The Effects of False Paths in High–Level Synthesis

Reinaldo A. Bergamaschi

IBM Research Division – T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

This paper discusses the effects of false paths and their consequences in scheduling and allocation during high–level synthesis. False paths through the control–flow graph may occur due to sequences of conditional operations. The detection of false paths during scheduling may result in smaller number of states, improved operator sharing and smaller control logic. An heuristic algorithm is presented for the detection and elimination of false paths during path–based scheduling. Results for benchmark examples are presented. For the designs which contained false paths, the percentage of false paths varied from 5% to 83%. A reduction of 15% in the final cell count for one benchmark was obtained by eliminating false paths.

1 Introduction

A common characteristic of most high–level synthesis systems is the lack of consideration to logic synthesis aspects during the high–level synthesis process. Control signals (e.g., multiplexer select, register load) are usually computed at the end of the high–level synthesis steps, by analyzing the conditional operations (in the control–flow graph) leading to the execution of each operation (in the data–flow graph). Scheduling and allocation are performed largely ignoring the relationships among the conditional operations. Limited consideration to the conditional use of operators and registers is given in [1] and [2] respectively, by analyzing operations on different branches of conditional blocks. A general approach has been implemented in [3], which globally determines the conditional usage of operators and registers by analyzing every execution path in the control–flow graph.

Dynamic computation of control conditions, during high–level synthesis, can be used for detecting *false control paths*. The elimination of false paths can lead to smaller number of states, improved operator sharing, and reduced control logic.

This paper presents an algorithm for detection and elimination of false control paths during high–level synthesis. This algorithm was integrated in the path–based scheduling and allocation steps in the HIS system [4]. It constitutes, to the best of the author's knowledge, the first work to consider false paths during high–level synthesis.

Section 2 introduces the concept of false paths in a high–level description and discusses their effect in the synthesis results. Section 3 describes the algorithm for detecting and eliminating false paths. Sections 4 and 5 present results and conclusions respectively.

2 False paths in a high–level description

2.1 Existence of false paths

A high–level description is usually represented by means of a control–flow and/or a data–flow graph. The control–flow graph (CFG) defines the relative sequencing of operations in the high–level description. A path in the CFG is uniquely defined by a first operation, a last operation, and a sequence of conditional branches. Multiple paths result from the presence of conditional operations (e.g., if, case). A path is executed when all its conditions are satisfied in the sequence required [5].

A false path in a high–level description is defined as a sequence of operations which can never be executed under any combination of input values, independently of how these operations are scheduled. False paths in a high–level description can be seen as an extension of the concept of false paths in combinational logic [6] with the time dimension (scheduling) added to it.

The falsehood of a path depends on the conditions being tested along the path. Each condition corresponds to a boolean function which must be expressed in terms of all its primary data dependencies. Primary data dependencies are either primary inputs or variables which were assigned by operations outside the path. Data dependencies depend on the path being considered.

Definition: *A path in the CFG constitutes a false path iff the logical and of all conditions along the path, expressed in terms of their primary data dependencies, is null for all input values.*

If a primary data dependency is also a primary input of the description then the definition above is only valid if it can be guaranteed that the value of the primary input remains the same along the path.

2.2 Effects of false paths in high–level synthesis

During scheduling and allocation, decisions may be taken to balance the usage of hardware resources in order to meet area and/or speed constraints. Therefore, it is important to know in advance (during scheduling) whether or not operations may be executed in parallel. In most systems to date, this information is extracted based on the topology of the CFG (or equivalent representation). Operations in different states or in different branches of a conditional block are mutually exclusive and therefore can share hardware. However, this is not the only condition upon which operations are mutually exclusive. Operations in a false path can also be mapped into the same hardware resources, since

CH3026-2/91/0000/0080/$01.00 © 1991 IEEE

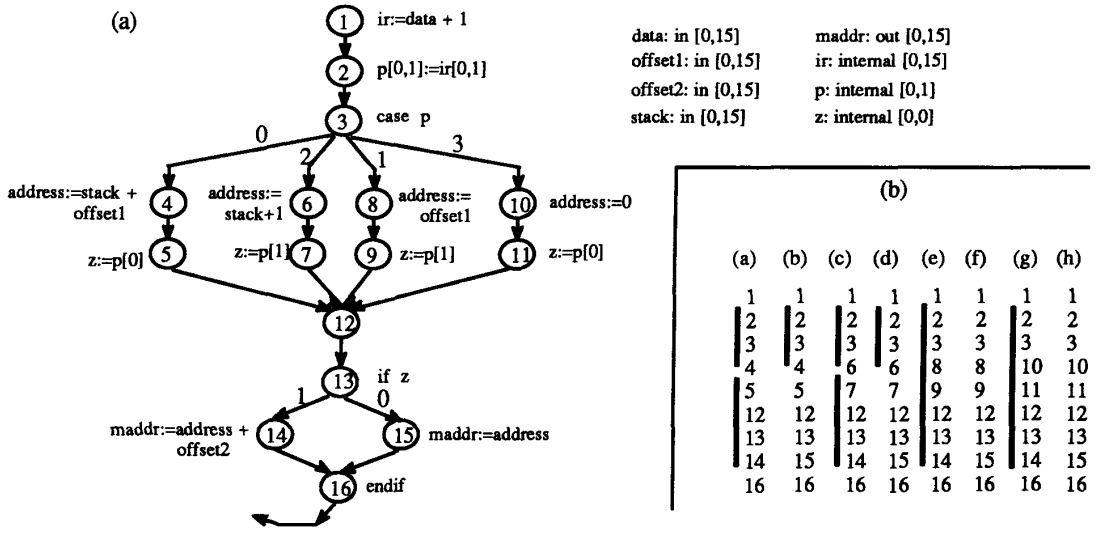

Figure 1: (a) Control–flow graph, (b) paths and constraint intervals for a maximum of one adder

they are never executed at the same time.

In order to analyze the effects of false paths during scheduling and allocation, consider the CFG of a simplified address computation unit shown in figure 1a.

The As–Fast–As–Possible (AFAP) path–based scheduling algorithm used in the HIS system [5], minimizes the number of control steps for all possible paths in the CFG. Each path is scheduled independently, based on constraints applied to it. Constraints are represented by intervals between operations, meaning that a new control step must start within the interval for the constraint to be satisfied. For example, a constraint of one adder results in constraint intervals between any two operations using adders (in the same path). These constraint intervals are shown in figure 1b.

The resulting finite–state machine (FSM), contains four states, as shown in figure 2a. This schedule uses one adder and requires variables *address* and *p* to be stored. During allocation, these two variables can share the same 16–bit register. The condition controlling the execution of each operation in a state is shown in parenthesis beside the operation number (if none is shown, the condition is '1').

Data–flow analysis on the graph in figure 1a reveals that both conditional operations (3 and 13) represent boolean functions with common support variables, and hence could be responsible for false paths. By tracing the data dependencies backwards along all possible paths, it can be seen that variable *p* is a function of bits [1,0] of the output of the addition in operation 1. Similarly, variable *z* is a function of bit [0] of the output of the addition, in paths *a*, *b*, *g* and *h*; and it is a function of bit [1] of the addition, in paths *c*, *d*, *e* and *f*. By checking the values assumed by *p* and *z* along each path, it can be shown that paths *a*, *c*, *f*, and *h*, are indeed false paths, hence they can be disregarded by the scheduler.

The FSM which considers only the non–false paths (*b*, *d*, *e*, and *g*) is shown in figure 2b. It needs only two states, and uses one adder and one 2–bit register (for storing *p*). Variable *address* does not need to be stored.

Another effect of not eliminating false paths during scheduling is that logic paths which are not sensitizable may be present in the final register–transfer level descrip-

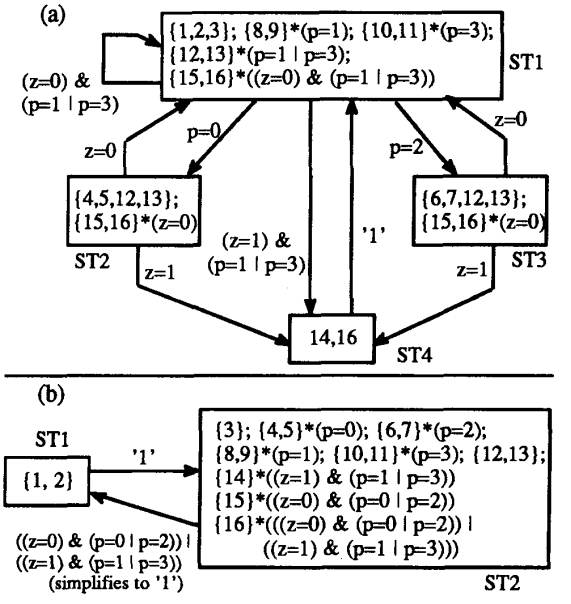

Figure 2: Resulting finite–state machines: (a) FSM1: considering all paths, (b) FSM2: without false paths.

tion. This redundant logic may be eliminated by logic synthesis afterwards. For example, in FSM1 (figure 2a), the transition signal from State 1 to itself, and the control signals of operations 15 and 16 in State 1 can be simplified to null (note that z is a function of p). This is indeed correct since a transition from State 1 to itself could only occur along a false path. Other effects, such as, larger number of states, extra registers and functional units, cannot be eliminated by logic synthesis alone and must be handled by high–level synthesis.

3 Detection and elimination of false paths

In order to determine whether a path is false, the path conditions must be expressed in terms of their primary data dependencies along the path. The expansion of each path condition in terms of its primary data dependencies may be very complex, since any type of operations may be involved. For example, in order to express the boolean condition on node 3 (figure 1) as a function of *data*, it is necessary to represent a 16–bit addition (in operation 1) as a boolean expression.

In the general case, the complexity of the boolean functions representing path conditions may be arbitrarily large. The task of *anding* all conditions will be similarly complex. Binary decision diagrams (bdds) and symbolic simulation techniques can be used for this purpose. However, the generality of the operations involved make it a very complex task even if bdds are used. This makes this technique impractical for its application during scheduling, since it would increase execution time unacceptably.

For this reason, an heuristic algorithm for detecting false paths was developed. Its three main concepts are:

1. Each condition along a path is expressed as a data–flow tree (called *condition tree*). The root node corresponds to the conditional variable itself; internal nodes correspond to the operations on which the variable depends; and leaf nodes are the primary data dependencies in the path. Each condition tree may have as many nodes as operations in the path. Each root node contains a set of values, which are the values assumed by the conditional variable along the path.
2. The falsehood of a path is determined by comparing the condition trees in a path pairwise. If two condition trees are *equivalent* and they assume *disjoint sets of values*, then the path is false; otherwise nothing can be asserted about the falsehood of the path.
3. Two condition trees are *equivalent* if they are isomorphic and corresponding nodes perform the same function. Commutative operations are considered.

The comparison of conditions described in Item 2. is equivalent to perform the *and* of every pair of conditions. If any pairwise *and* is null, then the *and* of all conditions is null and the path is false. The inverse, however, is not true, and therefore this algorithm may not find all false paths. *Equivalence* of condition trees is based on isomorphism, and is also an heuristic, since non–isomorphic trees are always considered not equivalent. The complexity of the above algorithm is $O(n^2)$ on the number of conditional operations in the path.

Figure 3 shows an example of a CFG with conditional operations and condition trees. The set of values assumed by each tree is indicated beside the root node. The values of

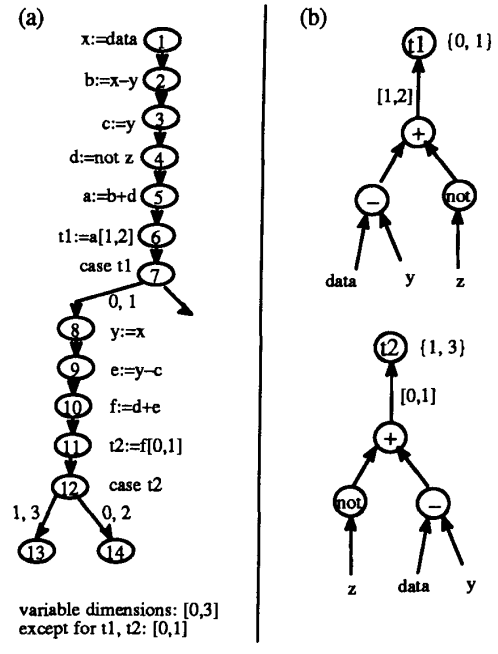

Figure 3: (a) Control–flow graph; (b) condition trees for t1 and t2 in path 1–7–8–12–13

individual bits are taken in consideration. In this example, the trees shown are equivalent, and path 1–7–8–12–13 is a false path since t1[1] and t2[1] (which belong to equivalent trees) assume different values. Path 1–7–8–12–14 is not a false path.

This algorithm was integrated in the As–Fast–As–Possible (AFAP) path–based scheduler in the HIS system [3], [4], [5], [7]. The computation of all paths in the CFG is an integral part of the scheduler. After all paths are determined, the false paths are eliminated and the remaining paths are used for scheduling and allocation.

4 Results

Several benchmark examples [8] were run through HIS with and without false path elimination, for the purpose of comparing execution times and measuring the effect of false paths in the final hardware. HIS generated a register–transfer level description, which was then passed to LSS [9] for logic synthesis. Results are reported in terms of control (paths, states and transitions), data–path (registers, functional units and multiplexers/mux inputs), and final cell count (after LSS – a cell is a basic unit of area).

Only three of the examples contained false paths: ADDRC, M6502 and TX8251. In ADDRC about 83% of all paths were false, compared to 49% in TX8251 and only 5% in M6502. Table 1 shows the scheduling, allocation and logic synthesis results, with and without false path elimination, for these three examples plus the example given in figure 1 (EX1). No hardware constraints were specified for ADDRC, M6502 or TX8251. EX1 had a constraint of one adder. Allocation was performed sharing as much hardware resources as possible.

Table 1 shows that false path elimination was very effective in reducing the final area in examples ADDRC and EX1. In these cases, the elimination of false paths resulted in a different schedule which used less functional units, less registers, and simpler control logic. In the case of TX8251, although there was a significant reduction in the number of paths, the final schedule was very similar to the one without false path elimination. The same was true for the M6502 which had only few paths eliminated. In these two examples, TX8251 and M6502, the data paths for the designs with and without false path elimination were identical. They differed in the initial size of the control logic (before logic synthesis). The designs without false path elimination had much larger control logic; however the difference was due to redundant logic (due to false paths) which was largely eliminated by logic synthesis. As a result the final size of the designs with and without false path elimination was virtually the same, for TX8251 and M6502.

design		paths	states/trans.	regs	fus	mux/inp	cells	Δ%
addrc	n	125	9/19	40	4+	21/75	1903	15%
	fp	21	9/16	40	3+	11/34	1626	
m6502	n	414	106/204	121	6+,1–	55/284	5527	—
	fp	392	106/202	121	6+,1–	55/284	5537	
tx8251	n	766	22/101	13	1–	11/27	1018	—
	fp	392	22/96	13	1–	11/27	1020	
ex1	n	8	4/9	16	1+	5/18	663	42%
	fp	4	2/2	2	1+	3/10	383	

Table 1: Scheduling, allocation and logic synthesis results, for examples with false paths. (n – no false path elimination; fp – with false path elimination.)

design	Scheduling		Allocation	
	n	fp	n	fp
i8251	3.0	3.3	7.1	7.1
rcv8251	2.7	2.9	6.1	6.1
tx8251*	11.9	7.0	6.9	6.4
hunt	0.2	0.2	0.8	0.8
m6502*	16.3	15.3	161.8	140.4
addrc*	3.7	0.4	40.0	1.5
memop	0.1	0.1	0.3	0.3
frisc	1.8	1.9	5.7	5.6
prefetch	0.1	0.1	0.3	0.3
kalman	0.3	0.3	2.9	2.8

Table 2: Execution times for scheduling and allocation, with and without false path elimination, in seconds. (n – no false path elimination. fp – with false path elimination. * Denotes examples with false paths.)

Table 2 gives the execution times for scheduling and allocation, with and without false path elimination, for several benchmark examples. Examples with no false paths showed a small increase in scheduling time and no difference in allocation time, due to false path check. Examples with false paths showed indeed a decrease in scheduling and allocation times, due to the reduction in the number of paths and corresponding reduction of control logic.

The reason for so few examples presenting false paths is due to several factors. Firstly, most of the benchmarks are small and do not contain a large number of conditional operations. Secondly, the presence of false paths depends very much on the way that the input description is written. Data–flow oriented descriptions (such as the Kalman filter) tend to present less (usually none) false paths than control–flow oriented ones.

False path elimination also helps checking the correctness of the input description. Operations which appear only in false paths will not be present in the final FSM, which may indicate an error in the description.

4 Conclusions

This paper discussed the effects of false control paths in high–level synthesis and presented an algorithm for detecting false paths. It was shown that the elimination of false paths during scheduling can result in schedules with smaller number of states and transitions, and implementations with smaller number of registers, functional units, multiplexers and control logic.

The algorithm presented is heuristic and cannot guarantee the detection of all false paths. Nevertheless, it did find all false paths in the small to medium size examples tried (where manually checking of all paths was feasible). In most cases the condition trees are small, with few data dependencies, which increases the probability of a false path being found by the algorithm. The time penalty involved in detecting false paths was practically negligible.

The detection and elimination of false paths during scheduling is an important step towards making the synthesis system independent of how a given functionality is expressed in the input description.

References

[1] P. G. Paulin and J. P. Knight, "Force–Directed Scheduling for the Behavioral Synthesis of ASICs", *IEEE Transactions on Computer–Aided Design*, vol. CAD–8, no. 6, pp. 661–679, June 1989.

[2] F. J. Kurdahi and A. C. Parker, "REAL: a Program for Register Allocation", in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, ACM/IEEE, June 1987.

[3] R. A. Bergamaschi, R. Camposano, and M. Payer, "Data Path Synthesis Using Path Analysis", in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, ACM/IEEE, June 1991.

[4] R. Camposano, R. A. Bergamaschi, C. Haynes, M. Payer and S–M Wu, "The IBM High–Level Synthesis System", in R. Camposano and W. Wolf, editors, *High–Level VLSI Synthesis*, Kluwer Academic Publishers, 1991.

[5] R. Camposano, "Path–Based Scheduling for Synthesis", *IEEE Transactions on Computer–Aided Design*, vol. CAD–10, no. 1, pp. 85–93, January 1991.

[6] P. C. McGeer and R. K. Brayton, *Integrating Functional and Temporal Domains in Logic Design*, Kluwer Academic Publishers, 1991.

[7] R. A. Bergamaschi, R. Camposano, and M. Payer, "Area and Performance Optimizations in Path–Based Scheduling", in *Proceedings of the European Conference on Design Automation*, The Netherlands, February 1991.

[8] "Benchmarks for the Fifth International Workshop on High–Level Synthesis", 1991. Available through EMail at HLSW@ics.uci.edu (InterNet).

[9] J. Darringer, D. Brand, J. V. Gerbi, W. Joyner, and L. Trevillyan, "LSS: A system for production logic synthesis", *IBM Journal of Research and Development*, vol. 28, September 1984.