HAL
open science

Efficient Scheduling of Conditional Behaviors for High-Level Synthesis

Apostolos Kountouris, Christophe Wolinski

To cite this version:

Apostolos Kountouris, Christophe Wolinski. Efficient Scheduling of Conditional Behaviors for High-Level Synthesis. ACM Transactions on Design Automation of Electronic Systems, 2002, 7 (3), pp.380-412. 10.1145/567270.567272 . hal-00544901

HAL Id: hal-00544901

https://hal.science/hal-00544901v1

Submitted on 9 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficient Scheduling of Conditional Behaviors for High-Level Synthesis

APOSTOLOS A. KOUNTOURIS
Mitsubishi Electric ITE-TCL
and
CHRISTOPHE WOLINSKI
IRISA

As hardware designs get increasingly complex and time-to-market constraints get tighter there is strong motivation for high-level synthesis (HLS). HLS must efficiently handle both dataflow-dominated and controlflow-dominated designs as well as designs of a mixed nature. In the past efficient tools for the former type have been developed but so far HLS of conditional behaviors lags behind. To bridge this gap an efficient scheduling heuristic for conditional behaviors is presented. Our heuristic and the techniques it utilizes are based on a unifying design representation appropriate for both types of behavioral descriptions, enabling the proposed heuristic to exploit under the same framework several well-established techniques (chaining, multicycling) as well as conditional resource sharing and speculative execution which are essential in efficiently scheduling conditional behaviors. Preliminary experiments confirm the effectiveness of our approach and prompted the development of the CODESIS HLS tool for further experimentation.

Categories and Subject Descriptors: J.6 [**Computer-Aided Engineering**]: Computer-Aided Design (CAD); B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis, Optimization*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Design automation, high level synthesis (HLS), scheduling, conditional behavior

1. INTRODUCTION

In spite of the fact that high-level synthesis (HLS) methodologies have many advantages with respect to register-transfer-level (RTL) ones they still lack general industrial acceptance. One of the main reasons for this is a gap in the quality of the synthesized results between dataflow- and controlflow-dominated behavioral descriptions. In much of the previous work on HLS these two types of design descriptions have been treated separately. A result of such separation is that efficient techniques have been developed for the HLS of

Authors' addresses: A. A. Kountouris, Mitsubishi Electric ITE-TCL, Immeuble Germanium, 80, Av. des Buttes de Coesmes, 35700 Rennes, France; email: kountouris@tcl.ite.mee.com; C. Wolinski, IRISA, Rennes, France.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2002 ACM 1084-4309/02/0700-0380 $5.00

dataflow-dominated behavioral descriptions but HLS of controlflow-dominated descriptions still lags behind. This fact has stimulated considerable research work in recent years. Also, as indicated by the excellent survey of Lin [1997], the motivation behind the development of HLS is stronger than ever given the even tighter time-to-market constraints in combination with the ever-increasing system complexity at both the algorithmic as well as the implementation levels.

To bridge this gap in HLS efficiency between data- and control-dominated designs, in this article a twofold approach is proposed: a unifying intermediate design representation is adopted called HCDG for hierarchical conditional dependency graph, and, based on this design representation heuristics for HLS activities/tasks have been developed to efficiently handle descriptions with complex conditional structure as well. The main reason for using the HCDG is to avoid the inconveniences of traditional design representations in representing complex conditional behaviors. Furthermore, thanks to the HCDG dataflow nature, dataflow-intensive behavioral descriptions can also be efficiently handled by existing well-established techniques and heuristics.

Although the focus of the article is on the scheduling problem, it is commonly agreed that the intermediate design representation is intimately related to the quality of the scheduling results. The design representation is also important for the simplification of the engineering of HLS techniques used to efficiently carry out the HLS design activities (e.g., scheduling/optimization, datapath/controller synthesis, allocation, and binding). HLS of control-intensive designs has long been tied to CDFG-based representations. The inherent constraints of such representations have been more or less countered by the development of optimization techniques and by the adoption of extensions to the CDFG or GFG models. In many cases key ideas adopted in HLS were suggested by research in the compiler arena. We strongly believe that in the future dataflow-oriented representations with more elaborate control representations will prove indispensable for substantially improving HLS efficiency. At the same time such representations will provide a unified framework for the HLS of both control- and data-dominated systems as well as systems mixing both types of behaviors. Through the study of the scheduling problem this article aims at demonstrating that all previously developed CDFG-based techniques are applicable to a dataflow-oriented representation; in our case the HCDG.

Before going further it must be underlined that previous work by others has been instrumental in the development of our understanding of the problem as well as the involved techniques in working out efficient solutions, therefore so this work is summarized in the following section.

1.1 Previous Work

Scheduling behaviors having a complex conditional structure have been thoroughly investigated in previous research work. This is mainly because traditional DFG-based scheduling heuristics are not adequate for efficiently handling such descriptions as explained in Bergamaschi et al. [1997]. Approaches better adapted to this kind of problems have been proposed.

These include both heuristics (e.g., Bergamaschi et al. [1997], Kim et al. [1991], Wakabayashi and Yoshimura [1992]; Tseng et al. [1988], and Lakshminarayama et al. [1999]) as well as exact methods (e.g., Redivojevic and Brewer [1995, 1996] and Camposano [1991][1]). In these approaches several scheduling optimization techniques were developed. The quality of high-level synthesis results depends heavily on the ability of the employed scheduling algorithms to exploit conditional resource sharing [Kim at al. 1991; Wakabayaschi and Yoshimura 1989; Radiovojvic and Brewer 1995; Camposano 1991; Huang et al. 1993] and speculative execution (also called preexecution) possibilities [Wakabayaschi and Yoshimura 1992; Radivojevic and Brewer 1996; Lakshminarayama et al. 2000; Huang et al. 1993]. Node duplication has also been proven useful [Wakabayaschi and Yoshimura 1992] in optimizing scheduling results. In resource-constrained scheduling these techniques permit better utilization of the hardware resources in the datapath and obtaining better schedules (lower number of control steps). This results in shorter execution paths and less control logic. In time-constrained scheduling such techniques also help obtain shorter schedules using less hardware resources. Furthermore, exploiting speculative execution results in optimized controller synthesis Dos Santos et al. 2000, Kifli et al. 1995.

At this point it is interesting to underline the commonalities in the research areas of conditional behavior scheduling for HLS and compilation for instruction-level-parallel (ILP) architectures. The resource-constrained scheduling problem in HLS is essentially the same as scheduling for VLIW or other ILP architectures. Work in Dos Santos et al. [2000] generalizes the concept of code motions to counter the inherent limitations of CDFG and basic block-based representations in exploiting parallelism across basic block boundaries and thus ameliorate scheduling results. For generalized code motion techniques to control the compensation code, introduced to preserve program semantics, were developed in Dos Santos [1997]. Code motion may result in speculative execution and node duplication due to the introduction of compensation code. The concept of code motion was developed by research in compilation seeking to increase ILP. Its origins can be traced to percolation scheduling Nicolau 1985 and synthesis Potasman et al. 1990 as well as trace scheduling and its extensions Fisher 1981. The problem of more general code motion has also been addressed in the Trailblazing approach Novack and Nicolau 1993 for scheduling for ILP processor architectures. The effectiveness of this approach has its roots in the Hirarchical task graph (HTG) representation proposed in Girkar and Polychronopoulos [1992] addressing issues in automatic parallelism extraction from sequential descriptions. Trace scheduling also introduced the important concept of using execution profiling statistics in order to account for the conditional nature of execution. This concept has been used in Lakshminarayama et al. [1999] for HLS scheduling.

[1]Camposano [1991] refers to *path-based sheduling* (PBS); whether PBS can be considered as an exact method depends on the particular formulation of the scheduling problem. Radivojevic in his thesis as well as in published work with Brewer, explicitly qualifies PBS as a heuristic and the discussion in Bergamaschi et al. [1997 Section B.3.b] also corroborates this.

An important issue that has been either explicitly [Chaiyakul et al. 1992; Bersamaschi 1998] or implicitly [Lin 1997] underlined in previous research work concerns the effects of the syntactic variance of the input descriptions on the synthesis results. These negative effects intervene in two distinct but interrelated point: mutual exclusiveness detection (identification) and the scheduling levels. For instance, CDFG-based techniques that detect mutual exclusiveness based on the struc-ture of the input description (e.g., Kim et al. [1994] and Wakabayashi and Yoshimura [1992]) may produce quite different schedules for semantically equivalent but syntactically different input descriptions due to the variability of the amount of detected mutual exclusiveness [Chaiyakul et al. 1992; Li and Gupta 1998]. Furthermore, CFG-based scheduling techniques (e.g., PBS in Camposano [1991]) are very sensitive to the statement order in the input description. The extension in Bergamaschi et al. [1997] alleviates this problem but only within the limits of basic blocks. Several techniques have been employed, such as definition of coding style guidelines [SYNOPSYS], graph restructuring techniques before scheduling in CFG-based approaches [Bergamaschi et al. 1997; Huang et al. 1993], and finally, use of dataflow-oriented design representations and presynthesis transformations to obtain partially unique (canonical) design representations [Juan et al. 1994; Li and Gupta 1998; Kountouris and Wolinski 1999a]. The first approach of coding style guidelines becomes quite difficult to apply as designs get larger and more complex. In some sense it also defeats the purpose of high-level synthesis by necessitating that the designer produce input descriptions coping with the limitations of existing tools, implying a certain knowledge of the employed algorithms, as explained in Chaiyakul et al. [1992]. The second approach of graph restructuring can be considered as a presynthesis transformation, but in the context of CFG-based scheduling complex techniques are needed to evaluate when such restructuring is to be applied. Even though these two approaches are not of extreme elegance, they correspond to pragmatic solutions in the context of existing high-level synthesis systems [SYNOPSYS; Bergamaschi and Kuehlmann 1993]. The third approach represents a more elegant alternative with the potential to better scale with the size and complexity of the handled designs. Using dataflow-oriented representations and formal graph transformations coupled with extensive mutual exclusiveness detection techniques [Juan et al. 1994; Kountouris and Wolinski 1999a], permits coping with syntactic variance at both the mutual exclusiveness detection and scheduling levels. Finally, in recent work the Wavesched approach [Lakshminarayama et al. 1999]. focuses on efficiently scheduling descriptions containing loop structures. In a sense this approach offers a complementary set of optimization techniques (i.e., loop unrolling and handling of concurrent loops) to further ameliorate the scheduling results. With its extension in Lakshminarayama et al. [2000], speculative execution is also exploited and considerable improvement with respect to Lakshminarayama et al. [1999] is demonstrated. However the Wavesched approach suffers from the effects of syntactic variance as there is no advanced mechanism to handle conditional resource sharing and, depending on the description style, different hints are given to the scheduler.

From the above discussion it is clear that in order to efficiently schedule conditional behaviors and optimize the resulting controllers, we have to exploit conditional resource sharing [Kim et al. 1991; Wakabayashi and Yoshimura 1989; Radivojevic and Brewer 1995; Camposano 1991] and speculative execution [Wakabayshi and Yoshimura 1992; Radivojevic and Brewer 1996; Dos Santos et al. 2000; Kifli et al. 1995; Lakshminarayama et al. 2000], shorten path lengths using node duplication techniques [Wakabayshi and Yoshimura 1992], and last but not least cope with syntactic variance [Chaiyakul et al. 1992]. All these techniques may be regarded as specific types of code motions as formulated in Dos Santos et al. [2000]. Finally, once noniterative conditional behaviors can be efficiently handled, scheduling techniques must consider descriptions with iterative constructs that present further optimization opportunities as explained in Lakshminarayama et al. [1999, 2000].

This by no means exhaustive review of previous work permits the identification of all the useful techniques that need to be considered to efficiently address conditional behavior scheduling issues. Several of the shortcomings of previous work are related to the inherent constraints of the adopted design representations. In this article it is shown that using the HCDG representation alleviates some of these limitations and that it is straightforward to combine all the aforementioned techniques and enjoy their consolidated benefits.

1.2 Article Outline

In Section 2 background information on the HCDG representation is given. In particular, in Section 2.1 the HCDG is compared to other similar representations. A small benchmark example used in the rest of the article for illustration is given. Details are also given on the graph transformations and the HCDG mutual exclusiveness identification and representation capabilities. These elements are essential for obtaining optimized scheduling results. In Section 3 the HCDG-based list-scheduling heuristic is presented in detail. It efficiently handles conditional behaviors by combining many scheduling techniques into the same framework and by defining a novel probabilistic priority function. In Section 4 experimental results on known benchmarks are given; Section 4.1 gives a direct comparison with other scheduling approaches based both on experimental results and qualitative arguments. Finally, in Section 5 conclusions are drawn and topics of future work are identified.

2. HIERARCHICAL CONDITIONAL DEPENDENCY GRAPH

There is some agreement that design representation is not a mature topic in HLS when intensive dataflow is combined with complex controlflow. The legacy of CDFG-based approaches is strong, however, the HLS community is certainly not looking forward to seeing yet another flavor of CDFG with special control nodes (which, by the way, is not the case with HCDGs). Our answer to this is a representation that is part of the research continuum and as such it builds upon and extends previous work. It also introduces two new elements: a hierarchical control representation and the explicit representation of both data and control

dependencies with the possibility of incrementally rearranging the latter in order to provide maximal parallelism exposure.

The HCDG is an elaborate representation suitable for designs where data and control flow are entangled together within a complex design. The representation is more general than traditional DFG or CFG representations and simpler than most CDFG representations in the literature. Furthermore in a hardware/software codesign context HCDGs can adequately accommodate subsystems targeting different implementation domains (software or hardware) in a unified framework. For a custom hardware implementation, being able to express the maximum parallelism at the design representation stage is essential since such implementations are potentially more parallel than a software implementation executing on a fixed hardware platform. However, this last point should be taken with caution since exploiting parallelism is also essential in deriving efficient software for ILP architectures. In the rest of the article a case is made in favor of the HCDG by considering the HLS scheduling problem, but the HCDG is much more than just a convenient support for heuristics.

Although in the proposed representation many of its constituent concepts were elaborated on in previous work, innovation stems from combining all of them in a coherent framework. In order to better situate the HCDG in the design representation landscape and see its merits, some comparative discussion is needed.

2.1 The HCDG Compared to Other Design Representations

The shortcomings of traditional design representations were initially underlined by others in Chaiyakul et al. [1992] and in Bergamaschi et al. [1997]; in our previous work [Kountouris and Wolinski 1999a, b] it was thoroughly explained and experimentally demonstrated why the adoption of a new intermediate representation, such as the HCDG is a good idea in the context of high-level synthesis to avoid the negative effects of syntactic variance. The HCDG is a special kind of directed graph that represents both data and control dependencies from a dataflow perspective. Although the HCDG conceptually has many similarities to CDFG-based models, its main difference is a hierarchical control representation that permits us to safely and efficiently perform HLS activities (i.e, scheduling, resource allocation, binding, etc.).

HCDG represents control and data dependencies at the same level of detail from a pure dataflow perspective. Thus, for a description having no control flow, the HCDG essentially reduces to a DFG. Therefore, it can be said that HCDG includes the expressive power of DFGs for dataflow-dominated descriptions and well-established DFG techniques and algorithms are readily applicable on HCDGs. Compared to CFGs, the HCDG are also more general in the sense that control flow does not prime over dataflow and both types of dependencies are node (operation) precedence constraints. In HCDGs there is no predefined node order dictated by the input description and thus we have the possibility of choosing the most appropriate order according to the scheduling optimization criteria. In this sense HCDGs are different from hierarchical task graphs (HTGs) [Girkar and Polychronopoulos 1992], which also prime control

over data dependencies. In addition, HCDGs represent a predicated static single assignment (P-SSA) form [Carter et al. 1999] of a program whereas this is not the case of HTGs. SSA, used in VLIW compilation, removes false data dependencies across basic blocks that find their way into HTGs or other types of controlflow-based representations. In respect to CDFGs there are conceptual similarities and essential differences. For instance, fork nodes that define basic block limits do not explicitly exist and thus ad hoc grouping (based on input description) of operations into basic blocks is avoided. Thus by using HCDGs we have a lot more flexibility in performing global dataflow analysis and discovering parallelism across basic blocks (in CDFG terms). This in turn facilitates considerably the application of optimizing graph transformations such as global code motions and the associated bookkeeping. For instance, Dos Santos et al. [2000] couple the CDFG to a BasicBlock-CFG in order to manage the complexities associated with global code motions and bookkeeping in the CDFG context. It is shown later that global code motions have equivalent representations in the HCDG context and essentially consist of a mix of control dependency rearrangement (for lazy and speculative execution) and node replication transformations.

Another point of comparison with other design representations is the special control representation adopted in the HCDG. In previous work special control representations have also been defined and associated either with CFG [Bergamaschi et al. 1992], CDFG Radivojevic and Brewer [1995, 1996], assignment decision diagrams (ADD) [Juan et al. 1994], or other design representations proposed in the compiler literature such as hierarchical task graphs with their execution tags [Polychronopoulos 1991]. Conceptually the control model of Radivojevic and Brewer [1995, 1996], also used by the global scheduling approach of Dos Santos et al. [2000], is similar to ours. The originality in the HCDG approach is the derivation of the control hierarchy (in a sense a hierarchical BDD) representing guard inclusion relations which is important information in optimizing the result of a variety of HLS activities. Perhaps the most conceptually similar (but quite different in terms of implementation as well as capabilities) to HCDG is the ADD [Chaiyakul et al. 1992] coupled with the CG [Juan et al. 1994]. The major difference is that the CG has no explicit hierarchy and since it is not based on BDDs it also has some inherent inefficiencies. In addition, since control dependencies in the ADD remain implicit, optimizing transformations for control dependency rearrangement during scheduling are not obvious to engineer.

2.2 HCDG Overview

Even though the focus of this article is not on the HCDG and its merits as a design representation, some basic notions that will help in understanding how the HCDG is used are needed. In this section the HCDG is briefly presented by means of a simple example which is used throughout the rest of the article for illustration purposes. A brief overview of the HCDG construction and mutual exclusiveness issues follows next. More details on the formal definitions of the HCDG can be found in Le Guernic et al. [1991], Besnard [1991], and

```
process jian(a, b, c, d, e, f, g, x, y)
in port a[8], b[8], c[8], d[8], e[8], f[8], g[8];
in port x, y;
out port u[8], v[8];
{
static T1, T2[8], T3[8], T4[8], T5[8];

        T1 = (a +1 b) < c;
H1 →    T2 = d +2 e;
        T3 = c +3 1;
H2 →    if (y) {
H6 →            if (T1)
                        u = T3 +4 d; /* u1 */
H10 →           else if (!x)
                        u = T2 +5 d; /* u2 */
H9 →            if (!T1 && x)
                        v = T2 +6 e;
        } else {
                T4 = T3 +7 e;
H3 →            T5 = T4 +8 f;
                u = T5 +9 g; /* u3 */
        }
}
```

Guard	Boolean Definition	Guard	Boolean Definition
H_1	1	H_6	$y \cdot T_1$
H_2	y	H_7	$y \cdot \overline{T_1}$
H_3	$\bar{y}$	H_8	$y \cdot T_1 + y \cdot \overline{T_1} \cdot \bar{x}$
H_4	$\bar{y} + y \cdot T_1$	H_9	$y \cdot \overline{T_1} \cdot x$
H_5	$\bar{y} + y \cdot T_1 + y \cdot \overline{T_1} \cdot \bar{x}$	H_{10}	$y \cdot \overline{T_1} \cdot \bar{x}$

Fig. 1. *Jian* benchmark description (left) and guard definitions (right).

Amagbegnon [1995]. Kountouris and Wolinski [1999a, b] give more details on the effective use of guard information in HLS activities.

Our example, taken from Li and Gupta [1998], exhibits a high conditional behavior. In the following it is referred to as the *jian* benchmark. Its textual description is shown on the left of Figure 1. The table on the right of Figure 1 lists the Boolean conditions, explicitly or implicitly present in the description, under which operations are executed and values are assigned to variables. A symbolic name H_i is given to each condition. For instance, operation $+_4$ is executed under $H_6(y \cdot T_1)$ and variable u is assigned a value under H_5 $(\bar{y} + y \cdot T_1 + y \cdot \bar{T}_1 \cdot \bar{x})$ which represents all three possible assignments to u (H_4, $\bar{y} + y \cdot T_1$, is simply a factor of H_5). As explained later these conditions are guards and are organized in a hierarchy through a Boolean factorization process. In Figure 2, the textual benchmark description was parsed into an HCDG. Some details on the HCDG construction is given in Section 2.3. To avoid cluttering Figure 2, guard details are shown separately in Figure 3.

In the HCDG graphical representation graph nodes are represented by rectangles and ovals. Rectangles correspond to guard condition nodes and ovals to operation nodes (I/O, computation, data multiplexing, and storage elements with either register or transparent latch semantics). Each operation node contains the operation symbol. I/O nodes have their names prefixed by "?" and "!," respectively. Guards are identified by the guard name and have a dependency from the subgraph corresponding to the Boolean expression defining them. Graph edges are precedence constraints (dependencies) on the nodes shown by dashed and solid arrows which represent control and data dependencies,

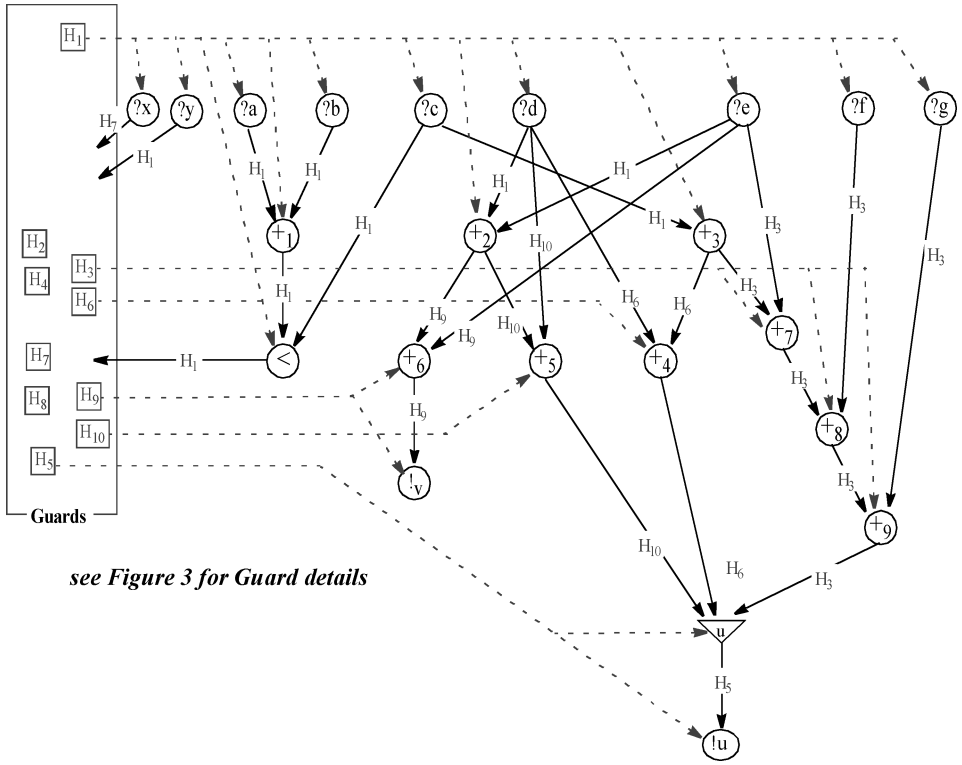

Fig. 2. HCDG of the *jian* benchmark; guard details are shown separately.

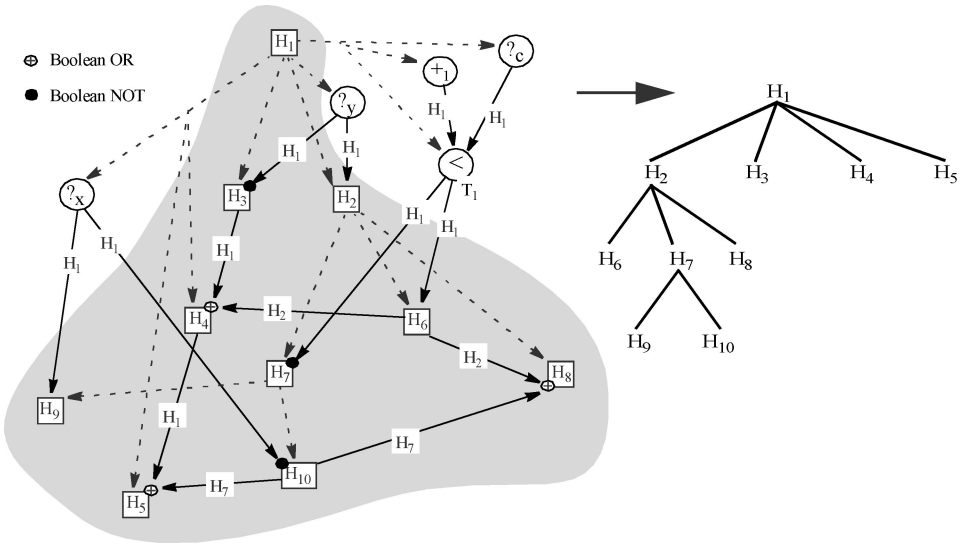

Fig. 3. Guard dependency graph (left) and guard hierarchy (right).

respectively. The former indicate the values of the nodes that need to be computed before the value of another node is computed, and the latter indicate which conditions need to evaluate to *true* before a data value can be computed (and considered as valid). Each node has a control dependency from its guard. Both HCDG nodes and edges are labeled by guards.

The HCDG obeys the *static single assignment* principle. For instance, in Figure 2 the data multiplexing node u (shown as a triangle) is introduced to enforce the single assignment principle for variable u (in the behavioral description) which has multiple definitions (u_1, u_2, u_3) under mutually exclusive conditions (guards H_6, H_{10}, H_3, respectively). In this example we see that guards can be defined by input Boolean variables (i.e., the *true*/*false* outcome of y defines H_2, H_3 computed Boolean variables (i.e., the result of the "<" node defines H_6, H_7 which are under H_2, or composite Boolean expressions defined by means of other simpler guard Boolean definitions (e.g., H_4 which is defined as the Boolean sum of H_3 and H_6). In Figure 1 (right) the resulting guard Boolean definitions are given.

Guards, as briefly explained earlier, are a special type of node representing Boolean conditions that guard the execution of nodes operating on the data. In a discrete-time model where time is considered as an infinite sequence of logical instants, a *guard* is considered as the set of logical instants that the Boolean condition defining it evaluates to *true*. The theoretical foundations of the HCDG consider guards as sets and guard formulas as the application of set operations on them. Equivalent representations of guard formulas as Boolean functions can be obtained and vice versa [Amagbegnon 1995]. Guards are equivalence classes of the HCDG nodes, meaning that nodes labeled by the same guard are active (carry a value) at the same logical instants. What is important is that inclusion relations can be determined on guards. Based on this inclusion relation it can be said that a guard occurs more or less frequently than another guard. Guards are organized in a treelike guard hierarchy (GH) which represents these inclusion relations.

Inclusion Relation: Lets denote by h_i the Boolean function corresponding to guard H_i.h_i evaluates to *true* whenever H_i is present and otherwise to *false*. The inclusion relation represented by the treelike structure of the guard hierarchy simply states that

$$\forall(H_j \in descendants(H_i)) \Rightarrow H_j \subset H_i.$$

In terms of their Boolean definitions this translates to the implication relation:

$$h_j = true \Rightarrow h_i = true.$$

In addition, inclusion can be extended to the following cases.

$$H_k = H_i \cup H_j \Rightarrow H_i \subset H_k, H_j \subset H_k \quad \text{and} \quad H_k = H_i \cap H_j \Rightarrow H_k \subset H_i, H_k \subset H_j,$$

where the set operators *union* and *intersection* correspond to Boolean *and* and *or*, respectively.

Such information is important in order to triangularize a larger number of systems of guard equations than would be possible by using a rewriting system based on the axioms of Boolean algebra [Besnard 1991]. In Amagbegnon

[1995], the guard hierarchy is implemented as a hierarchy of BDDs. Using BDDs two things are achieved: equivalence between guard formulas can be easily established resulting in a minimal representation by avoiding redundancy, and, hierarchy, during the hierarchy process by factoring it is easy to find the maximum depth in the tree to which a guard can be inserted, thus obtaining an optimally refined inclusion hierarchy. Control representations based on BDDs have already been used in previous work [Bergamaschi et al. 1992; Radivojevic and Brewer 1995, 1996; Dos Santos et al. 2000]. The CG, described in Juan et al. [1994], is another special control representation for which Radivojevic and Brewer [1995] argue that is less efficient than a BDD-based one. The originality of the hierarchical control representation as BDD trees, described in Besnard [1991] and Amagbegnon [1995] and adopted in this work, lies on the hierarchy construction and not on the use of BDDs. BDDs are simply used for their efficiency. Guard inclusion is very important in efficiently detecting mutual exclusiveness minimizing the number of necessary mutual exclusiveness tests, as demonstrated in Kountouris and Wolinski [1999a] and Kountouris [1998], where free-from-syntactic-variance mutual exclusiveness detection techniques were described.

2.3 Deriving HCDGs from High-Level Descriptions

To construct the HCDG internal representation from a behavioral description the following methods are envisioned. The first consists of parsing source code in some conventional high-level language and the second in using the SIGNAL dataflow formal specification language. To construct an HCDG from descriptions using standard description languages (e.g., C, VHDL, HardwareC, etc.) necessitates a construction process similar to the one used in Chaiyakul et al. [1992] or algorithms that compute the static single assignment form of imperative programs [Cytron et al. 1989]; of particular interest for the HCDG is the predicated static single assignment [Carter et al. 1999]. In addition the hierarchy process described in Besnard [1991] and Amagbegnon [1995] is also applied to obtain a refined guard hierarchy. Guard exclusiveness information may also be introduced including information about guards defined by arithmetic relations. How this information is obtained is described in detail in Kountouris and Wolinski [1999a]. Finally, a third HCDG construction option is to automatically obtain from an imperative style description its SIGNAL equivalent applying strict translation rules. For instance a translation for Statecharts is described in Beauvais et al. [1998].

Although in this article we deal with descriptions having no iterative constructs, the HCDG can handle them but the construction process becomes more complicated; the aforementioned process applies for the loop bodies. Deriving HCDGs of descriptions containing iterative constructs is the topic of current work not yet completed. However, a brief overview is given here for the sake of completeness. Our treatment of looping constructs is inspired by the ADD approach [Chaiyakul et al. 1992], where loops imply state transitions and a system FSM can be extracted by translating the program CFG into an FSM. However, as pointed out in Lakshminarayama et al. [1999], this approach has

the inconvenience that sequential execution semantics find their way in the resulting graph implicitly in the state transition order and so parallel (independent) loops (i.e., concurrent states) cannot be inferred. In our approach it is as if instead of using the program CFG to derive the FSM, we used the corresponding hierarchical task graph [Girkar and Polychronopoulos 1992] where loop parallelism is explicit and HTG execution tags correspond to the HCDG activation guards of the composite HCDG nodes representing loops. Loop nesting levels can be regarded as description hierarchy levels. Each level can be represented by an HCDG containing composite HCDG nodes corresponding to loop bodies. In this way a loop body can be abstracted by a composite HCDG node with an activation guard as well as inbound and outgoing guarded data and control dependencies. Inside, a composite HCDG node is represented by an HCDG subgraph capturing the loop body processing. At each level data dependency analysis permits us infer parallel loops. On the other hand, iterative constructs in a system description imply system state. Therefore a description can be analyzed to extract a hierarchical FSM that can be represented by an HCDG (as the Statecharts translation [Beauvais et al. 1998] demonstrates). The FSM hierarchy is the direct result of loop nesting. Parallel (independent) loops result in concurrent FSMs within the global system FSM. Inferring the system FSM permits the construction of a basic skeleton of the HCDG's guard hierarchy. A guard corresponds to each FSM state. These guards become the activation guards of the composite HCDG nodes representing loop bodies. These activation guards become in turn the root guards of the HCDG subgraph representing the loop body. Under it, the loop condition defines complementary guards corresponding to FSM state transitions. One corresponds to the loop iteration condition; another one to the fall-through (or loop exit) condition. An extra guard may correspond to the loop entry condition used for initialization of loop variables (i.e., useful with for-loop constructs). Guards defined inside the loop body are placed under the iteration condition guard.

2.4 Mutual Exclusiveness Discovery and Representation

The mutual exclusiveness of the conditions under which operations execute is important information for the scheduling of conditional behaviors. This information permits us to conditionally share resources and schedule operations efficiently. Using the HCDG reasoning on condition mutual exclusiveness is equivalent to reasoning on guard mutual exclusiveness. *Guard exclusiveness* is denoted by $\otimes$. Two guards are mutually exclusive if their intersection is empty: $(H_1 \cap H_2 = \oslash) \Leftrightarrow H_1 \otimes H_2$. To identify if two guards are mutually exclusive all we have to do is find whether the Boolean product their BDD representations in the guard hierarchy is *false*; that is, $h_1 \cdot h_2 = 0 \Leftrightarrow H_1 \otimes H_2$. To identify mutual exclusiveness of guards and include this information in the HCDG for later use, the exclusiveness list of a guard is defined. This list, denoted by *excl_list(H)*, is a set of guards satisfying the property *excl_list(H)* $= \{(H_i \in N_H) : H \otimes H_i\}$, with N_H the set of all guards in the hierarchy. To construct these lists, the straightforward approach is to perform a test for mutual exclusiveness for each *pair* of guards. For a large number of guards this can be very expensive. The *inclusion*

Guard	Exclusiveness List	Guard	Exclusiveness List
H_1	-	H_6	H_3, H_7, H_9, H_{10}
H_2	H_3	H_7	H_3, H_4, H_6
H_3	H_2, H_6, H_7, H_8, H_9, H_{10}	H_8	H_3, H_9
H_4	H_7, H_9, H_{10}	H_9	H_3, H_4, H_6, H_5, H_8, H_{10}
H_5	H_9	H_{10}	H_3, H_4, H_6, H_9

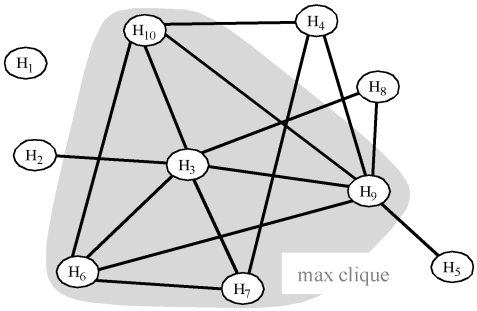

Fig. 4. Guard mutual exclusiveness for *jian* (left) and MEG (right).

relations represented by the guard hierarchy can be used to reduce the number of mutual exclusiveness tests [Camposano 1991]. In the worst case of a flat guard hierarchy, for n guards we need to perform $n!/(2! \cdot (n-2)!$ mutex tests (n by 2 combinations). For the *jian* example (Figure 1) due to guard inclusion relations, instead of 45 tests (for $n = 10$ guards), only 20 tests are needed. Also conditions defined by simple arithmetic relations are treated as described in Koutouris and Wolinski [1999a] and Kountouris [1998] increasing the capabilities of our method with respect to other approaches [Radivojevic and Brewer 1995; Camposano 1991].

To efficiently represent and subsequently manipulate the exclusiveness information, we use a special graph representation named MEG. A *mutual exclusiveness graph* (MEG) is an undirected graph and consists of a set of vertices V and a set of undirected edges E. Such graphs are mentioned in the literature as *compatibility graphs*. Each vertex v in V corresponds to a guard in the HCDG and each edge $e = (u, v)$ in E represents the mutual exclusiveness of the guards represented by vertices u, v. For our example guard mutual exclusiveness information is shown in Figure 4. This graph representation facilitates testing two guards for exclusiveness and it also facilitates finding groups of pairwise mutually exclusive guards. Such groups correspond to MEG *cliques*. Heuristics such as those in Tseng and Siewiorek [1986] to find max cliques and minimum clique partitions (NP-complete problems [Garey and Johnson 1979]) can be used. A detailed discussion on these topics and experimental results can be found in Kountouris and Wolinski [1999a] and Kountouris [1998].

2.5 HCDG Transformations

Graph transformations or graph restructuring is not a new concept. It has been used often in previous work mainly related to scheduling optimization. For instance, in *tree-based scheduling* [Huang et al. 1993] deriving the tree representation form from the CFG necessitates code motions that correspond to lazy execution as well as operation duplication into mutually exclusive control branches. In Juan et al. [1994] variable assignment conditions are computed for lazy execution as well. An instance of CDFG node duplication is introduced in Wakabayashi and Yoshimura [1989, 1992]. Speculative execution has been either represented by code motion [Dos Santos et al. 2000] (in this case

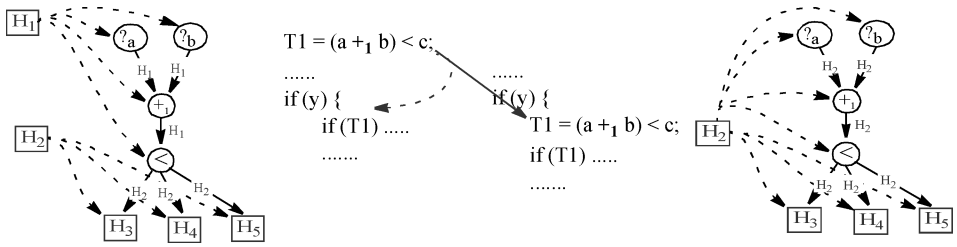

Fig. 5. Lazy execution transformation.

compensation code can be viewed as a sort of node replication), guard expression recalculation Radivojevic and Brewer [1996], or dynamic condition vectors [Wakabayashi and Yoshimura 1992]. In Kim et al. [1994] conditional sharing transformation is used before scheduling to reflect a priori decisions in conditional resource sharing and in Chaiyakul et al. [1992] such a transformation can be used either before or during scheduling. Finally, reasoning on false paths was used to refine path enumeration in path-based scheduling [Bergamaschi et al. 1997].

In the following HCDG transformations are explained in more detail and by means of simple examples it is shown that their application on the HCDG is quite simple. In Section. 3.1 HCDG transformations are revisited in order to explain in more detail how they are used to optimize the results of HCDG-based scheduling.

Lazy Execution Transformation. This transformation consists of replacing the *definition guards* of the nodes by their *use guards*. The *definition guard* (*d_guard*) gives the condition under which a node is defined in the high-level specification and corresponds to the condition under which a value is computed. The *use guard* (*u_guard*) corresponds to the condition under which a value is used by other nodes in subsequent computations; it is calculated as the Boolean sum of the guard conditions associated with a node's outgoing edges. The reasoning in Chaiyakul et al. [1992] to determine assignment conditions is similar. To find *u_guards* a recursive HCDG traversal starting from the output nodes and going upwards towards the input nodes is used. Figure 5 depicts a piece of the HCDG of the *jian* example; the value of T_1 is computed more often (under H_1) than it is used (under H_2 as shown by the outgoing dependencies of node $<$) so simply changing the control dependencies of nodes $<$, $+_1$, ?a, ?b from guard H_1 to guard H_2 (and consequently the data dependency labeling) enforces lazy execution and a value for T_1 is computed only as often as it is used.

The lazy execution transformation ensures that operations will be executed only as often as their computed results will be used in subsequent computations. In the case of a high- level specification written in some imperative language, this transformation corresponds to downward code motion (i.e., from outside conditionals we move statements inside conditionals as is the case in Dos Santos et al. [2000] and Huang et al. [1993]).

Node Duplication Transformation. This transformation duplicates the nodes whose results are used under mutually exclusive conditions; an example

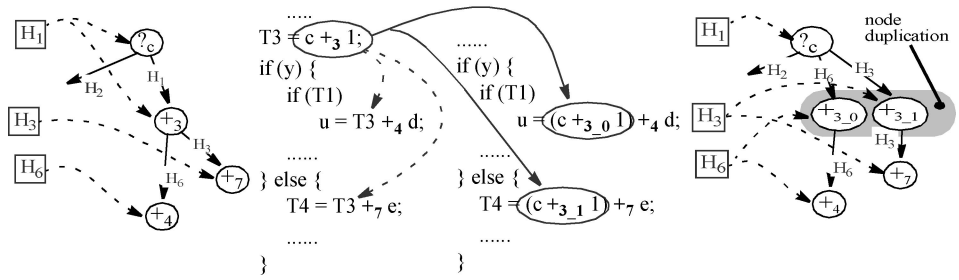

Fig. 6. Node duplication transformation.

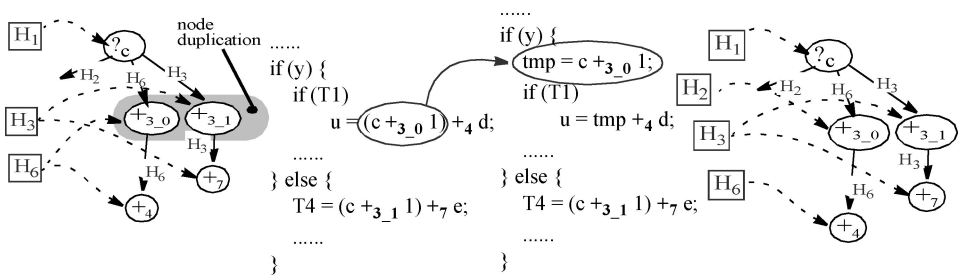

Fig. 7. Specualtive execution transformation.

is shown in Figure 6. In many cases this transformation may reduce the length of control paths and may also offer more possibilities for conditional resource sharing. Even though the number of nodes increases this effect will be partially undone during scheduling by the conditional resource sharing policy of the scheduling algorithm. In order to limit the increase of replicated nodes we may choose to apply this transformation after the lazy execution transformation and impose as a constraint that a node is replicated as long as there is at least a control path to one of the replicas that is shorter than the control path into the initial node. Another type of node duplication (introduced in Wakabayashi and Yoshimura [1989, 1992]) occurs when an operation node succeeding a data merge node is moved before it. This necessitates replicating the operation node at each incoming path to the data merge node and possibly creating a new data merge node to connect the replicas to the successors of the initial operation node.

Speculative Execution Transformation. This transformation consists of changing the guard of a node for one of its ancestors higher up in the guard hierarchy. Speculative execution in some cases can be considered as upward code motion where a statement is moved before the control expression that defines a basic block. In Figure 7 we take the example of Figure 6 a step further and we see that by speculating node $+_{3_1}$ we minimize the length of its control dependency path. The initial node guard (H_6) is substituted by one of its ancestors in the guard hierarchy (H_2) and control dependencies are changed accordingly.

False Path Elimination Transformation. In a behavioral description false paths may be contained and thus unnecessary dependencies. With the HCDG

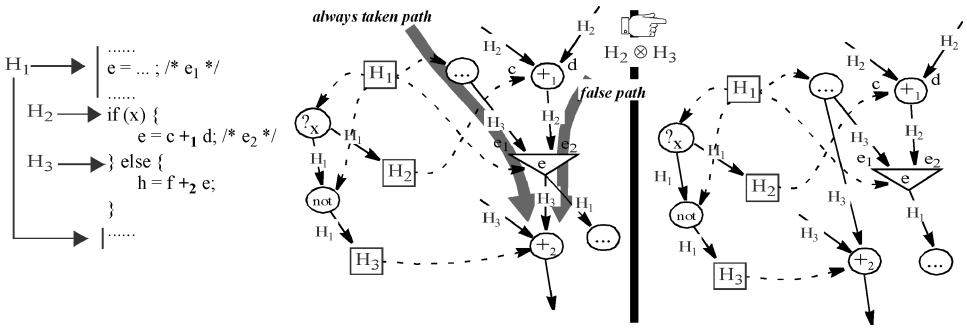

Fig. 8. Transformation for false path elimination.

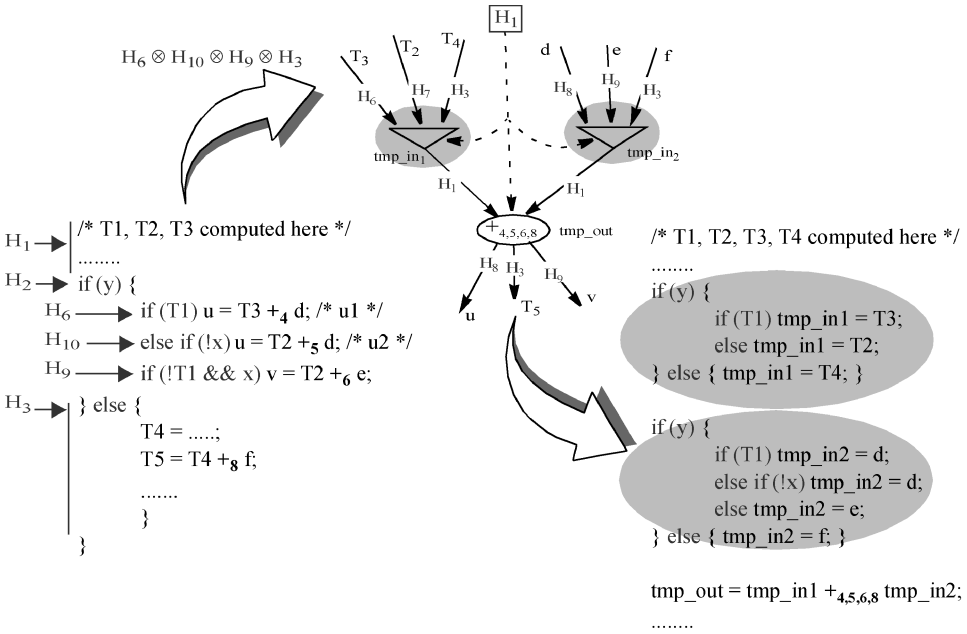

Fig. 9. Conditional resource sharing transformation.

it is very simple to identify and eliminate them since all the HCDG edges are labeled by guards. If the Boolean product of the guards of the edges along a path is always *false* then this path can be eliminated by dependency rearrangement. This is shown in the example of Figure 8.

Resource Sharing Transformation. This transformation corresponds to merging operation nodes that may conditionally share the same functional unit into a single node. This necessitates introducing data merge nodes to multiplex the inputs of the initial nodes into the inputs of the new common node. In order to route the appropriate input data to the inputs of the new node the multiplexing control logic is derived from the guards of the merged nodes. Figure 9 graphically depicts this transformation for the example of Figure 2 based on

the scheduling decisions about conditional sharing shown in Figure 13. The add nodes $+_4$, $+_5$, $+_6$, $+_8$ have mutually exclusive guards and can be scheduled in the same control step conditionally sharing the same FU.

3. HCDG-BASED LIST-SCHEDULING HEURISTIC

Our list-scheduling scheme under resource constraints has three inputs: the HCDG, the priority function, and the scheduling constraints consisting of the set of available functional units (FUs) and clock period (cycle time). The novelty of this work is in the choice of the priority function and the policy of scheduling nodes on the available resources. Moreover, our scheduling core supports operation chaining and multicycle operations. The advantage of using as input the priority function is that the same scheduling core can be used with different priority functions so different priority functions can be used for different types of designs. In addition, as more effective priority functions become available their integration into our system is straightforward.

It is worth mentioning at this point that the quality of the scheduling results heavily depends on the application of HCDG transformations. Their mechanics were described in Section. 2.5. Here before describing the scheduling philosophy (Section. 3.2) several aspects of the HCDG transformations relevant to the scheduling problem are presented first. These aspects concern mainly when and how these transformations are applied.

3.1 HCDG Transformations for Scheduling Optimization

For conditional behaviors, represented by HCDGs, scheduling optimization comes from two sources. Both rely on HCDG transformations. The first source is transformations that are applied before scheduling to minimize the influence of *syntactic variance* and the second source consists of transformations that are applied during scheduling to optimize resource utilization.

The term *syntactic variance* describes semantically equivalent (i.e., same behavior) but syntactically different input descriptions. This is due to different description styles such as statement order, statement location within conditional constructs, and nesting of conditionals. In certain representations such as CFG and CDFG, description style influences their construction and form. So the scheduler may be given different input for the same behavior and thus produce different results. In addition, if the employed algorithms rely on description structural information to extract information useful in their decision making, such as condition mutual exclusiveness information, then results will also be different. From the previous discussion and as explained in Chaiyakul et al. [1992] and Li and Gupta [1998], syntactic variance influences synthesis results at two different but closely related levels: the design representation construction and mutual exclusiveness detection both used in scheduling conditional behaviors.

In the HCDG case syntactic variance influences the construction process of the initial HCDG given as input to the scheduler. To cope with this we rely on graph transformations applied before scheduling. Such prescheduling transformations mainly have the intention of obtaining an as much as possible "unique"

representation (free from syntactic variance in the sense of Chaiyakul et al. [1992]). This, in turn, for syntactically different design descriptions, will provide a common scheduling input and thus the same scheduling results will be obtained for all of them when the same scheduling heuristic is used. These transformations are applied on the HCDG in the order:

—*false path elimination*,
—*lazy execution*, and
—*node duplication* under the condition of control path length reduction.

In general terms syntactic variance influences the amount of explicit information that is useful in order to obtain good scheduling results. For operations this information is of three types: parallelism, mobility, and mutual exclusiveness.

Different representations of the same behavior make explicit different amounts of these three types of information. After the prescheduling transformations thanks to the dataflow nature of the HCDG operation parallelism and operation mobility become explicit as much as possible. In addition, as explained in Section. 2.4, mutual exclusiveness detection using HCDGs does not suffer from the effects of syntactic variance (discussed in Li and Gupta [1998]). Thus the third type of information also becomes explicit as much as possible. Since the results of mutual exclusiveness detection also influence scheduling lengths by permitting more or less efficient resource utilization in each control step, this is also a first step in minimizing the effects of syntactic variance relative to scheduling.

Finally, for the second time scheduling optimization comes into play. During scheduling we apply on-the-fly several transformation combinations with the intent of increasing resource utilization as much as possible in each control step. Examples of such scheduling optimizing transformation combinations are:

—conditional resource sharing in combination with speculative execution,
—node duplication before data merge nodes in combination with conditional resource sharing. This instance of node duplication is applied during scheduling to increase the resource usage in previous control steps. As explained in Section. 2.5, this necessitates replicating the successor node at each incoming path to the data merge node. As long as all the replicas can be scheduled in some previous control step the transformation is applied. This is similar to the node duplication transformation introduced in Wakabayashi and Yoshimura [1989, 1992].

The exact mechanics of how these on-the-fly transformations are applied is part of the scheduling policy described next. An interesting observation is that transformations during scheduling effectively amortize on a need basis the side effects of prescheduling transformations. Lazy execution may lengthen several control paths. However, during scheduling this effect will be amortized to a certain degree by the speculative scheduling of nodes used by our scheduling algorithm. In a sense speculative execution shortens control paths as needed and as permitted by the resource availability. The overall effect will

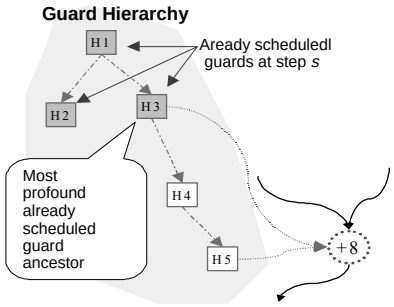

Fig. 10. Dynamic computation of *sguards* for *spec* nodes.

be an optimized scheduling result in terms of both schedule length and resource utilization. The conditional resource sharing transformation, which is applied during scheduling to increase resource utilization, has the side effect of undoing certain *prescheduling* node duplications; this happens whenever replicas created under mutually exclusive guards are scheduled to share the same resource.

3.2 Scheduling Policy Outline

At each scheduling step nodes that can be scheduled on the available resources are partitioned into two sets, *ready* and *spec*, corresponding to nodes that can be scheduled without and with speculative execution, respectively. Nodes in *ready* observe both their data and control dependencies and nodes in *spec* fully observe only their data and partially their control dependencies (weak dependencies in Kifli et al. [1995] is a similar concept). For brevity the former are simply called *ready* nodes and the latter *spec* nodes. At each step, *ready* nodes are scheduled first, exploiting conditional resource-sharing possibilities (in Kifli et al. [1995] *spec* nodes have lower priorities). Once no more *ready* nodes can be scheduled (either due to resource constraints or because the *ready* set becomes empty), we attempt to increase resource utilization by further conditionally sharing already used resources with *spec* nodes. Finally, if unused resources still remain we attempt to utilize them by conditionally sharing *spec* nodes. Nodes are selected for scheduling on a priority basis; priorities are computed by the priority function described in the next section.

Each schedulable node is associated with a schedule guard denoted by *sguard* which indicates the condition under which the node will execute if it is scheduled in the current control step. For *ready* nodes the *sguard* corresponds to the guard labeling the node in the HCDG; hence it is statically available. For *spec* nodes the *sguard* has to be dynamically computed (similarly to guard smoothing Dos Santos et al. [2000], dynamic guards Radivojevic and Brewer [1996], and dynamic CVs Wakabayashi and Yoshimura [1992]). In the HCDG context this *sguard* is found starting from the guard of the speculated (*spec*) node and going upwards following its ancestors in the guard hierarchy. The first scheduled guard ancestor found becomes the speculated node's *sguard*. For instance, in Figure 10 node $+_8$ is defined under guard H_5. Once the data dependencies of $+_8$

are observed the node can be considered ready for scheduling in a speculative manner. Its *sguard* has to be found among the ancestors of H_5 in the guard hierarchy (i.e., H_4, H_3, H_1). Guard H_4, which is the immediate ancestor of H_5, does not qualify because it is not yet scheduled; H_3 is scheduled and so becomes the *sguard* of *spec* node $+_8$. This corresponds to the most profound already scheduled guard ancestor of the *spec* node's guard in the guard hierarchy. At the end of each scheduling step *sguards* of *spec* nodes have to be readjusted in order to account for guards scheduled at the current control step that may correspond to more profound guard ancestors of *spec* node guards. In our example if at some point guard H_4 is scheduled but $+_8$ has not yet been scheduled then its *sguard* will be adjusted to H_4. Whenever H_5 is scheduled, after *sguard* adjustment $+_8$ will become *ready* in a normal manner since now both its data and control dependencies are observed. Using this *sguard* adjustment technique we maximally exploit conditional resource sharing possibilities even for nodes that are speculatively executed.

3.3 An Adaptive Probabilistic Priority Function

To obtain the node priorities a special priority function based on guard hierarchy information and node mobilities is elaborated. The priority function for a node n, $pr(n)$ is based on the three sorting keys:

$$pr_1(n) = awurg(n), \qquad pr_2(n) = p_s(n), \qquad pr_3(n) = wprd(n).$$

To sort nodes in the *ready* lists, every time the first key for two nodes is equal the second is used, and if this is equal too then the third is used. This three-level priority function accounts for the conditional nature of the scheduled behaviors by weighting the priority measures using statically computed node execution probabilities. The first key (pr_1) represents a node's scheduling urgency and accounts for the node mobilities in a conditional execution context. The second key (pr_2) further takes into account the conditional nature of the scheduled behaviors giving higher priority to more frequently executing nodes among nodes of equal urgencies. Finally, the third key (pr_3) provides a local measure of how the available resources will be able to accommodate the future resource demands provoked by scheduling a node in the current control step and permits us to mitigate the horizon effect of list scheduling due to a limited observation window. After this brief overview, the meaning of each of these three elements is explained in more detail below.

For each guard its activation probability p is computed. The probability of the root guard is set to 1. This reflects the fact that nodes labeled by the root guard will execute at every execution instance (path). For other guards their probability is computed from their Boolean definition formulas. Every guard has a sum-of-products definition and its probability is given by the sum of products of the corresponding variable probabilities. For Boolean variables that are results of comparison operations and input Booleans a probability of 0.5 is given to both *true* and *false* outcomes. Predicate probabilities have also been used in Wakabayashi and Yoshimura [1989] exploiting the conditional nesting structure of the input description. For the example of Figure 2 the guard probabilities

Table I. Guard Probabilities for the *jian* Benchmark

Guard	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8	H_9	H_{10}
Probability	1	0.5	0.5	0.75	0.875	0.25	0.25	0.375	0.125	0.125

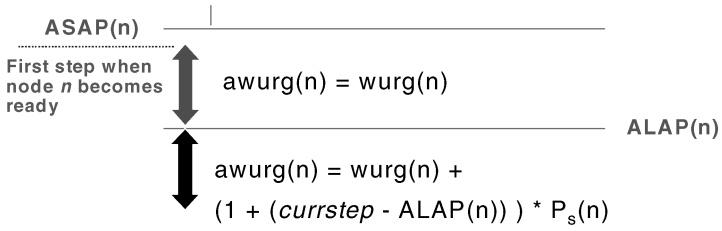

Fig. 11. Node urgency adjustment.

are given in Table I. It is also possible to adjust the guard probability values using application execution profiles obtained by simulation for test inputs in order to obtain guard probabilities; this approach was first proposed for trace scheduling [Fisher 1981] and is adopted in Wavesched [Lakshminarayama et al. 1999] as well. Profiling can be automatically performed on HCDGs as described in Kountouris [1998]. This technique has its inconveniences but nevertheless it is very useful.

Next, ASAP and ALAP schedules are used to obtain node mobilities when both control and data dependencies are fully observed ($mob_{c/d}$) and when only data dependencies are observed (mob_d). The urgency (urg) of a node n is then defined as

$$urg(n) = 1/(1 + mob(n)) \rightarrow \begin{cases} urg_{c/d} = (1 + mob_{c/d}(n)) \\ urg_d = 1/(1 + mob_d(n)). \end{cases}$$

The initial weighted urgency of node n is given by: $wurg(n) = p_s(n) \cdot urg(n)$; depending on the case, either $mob_{c/d}$ or mob_d is used in the formula; p_s corresponds to the probability of the node's scheduling guard ($sguard$). As shown, this coefficient is proportional to the probability of node execution and inversely proportional to node mobility. With this urgency criterion more importance is given to nodes having the highest execution probability. This is an important consideration in a conditional execution context.

At the beginning of each scheduling step initial weighted urgencies of the *ready* and *spec* nodes are adjusted. This adjustment becomes effective once a node has remained unscheduled for a number of iterations higher than its mobility since it became schedulable for the first time (i.e., its scheduling *cstep* under ALAP is attained and the node remains unscheduled as shown in Figure 11).

After this point at each succeeding iteration the urgency is increased by the node's *sguard* probability. This is achieved by:

$$awurg = wurg(n) + a(n) \cdot P_s(n).$$

For *ready* and *spec* nodes $wurg_{c/d}$ and $wurg_d$ are used, respectively, for $wurg$ and $a(n)$ is an adjustment coefficient for node n defined as

$$a(n) = \begin{cases} 0 & \text{if } ((currcstep - ALAP(n)) < 0) \\ 1 + (currcstep - ALP(n)) & \text{otherwise.} \end{cases}$$

The *currcstep* step is calculated by the formula below where *SchedulingTime-OfNode* is the time the processing of a node can start; this is not necessarily the same as the start time of the control step due to operation chaining possibilities.

$$currcstep = \frac{\sum_{\forall Unit} \frac{schedulingTimeOfNode}{executionTimeOfUnit(t)}}{|Unit|}.$$

A *weighted-projected-resource-demand* (*wprd*) measure is also defined as

$$wprd(n) = \frac{\sum_{\forall_t} |R_t| \cdot \sum_{\forall_s \in succ(n,t)} p(h(n,s))}{|R|},$$

where R is the set of all available resources with cardinality $|R|$, R_t is the set of available resources of type t with cardinality $|R_t|$, $succ(n, t)$ are the successors of node n that can be scheduled on the next control step and can be mapped on a resource of type t, $h(n, s)$ is the guard of the dependency from node n to node s, and $p(h(n, s))$ is its probability. With *wprd* we try to measure the impact of the selection of node n in the current *cstep* on the occupation of functional units in the next *cstep*. This is one possibility to limit the horizon effect of the list scheduler.

3.4 Conditional Resource Sharing

Conditional resource sharing is performed accounting for the node priorities. For each resource type a mutual exclusiveness graph, described in Section. 2.4, is constructed. Each vertex, corresponding to a scheduling guard, has an associated list of operation nodes being active under this guard and can be assigned to the resource of that type. Operations in the list are sorted according to their priorities with *ready* nodes sorted before *spec* nodes. The list of nodes that can share a resource is constructed in the following steps.

Step1. *ready* nodes are considered; the clique containing the higher priority node and a maximum number of other high priority nodes is iteratively constructed.

Step2. *spec* nodes are considered; the clique found in Step 1 is enlarged with guards of higher priority nodes that are candidates for speculative execution.

Step3. For each guard in the clique the first node in its operation list is taken; return list.

In this way the list of nodes that can be scheduled sharing a resource is obtained. The best adapted algorithm to find such cliques is based on the initial-graph-partition algorithm presented in Puri and Gu [1992]. Heuristics such as those of Tseng and Siewiorek [1986] are not as well adapted to satisfy our clique construction objectives since, in our case, clique maximality is not a good optimization objective. A clique is iteratively constructed. At each iteration a seed vertex containing the highest priority node is selected and the vertices not

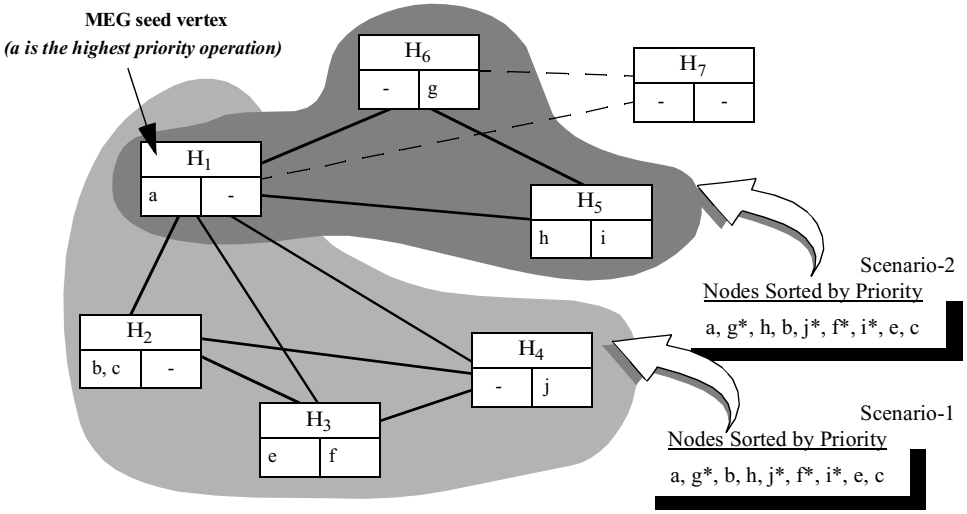

Fig. 12. Clique construction in $cstep_i$ for FU of type t.

connected to it will not be considered in future iterations. Initially, the seed vertex corresponds to the highest priority node. Normally executing nodes (*ready*) are considered before candidates for speculative execution (*spec*). Once there are no more vertices the clique with the desired properties is returned. At each scheduling iteration, the conditional resource-sharing process is repeated for each resource type and for each available resource of such type. A clique found in this way may be sometimes split due to chaining if the time necessary to execute certain nodes in the clique is too important to fit all of them in the same control step.

The searching process for cliques has the following characteristics: The list-scheduling priority criterion is satisfied for the greatest number of distinct execution instances (paths) simultaneously. The constructed clique contains the highest priority node and the largest number of other higher priority nodes that can share a resource with it. Further increase in resource utilization is achieved by enlarging the initial clique with nodes that are candidates for speculative execution. The list-scheduling priority criterion is also satisfied as before.

The example of Figure 12 graphically depicts the clique construction for operations that can be scheduled, normally or speculatively, during a *cstep* on a resource of a particular type. Two scenarios (Scenario-1, Scenario-2) for operations priorities that lead to different cliques are shown. *Spec* nodes are indicated by *. Operations are partitioned by their scheduling guards and are inserted in the operation lists of the MEG vertices (shown as rectangles) according to their priority; *ready* and *spec* operations are kept in separate lists shown in the rectangles under the guard name. MEG vertices with empty lists are not considered during the clique construction process (their edges are shown as dotted).

Operation a has the highest priority in both scenarios so it will be present in the constructed clique in both cases. The shaded regions in Figure 12 delimit

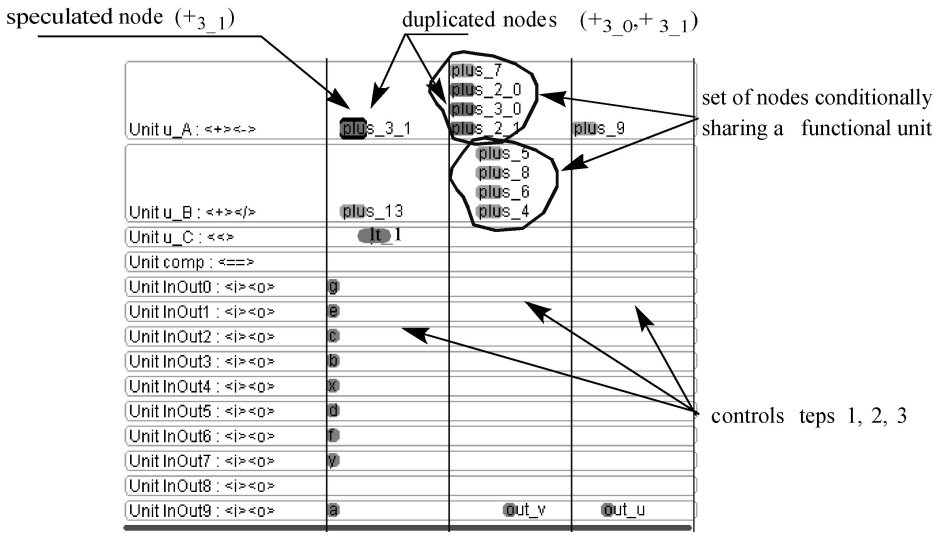

Fig. 13. Schedule obtained for the *jian* benchmark.

the constructed clique for each scenario. For the scenario shown at the bottom when only *ready* operations are considered, the partial clique $\{H_1, H_2, H_3\}$ is constructed in this order since $pr(b) > pr(e)$; next *spec* operations are taken into account and the clique becomes $\{H_1, H_2, H_3, H_4\}$ and so operations a, b, e, j^* (i.e., first entries in the nonempty operation lists of the MEG vertices in the clique) will conditionally share a resource of type t in this *cstep*. In the top scenario where $pr(h) > pr(b)$ a different clique will be constructed by the same process. This is $\{H_1, H_5, H_6\}$ and in this case operations a, h, g^* will conditionally share the resource of type t in this *cstep*.

Finally, a point that needs some explanation is our choice of considering normally *ready* operations before speculatively *ready* operations. If both are considered simultaneously under tight resource constraints it may happen to schedule a speculative candidate instead of a normally *ready* operation. In some cases this may (but not necessarily) result in lengthening of the critical paths in the lazy execution HCDG. Our objective is first to avoid lengthening such critical paths and second to shorten them if possible by means of speculation. This has been confirmed by experimentation; however, we acknowledge that this issue needs to be further investigated.

Figure 13 shows the schedule obtained for the *jian* example benchmark using the *CODESIS* HLS tool. The first column corresponds to the available functional units where the FU identifier and the type of operations it can perform are given. Each of the rest of the columns corresponds to a control step. Each line corresponds to an FU. In each *cstep* operations are scheduled on FUs; when multiple operations are scheduled on the same FU in the same *cstep* this implies conditional resource sharing as shown in the figure for FUs *Unit_A*, *Unit_B* in *cstep* #2. This schedule was obtained when node $+_3$ was duplicated using the transformation described in Section. 3.1 (Figure 6). This duplication

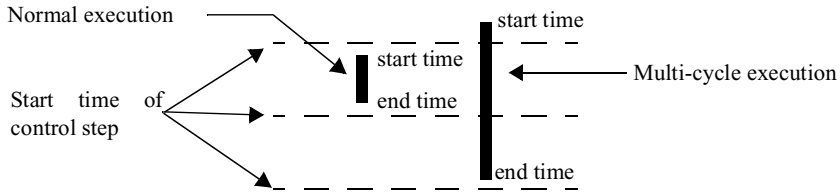

Fig. 14. Start times of scheduled *ops* within and across *cstep* boundaries.

caused reduction of graph depth and improved conditional resource sharing. Node $plus_3_1$ ($+_{3_1}$) is scheduled in the first control step by speculation.

3.5 Considerations for Operation Chaining and Multicycle Operations

In the context of scheduling conditional behaviors under resource constraints and with the possibility of conditional resource sharing, the notion of resource (functional unit—FU) availability becomes more complicated. This complexity further increases when operation chaining and/or multicycle operations are considered. Thanks to guard information this complexity can be managed. Before going further, it should be said that in our system each operation can start at any time within its scheduled control step and can finish either before the end time of the control step, when no multicycle operations are allowed, or in some subsequent control step if multicycle operations are allowed. This is graphically depicted in Figure 14 and depends on the chosen scheduler mode which is a scheduling parameter.

The following conditions must be satisfied to consider an FU as *"not busy"* (*available*):

$$(NotBusy(FU) = true)$$
$$\Leftrightarrow (1. T_a + T_e \leq T_{eac}) \text{ and } \begin{cases} 2. SN = \emptyset \\ \text{or} \\ 3. \forall (G_a, G_b) \in \{SG \times SNG\} \Rightarrow G_a \otimes G_b, \end{cases}$$

where T_{sac}, T_{eac} are, respectively, the start and end times of the current control step; T_a is the current time; T_e is the execution delay in time units of operations that can be scheduled on *FU*; *SN*, is *the* set of scheduled nodes on *FU* in the interval I ($I = T_{sac} - (T_a + T_e)$), *SG* is the set of guards associated with nodes in *SN*; *SNN*, *SNG* are the sets of new nodes to schedule and the set of their associated guards, respectively. The first condition says that an *FU* can be considered as potentially available during a *cstep* only if there is enough time remaining given a start time and an operation delay. Then either the second or the third condition (or both) must also hold. The second condition means that there is no operation currently scheduled on the *FU*. The third condition says that even if operations are already scheduled on a unit this remains available for all those *ready* operations whose *sguards* are pairwise mutex to the *sguards* of the already scheduled operations. Next it is examined how this reasoning is applied in order to handle the scheduling options for multicycle operations and operation chaining.

If multicycle operations are allowed (meaning that some types of operations are allowed to have delays greater than the *cstep* duration) then we have to employ a mechanism to mark a unit as busy for a certain number of subsequent *csteps*; this can be realized by means of a counter. In the case of conditional behaviors where it is possible to follow a plurality of paths depending on the control conditions, the terms *busy* and *not busy* have a more extended significance that relates the conditions (guards) under which an FU is currently used, to the conditions (guards) of the operations that could be scheduled on the FU on a subsequent *cstep*. The FU is considered busy only for the execution paths corresponding to the multicycle operation (or operations if the FU is conditionally shared) currently assigned to it. This means that for operations under *sguard*s that are pairwise mutex to the *sguard*s of already scheduled operations on the FU, the FU may be considered as available (not busy). Thus the busy counter mechanism needs to be extended to handle multiple execution instances: one counter for each execution instance is needed. Once again guard information is very useful in making such distinctions and considerably facilitates the scheduling task when multicycle operations are allowed.

Once scheduling of operations on the available FUs has been performed in a *cstep*, if operation chaining is allowed several conditions must be satisfied. First, there must be enough time left in the *cstep* to schedule an immediate successor of some scheduled operation and second an FU must be available for it. Note that immediate successors may conditionally share the available FU under mutually exclusive guards. For this a clique is constructed for the successors of a scheduled node for each type for which an FU is still available; these are the chaining candidates. When evaluating the chaining conditions guard information needs to be considered at many points. First, whether enough time remains needs to be evaluated for all possible execution instances (paths) due to conditionally sharing an FU among mutex operations whose immediate successors are chaining candidates. For some chaining combinations it may happen that there is not enough time remaining in the *cstep*. Second, a unit may be available for chaining although it may be already used in the *cstep* as long as the *sguard*s of the chaining candidates are pairwise mutex to the *sguard*s of the operations already scheduled on the resource. As an example consider that $+_5$, $+_3$ in Figure 2 are scheduled in a *cstep* and that two adders are available; then $+_4$, $+_7$ which are successors of $+_3$ but under mutex guards ($H_6 \otimes H_3$) can be simultaneously chained using the adder on which $+_5$ is assigned since the guard of $+_5$, H_{10} is pairwise mutex to guards H_6, H_3 (see Figure 4). This example is just a sample of how guard information can be used to maximize chaining possibilities. It is clear that flexible exploitation of chaining is more complicated when scheduling conditional behaviors than in the classical case of dataflow behaviors. Guard information is a valuable tool in coping with this complexity.

4. EXPERIMENTAL RESULTS

The HCDG-based list-scheduling heuristic exploiting conditional resource sharing and speculative execution was tested on a set of benchmarks that have appeared in previous related literature. These are: *kim*, *waka*, *maha*, and *jian*

Table II. Insensitivity to Syntactic Variance

Benchmark	*waka*		*maha*		*kim*		*jian*	
Resources	**cmp**: 1 +: 1, −: 1	**cmp**: 1 **ALU**: 2	**cmp**: 0 +: 1, −: 1	**cmp**: 0 +: 2, −: 3	**cmp**: 2 +: 2, −: 1	**cmp**: 2 **ALU**: 2	**cmp**: 1 +: 1, −: 1	**cmp**: 1 +: 2, −: 1
descr. 1	7/7/4	7/7/4	5/5/4	4/4/2	6/6/6	6/6/6	4/4/4	4/4/2
descr. 2	7/7/4	7/7/4	5/5/4	4/4/2	6/6/6	6/6/6	4/4/4	4/4/2

descr. 1
maximal nesting
use of "if-then-else"

```
.....
T3 = c +3 1;
if (y) {
    if (T1)
        u = T3 +4 d;
    ......
} else {
    T4 = T3 +7 e;
    ......
}
```

```
.....
T3 = c +3 1;
if (!y)
    T4 = T3 +7 e;
......
if (y && T1)
    u = T3 +4 d;
......
```

descr. 2
no nesting
"else" not used
diff. statement order

Fig. 15. Example of different description styles; descr.1 (left), descr.2 (right).

from Kim at al. [1991], Wakabayashi and Yoshimura [1989], Parker et al. [1986] and Li and Gupta [1998], respectively. For each benchmark the HCDG was constructed, the guard hierarchy was refined, and the guard mutual exclusiveness was established using the techniques described in Kountouris and Wolinski [1999a]. Before scheduling is applied the HCDG is transformed for lazy execution and the node duplication transformation is applied wherever appropriate as previously described.

The first experiment (Table II) consists of evaluating the insensitivity of the scheduling results to the effects of syntactic variance. For this our heuristic was applied for each benchmark on two semantically equivalent but syntactically different descriptions.

The first description (descr. 1) has a maximal conditional nesting as opposed to the second one (descr. 2) where all conditions are flattened and each assignment statement is in its own conditional block. An example is given in Figure 15; both descriptions result in the same HCDG and the same guard mutual exclusiveness information.

The insensitivity of our approach to the effects of syntactic variance can be attributed first to the dataflow nature of the HCDG (statement order does not influence the results); second to the prescheduling transformations that set different input descriptions into common form so that the scheduler is fed with the same graph; and third to the guard hierarchy construction and the mutual exclusiveness detection process that discovers the same amount of operation exclusiveness for both descriptions. The "*kim*" benchmark is a good example to illustrate how the proposed priority function can produce better results compared to a traditional one that does not account for the conditional nature of the behavior. In Table III the benchmark is scheduled for various resource constraints using list-scheduling heuristics. The first uses a traditional priority function based on mobility and number of immediate successors, observes only data dependencies to exploit speculative execution, and whenever

Table III. Scheduling for *"kim"* Using a Probabilistic Priority Function

Heuristic	**Resource Constraints**		
	cmp:2 **ALU**: 4	**cmp**:2 **ALU**: 3	**cmp**:2 **ALU**: 2
Classic List Sched.	5/5/4	6/6/6	8/8/7
ours	5/5/4	6/6/5	6/6/6

Resources	**Schedule by approach**					
	Kim	PBS	crit. path	Brewer	Dos	ours
cmp: 0, +: 1, -: 1 cn: 1	8/8/3	-	-	-/5/-	-/5/-	5/5/4
cmp: 0, +: 1, -: 1 cn: 2	6/5/2	9/5/2	8/8/-	-	-	5/5/4
cmp: 0, +: 2, -: 3 cn: 1	-	-	-	-/4/-	-/4/-	4/4/2
cmp: 0, +: 2, -: 3 cn: 3	3/3/2	-	4/4/-	-	-	3/3/2
cmp: 0, +: 2, -: 3 cn: 5	-	4/3/1	-	-	-	3/3/2

(a)

Resources	**Schedule by approach**					
	CVLS	Kim	PBS	Brewer	Dos	ours
cmp: 1, +: 1, -: 1 cn: 1	7/7/5	7/7/4	-	-/7/-	-/7/-	7/7/4
cmp: 1, +: 1, -: 1 cn: 2	-	7/7/3	8/7/3	-	-/7/-	6/6/3
cmp: 1, ALU: 2 cn: 1	-	-	-	-	-	7/7/4
cmp: 1, ALU: 2 cn: 2	-	6/6/3	6/6/3	-	-/6/-	6/6/3

(b)

Resources	**Schedule by approach**	
	ours (cn=1)	ours (cn=2)
cmp: 1, +: 1	4/4/3	4/4/3
cmp: 1, +: 2	4/4/2	3/3/2

(c)

Resources	**Schedule by approach**				
	Kim	Brewer	ADD	Dos	ours
cmp: 2, +: 2, -: 1 cn: 1	8/8/6	-	6/6/5	-	6/6/6
cmp: 1, +: 2, -: 1 cn: 1	-	-/6/-		-/6/-	6/6/6

(d)

Fig. 16. Benchmark comparative scheduling results for: (a) "maha"; (b) "waka"; (c) "jian"; (d) "kim".

control dependencies are observed exploits conditional sharing. The second is our heuristic as described in Section 3.

As resource constraints get tighter, the probabilistic priority function considering candidates for speculative execution afterwards yields better results. For instance, the schedule for two ALUs when only data dependencies are observed results in a total of eight control steps instead of six mainly because all paths are considered equiprobable and speculatively scheduled nodes may displace normally scheduled nodes with higher probability.

4.1 Comparative Results

Finally, the HCDG-based list-scheduling heuristic is compared to other similar heuristics. The obtained results are given in Figure 16 for various resource constraints (one cycle resources) in terms of *total*/*longest* path/*shortest* path numbers of states. Published results of other approaches (i.e., *Kim* [Kim et al. 1991], *CVLS* [Wakabayashi and Yoshimura 1989, 19992], *PBS* [Camposano 1991], *Brewer* [Radivojevic and Brewer 1996], *ADD-FDLS* [Chaiyakul et al. 1992], Dos [Dos Santos et al. 2000]), when available for the particular resource constraints, are also given. Some approaches assume that comparison results

are available at the same control step (e.g., ADD in Figure 16 (d) which results in longer clock cycles. For each benchmark, our results are at least as good as the best previously published results. Chaining (indicated by cn) is also considered and it is worth noting that even without it ($cn = 1$) good results are obtained. The differences in terms of the higher number of states in the shortest paths can be explained by the fact that usually the total number of states is lower than in other approaches and thus resources are better utilized at each step because of conditional sharing and speculative execution.

The HCDG-based scheduling approach effectively exploits all of the existing scheduling optimization techniques enjoying their combined benefits. Both speculative execution and conditional resource sharing are combined in a uniform and consistent framework similarly to the dynamic CVs of Wakabayashi and Yoshimura [1992] and guards in Radivojevic and Brewer [1995, 1996]. Operation chaining during scheduling and multicycle operations which are necessary in a practical scheduling approach as indicated in Bergamaschi et al. [1997] are also supported. Even more, HCDG-based scheduling does not suffer from the effects of syntactic variance at both the mutual exclusiveness detection and scheduling levels, as it is the case with CDFG, CFG-based approaches. The hierarchical control representation permits us to minimize the number of mutual exclusiveness tests and develop probabilistic priority functions accounting for the conditional nature of a design.

With respect to Chaiyakul et al. [1992] and Radivojevic and Brewer [1996], speculative execution is considered only after normally executing nodes have been scheduled. Control dependencies are not observed only if in a control step there are unused resources or resources that can be conditionally shared. In this way the risk of lengthening execution paths by displacing normally executing operations in favor of speculatively executing ones is avoided. Conditional resource sharing is exploited during scheduling and not before. In this way the risk of lengthening of execution paths due to inappropriate conditional resource sharing (i.e., Kim et al. [1991, 1994]) is avoided. In our case, conditional resource sharing considers multiple execution instances simultaneously; operations in different conditional trees can effectively share the same resource and syntactic variance effects, due to differences in conditional nesting (i.e., Wakabayashi and Yoshimura [1989, 1992]), are avoided. With respect to Camposano [1991] and Bergamaschi et al. [1997] the order of statements in the description does not influence the results thanks to the uniform dataflow representation of both control and data dependencies. Complex graph restructuring whose results are not surely positive (i.e., Bergamaschi et al. [1997]) is not needed. Finally, speculative execution and conditional resource-sharing information obtained by the HCDG-based scheduling heuristics can be modeled by transforming the initial HCDG similar to transformations described in Chaiyakul et al. [1992]. This can be quite advantageous in HLS environments where the final design is obtained by many iterations of scheduling allocation.

Our approach was not compared to the most recent results of the Wavesched approach provided in Lakshminarayama et al. [1999] and especially Lakshminarayama et al. [2000] which incorporates speculative execution. This is because currently our approach does not handle loops in addition to the

important loop-related optimizations, and so a fair comparison is difficult. Most of the benchmarks used in Lakshminrayama et al. [1999, 2000] put the emphasis on these aspects. In the future we hope to integrate such ideas into our framework. However, it seems to us that if the loop bodies are not unrolled then the Wavesched approach suffers from some of the major drawbacks of CDFG-based approaches, namely, syntactic variance and limitations in exploiting conditional sharing and speculative execution within the bounds of a single iteration. Our approach is to be able to derive the most compact scheduling for loop bodies using to the greatest possible extent a combination of well-established concepts and in the future apply loop-specific optimization. As an indication, the very elementary loop bodies of the *cordic* and *gcd* benchmarks (used in Lakshminarayama et al. [1999, 2000] resp.) for the same resource constraints are scheduled in three and two control steps, respectively.

5. CONCLUSIONS

The HCDG is a powerful internal design representation and can effectively accommodate design descriptions with dataflow-intensive and/or controlflow-intensive behaviors. Existing HLS heuristics successful for dataflow designs can be easily adapted to HCDG and novel scheduling heuristics for conditional behaviors like the one presented in this work can complete the picture. The hierarchical control representation, mutual exclusiveness identification capabilities, and formal graph transformations lead to our HCDG-based scheduling approach effectively exploiting all of the existing scheduling optimization techniques and enjoying their combined benefits. Both speculative execution and conditional resource sharing are combined in a uniform and consistent framework. Recent work applying a constraint logic programming algorithm on HCDGs [Kuchcinski and Wolinski 2001] indicates that schedules provided by the described heuristic are close to optimal.

The proposed approach and the presented techniques have been implemented in CODESIS, a graphical interactive tool using the HCDG as the internal design representation. This system was developed for both research and educational purposes and presently is under intensive testing. Further work is also needed in order to extend the capabilities of the CODESIS tool with complementary optimization techniques such as the ones proposed in Lakshminarayama et al. [1999] for loops which so far have not been considered. Also, the pipelining (structural and functional) issues need to be addressed; ample hints in the literature exist for doing so. The time-constrained scheduling problem [Park and Kyung 1993] has not been addressed and thus this can also be a topic of future investigation. Finally, for large designs, applying the prescheduling transformations may lead to very large graphs and so investigation of cost functions that would help to selectively apply these transformations may be needed. Further experimentation with larger designs will help us gain more insight into such issues.

It is worth mentioning that in CODESIS it is possible to follow either one of two implementation paths. The first relies on generation from the HCDG of intermediate behavioral source code, such as VHDL, which is accepted as input by

existing high-level synthesis tools. Code generators for C and VHDL already exist in CODESIS. In this context prescheduling transformations, the scheduling process, and postscheduling transformations are considered as a presynthesis optimization process [Kountouris and Wolinski 1999b]. The second implementation path is to perform the entire behavioral synthesis using the HCDG to produce an efficient RTL description (datapath + control FSM). Then using HCDG code generation facilities we can access existing RTL synthesis tools. For this implementation path efficient tools for scheduling, allocation/binding, register allocation, datapath, and control FSM synthesis were developed based on the HCDG. Details of how these activities are performed in CODESIS can be found in Kountouris and Wolinski [2001]. It is important to remember that the control representation by a hierachy helps considerably in optimizing the result of the aforementioned design activities. Mutual exclusiveness information is successfully used for efficient register allocation (another form of conditional resource sharing) and the same approach can be used in sharing other types of resources such as interconnects. In this sense guards and mutual exclusiveness define a generic resource-sharing framework. Finally, in the future we hope to extend CODESIS into a full system-level synthesis framework with a hardware/software codesign orientation.

ACKNOWLEDGMENTS

Thanks go to our colleagues at the EPATR team of IRISA for their fruitful interactions and discussions. Also, many thanks to the anonymous reviewers for their constructive comments which greatly helped us in completing and improving the manuscript.

REFERENCES

AMAGBEGNON, T. P. 1995. Forme canonique arborescente des horloges de SIGNAL. PhD Thesis (December), University of Rennes I.

BEAUVAIS, J. R., GAUTIER, T., LE GUERNIC, P., HOUDEBINE, R., AND RUTTEN, E. 1998. A translation of statecharts into Signal. In *Proceedings of the IEEE International Conferance on Application of Concurrency to System Design* (*CSD'98*, Japan, March), 52–62.

BERGAMASCHI, R. A. 1998. Behavioral synthesis: an overview. IBM Tech. Rep. RC20944.

BERGAMASCHI, R. A., CAMPOSANO, R., AND PAYER, M. 1992. Allocation algorithms based on path analysis. *Integration: VLSI J. 13*, 3 (Sept.), 283–299.

BERGAMASCHI, R. A. AND KUEHLMANN, A. 1993. A system for production use of high-level synthesis. *IEEE Trans. VLSI Syst. 1*, 3, 233–243.

BERGAMASCHI, R. A., RAJE, S., NAIR, I., AND TREVILLYAN, L. 1997. Control-flow versus data-flow based scheduling: Combining both approaches in an adaptive scheduling system. *IEEE Trans. VLSI 5*, 1, 82–100.

BESNARD, L. 1991. Compilation de SIGNAL: Horloges, dependances, environment. PhD Thesis, University of Rennes I.

CAMPOSANO, R. 1991. Path-based scheduling for synthesis. *IEEE Trans. CAD 10*, 1, 85–93.

CARTER, L., SIMON, B., CALDER, B., CARTER, L., AND FERRANTE, J. 1999. Predicated static single assignment. In Proceedings of the *IEEE PACT—International Conferance on Parallel Architectures and Compilation Techniques* (October), 245–255.

CHAIYAKUL, V., GAJSKI, D. D., AND RAMACHANDRAN, L. 1992. Minimizing syntactic variance with assignment decision diagrams. UCI Tech. Rep. ICS-TR-92-34 (April).

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. K., AND ZADECK, F. K. 1989. An efficient method of computing, static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 25–35.

DOS SANTOS, L. C. V. 1997. A method to control compensation code during global scheduling. *Proceedings of the ProRISC CSSP97 Workshop*, 457–464.

DOS SANTOS, L. C. V., HEIJLIGERS, M. J. M, EIJK, C. A. J., VAN EIJNDHOVEN, J. T. J., AND JESS, J. A. G. 2000. A code motion pruning technique for global scheduling. *ACM Trans. Des. Autom. Electron. Syst. 5*, 1, 1–38.

FISHER, J. A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. 30*, 7, (July), 478–490.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco.

GIRKAR, M. AND POLYCHRONOPOULOS, C. 1992. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Par. Dist. Syst.* 166–178.

HUANG, S. H., JEANG, Y. L., HWANG, C. T., HSU, Y. C., AND WANG, J. F. 1993. A tree-based scheduling algorithm for control dominated circuits. In *Proceedings of the ACM/IEEE Design Automation Conference*, 578–582.

JUAN, H.-P., CHAIYAKUL, V., AND GAJSKI, D. D. 1994. Condition graphs for high quality behavioral synthesis. In *Proceedings of* ICCAD'94 (San Jose, Calif.).

KIFLI, A., GOOSSENS, G., AND DE MAN, H. 1995. A unified scheduling model for high-level synthesis and code generation. In *Proceedings of the EDTC'95* (Paris, March), 234–238.

KIM, T., LIU, J. W. S., AND LIU, C. L. 1991. A scheduling algorithm for conditional resource sharing. In *Proceedings of the ICCAD 91*, 84–87.

KIM, T., YONEZAWA, N., LIU, J. W. S., AND LIU, C. L. 1994. A scheduling algorithm for conditional resource sharing—A hierarchical reduction approach. *IEEE Trans. CAD 13*, 4 (April), 425–438.

KOUNTOURIS, A. 1998. Outils pour la validation temporelle et l'optimisation de programmes synchrones. PhD Thesis, University of Rennes 1 (October).

KOUNTOURIS, A. AND WOLINSKI, C. 1999. Hierarchical conditional dependency graphs for mutual exclusiveness identification. In *Proceedings of the VLSI'99* (January), IEEE CS Press, Los Alamitos, Calif.

KOUNTOURIS, A. AND WOLINSKI, C. 1999. High level pre-synthesis optimization steps using ierarchical conditional dependency graphs. In *Proceedings of the 25th Euromicro Conference*, Milano, Italy, (September), IEEE CS Press, Los Alamitos, Calif.

KOUNTOURIS, A. AND WOLINSKI, C. 2001. High-level synthesis using hierarchical conditional dependency graphs in the CODESIS system. *J. Syst. Arch. 47*, 293–313.

KUCHCINSKI, K. AND WOLINSKI, C. 2001. Synthesis of conditional behaviors using hierarchical conditional dependency graphs and constraint logic programming. In *Proceedings of the Euromicro Conference*, Poland, (September), IEEE CS Press. Los Alamitos, Calif.

LAKSHMINARAYAMA, G., KHOURI, K. S., AND JHA, N. K. 1999. Wavesched: A novel scheduling technique for control-flow intensive designs. *IEEE Trans. CAD 18*, 5 (May), 505–523.

LAKSHMINARAYAMA, G., RAGHUNATHAN, A., AND JHA, N. K 2000. Incorporating speculative execution into scheduling of control-flow intensive designs. *IEEE Trans. CAD 19*, 3 (March), 308–324.

LE GUERNIC, P., LE BORGNE, M., GAUTIER, T., AND LE MAIRE, C. 1991. Programming real-time applications with SIGNAL. *Proc. IEEE 79*, 9 (Sept.), 1321–1336.

LI, J. 1998. Timed decision tables and its applications in pre-synthesis and partial synthesis of digital circuits. PhD Thesis, UIUC.

LI, J. AND GUPTA, R. K. 1998. An algorithm to determine mutually exclusive operations in behavioral descriptions. In *Proceedings of the DATE'98* (Paris, February).

LIN, Y.-L. 1997. Recent developments in high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst. 2*, 1 (Jan.), 2–21.

NICOLAU, A. 1985. Percolation scheduling: A parallel compilation technique. Tech. Rep. TR85-678, Cornell University, Computer Science Department (May).

NOVACK, S. AND NICOLAU, A. 1993. Trailblazing: A hierarchical approach to percolation scheduling. In Proceedings of the International Conference on Paralel Processing (2, August), 120–124.

PARK, I. C. AND KYUNG, C. M. 1993. FAMOS: An efficient scheduling algorithm for high-level synthesis. *IEEE Trans. CAD 12*, 10 (Oct.), 1437–1448.

PARKER, A. C., PIZARRO, J. T. AND MLINER, M. 1986. MAHA: A program for data path synthesis. In *Proceedings of the 23rd DAC*, 252–258.

POLYCHRONOPOULOS, C. 1991. The hierarchical task graph and its use in auto-scheduling. In *Proceedings of the ACM International Conference on Supercomputing*, 252–263.

POTASMAN, R., LIS, J., NICOLAU, A., AND GAJSKI, D. 1990. Percolation based synthesis. In *Proceedings of the ACM/IEEE Design Automation Conference*, 444–449.

PURI, R. AND GU, J. 1992. An efficient algorithm for microword length minimization. In *Proceedings of the DAC'92*, 651–656.

RADIVOJEVIC, I. AND BREWER, F. 1995. Analysis of conditional resource sharing using a guard- based control representation. In *Proceedings of the ICCD'95* (October), 434–439.

RADIVOJEVIC, I. AND BREWER, F. 1996. A new symbolic technique for control dependent scheduling. *IEEE Trans. CAD 15*, 1, 45–57.

RIM, M. AND JAIN, R. 1992. Representing conditional branches for high-level synthesis applications. In *Proceedings of the 29th DAC* (June), 106–111.

STOK, L. 1994. Data path synthesis. *Integration: VLSI J. 18*, 1 (December), 1–71.

SYNOPSYS, Behavioral Compiler online documentation.

TSENG, C. J., WEI, R. S., ROTHWEILER, S. G., TONG, M. M., AND BOSE, A. K. 1988. Bridge: A versatile behavioral synthesis system. In *Proceedings of the 25th DAC* (June), 415–420.

TSENG, C. J. AND SIEWIOREK, D. P. 1986. Automated synthesis of data paths on digital systems. *IEEE Trans. CAD 5*, 3 (July), 379–395.

WAKABAYASHI, K. AND YOSHIMURA, T. 1989. A resource sharing and control synthesis method for conditional branches. In *Proceedings of the IEEE ICCAD-89*, 62–65.

WAKABAYASHI, K. AND YOSHIMURA, T. 1992. Global scheduling independent of control dependencies based on condition vectors. In *Proceedings of the 29th DAC*.

Received February 2001; accepted February 2002