number of iterations is bound by a minimum and a maximum value and, on the other hand, the algorithm also stops if the population has not been improved in the last 30% of the iterations. This important second termination condition is responsible for the fluctuation of the running times.

5. Conclusion

This paper deals with control-intensive, pipelined high-level synthesis problems with time constraints. It gives a simple algorithm for the detection of compatibility of operations, which can be regarded as the generalization of mutual exclusivity of conditional branches in non-pipeline processing. The two schedulers and the allocation module in the HLS tool PIPE working in the time-constrained model has been extended accordingly and a new module extracting the conditional hierarchy has been added. The experimental results demonstrated that the new methodology could reduce the required number of processing units in control-flow-dominated problem instances significantly.

Our future research includes the analysis of inter-iteration-dependent loops containing conditional branches and working in functional pipelining.

References

[1] ARATÓ, P. – MANN, Z. Á. – ORBÁN, A., Genetic Scheduling Algorithm for High-Level Synthesis, in: *Proceedings of the IEEE 6th International Conference on Intelligent Engineering Systems*, 2002.

[2] ARATÓ, P. – VISEGRÁDY, T. – JANKOVITS, I., *High-Level Synthesis of Pipelined Datapaths*, John Wiley & Sons, Chichester, United Kingdom, first edition, 2001.

[3] BERGAMASCHI, R. – RAJE, A. – NAIR, I. – TREVILLYANET, L., Control-Flow Versus Data-Flow Based Scheduling: Combining Both Approaches in an Adaptive Scheduling System, *IEEE Trans. Very Large Scale Integration Systems*, **5** No. 1 (1997), pp. 82–100.

[4] BHATTACHARYA, S. – DEY, S. – BRGLEZ, F., Performance Analysis and Optimization of Schedules for Conditional and Loop-Intensive Specifications, in: *Design Automation Conference*, (1994), pp. 491-496.

[5] CAMPOSANO, R., From Behaviour to Structure: High-Level Synthesis, *IEEE Design and Test of Computers*, **10** (1990), pp. 8–19.

[6] CAMPOSANO, R., Path-Based Scheduling for Synthesis, *IEEE Transactions on Computer-Aided Design*, **10** No. 1 (1991), pp. 85–93.

[7] DOS SANTOS, L. – HEIJLIGERS, M. – VAN EIJK, C. – VAN EIJNHOVEN, J. – JESS, J., A Code-Motion Pruning Technique for Global Scheduling, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, **5** No. 1 (2000), 1–38.

[8] EBCIOGLU, K. – NICOLAU, A., A Global Resource-Constrained Parallelization Technique, *ACM SIGARCH International Conference on Supercomputing*, June, 1989.

[9] GAJSKI, D., *High-Level Synthesis*, Kluwer Academic Publishers, 1992.

[10] GUPTA, S. – SAVOIU, N. – DUTT, N. D. – GUPTA, R. K. – NICOLAU, A., Conditional Speculation and its Effects on Performance and Area for High-Level Synthesis, in: *International Symposium on System Synthesis*, October, 2001.

[11] GUPTA, S. – SAVOIU, N. – KIM, S. – DUTT, N. D. – GUPTA, R. K. – NICOLAU, A., Speculation Techniques for High Level synthesis of Control Intensive Designs, in: *Design Automation Conference*, June, 2001.

[12] HOLTMANN, U. – ERNST, R., Combining MBP-Speculative Computation and Loop Pipelining in High-Level Synthesis, in: *European Design and Test Conference*, 1995, pp. 550–556.

[13] KIFLI, A. – GOOSSENS, G. – DE MAN, H., A Unified Scheduling Model for High-Level Synthesis and Code Generation, in: *European Design and Test Conference*, 1995, pp 234–238.

[14] YONEZAWA, K. N. – LIU, J. W. S. – LIU, C. L., A Scheduling Algorithm for Conditional Resource Sharing – a Hierarchical Reduction Approach, *IEEE Transactions on CAD*, April, 1994.

[15] KOUNTOURIS, A. A. – WOLINSKI, C., Efficient Scheduling of Conditional Behaviors for High-Level Synthesis, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, **7** No. 3 (2002), pp. 380–412

[16] KUCHCINSKI, K., An Approach to High-Level Synthesis Using Constraint Logic Programming, in: *Proceedings of the 24th Euromicro Conference, Workshop on Digital System Design*, 1998.

[17] LAKSHMINARAYANA, G. – JHA, N. K., FACT: A Framework for the Application of Throughput and Power Optimizing Transformations to Control Flow Intensive Behavioral Descriptions, in: *Design Automation Conference*, 1998, pp. 102–107.

[18] LAKSHMINARAYANA, G. – RAGHUNATHAN, A. – JHA, N. K., Incorporating Speculative Execution into Scheduling of Control-Flow Intensive Behavioral Descriptions, in: *Design Automation Conference*, (1998), pp. 108–113".

[19] MANN, Z. Á. – ORBÁN, A., Optimization Problems in System-Level synthesis, in: *Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, 2003.

[20] MOON, S.-M. – EBCIOGLU, K., An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors, in: *25th Annual International Symposium on Microarchitecture*, 1992.

[21] PAULIN, P. G. – KNIGHT, J. P., Force-Directed Scheduling for the Behavioural Synthesis of ASICs, *IEEE Transations on Computer Aided Design*, 1989.

[22] PRABHAKARAN, P. – BANERJEE, P., Parallel Algorithms for Force Directed Scheduling of Flattened and Hierarchical Signal Flow Graphs, *IEEE Transactions on Computers*, **48** No. 7 (1999), pp. 762–768.

[23] RADIVOJEVIC, I. – BREWER, F., Analysis of Conditional Resource Sharing using a Guard-based Control Representation, in: *International Conference on Computer Design – ICCD'95*, (1995), pp. 434–439.

[24] WAKABAYASHI, K. – TANAKA, H., Global Scheduling Independent of Control Dependencies Based on Condition Vectors, in: *Design Automation Conference (DAC)*, (1992), pp. 112–115.

[25] WAKABAYASHI, K. – YOSHIMURA, T., A Resource Sharing and Control Synthesis Method for Conditional Branches, in: *International Conference on Computer-Aided Design (ICCAD)*, 1989, pp. 62–65.

[26] YU, T. – SHA, E. – PASSOS, N. – JU, R., Algorithm and Hardware Support for Branch Anticipation, in: *7th Great Lakes Symposium on VLSI*, 1997.