Exploiting State Equivalence on the Fly while Applying Code Motion and Speculation

Luiz C. V. dos Santos* and Jochen A. G. Jess
Design Automation Section, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{luiz, jess}@ics.ele.tue.nl

Abstract

Emerging design problems are prompting the use of code motion and speculation in high–level synthesis to shorten schedules and meet tight time–constraints. Unfortunately, they may increase the number of states to an extent not always affordable for embedded systems. We propose a new technique that not only leads to less states, but also speeds up scheduling. Equivalent states are predicted and merged while building the finite state machine. Experiments indicate that flexible code motions can be used, since our technique restrains state expansion.

1. Introduction and Related Work

Emerging applications combine intensive data–flow, complex control–flow and tight time constraints [8], creating challenging problems whose solution requires multiple functional units and exploitation of parallelism. Traditionally, the scope of such exploitation is the *basic block* (BB), a straight–line code sequence with no branches, except at its entry and exit points. As the parallelism in a BB is limited, the multiple functional units are poorly utilized. This prompts the use of instruction–level parallelism (ILP) techniques [2] in high–level synthesis (HLS), by moving operations across BB boundaries, which is called *code motion*. Some code motions place instructions ahead of branches, leading to *speculation*. Code motion may require the insertion of copies of operations to preserve semantics. This is known as *compensation code*. On the one hand, code motion and speculation may be vital for meeting a tight time–constraint. On the other hand, compensation code may increase the number of states. Such increase may represent just the price to pay for a shorter schedule, but it can also be due to redundant states. Although redundant states could be removed later on (e.g. during sequential synthesis), their scheduling implies that code motion bookkeeping (a main source of global scheduling overhead) would be uselessly performed many times. Therefore, we devise a method to prevent scheduling redundant states.

In this paper, we propose a new technique that *predicts* state equivalence while the finite state machine (FSM) is built *on the fly*. It guarantees a minimal number of states, given an arbitrary priority encoding. If it is predicted that a state is equivalent to an already scheduled state, it is not scheduled, but merged. The technique works not only as a mechanism for restraining code expansion (as less states are obtained), but it also speeds up scheduling (as less states are actually scheduled). This is the advantage of our technique over methods exploiting state equivalence *afterwards*, since they spend time on scheduling many redundant states.

A survey of ILP techniques, like Trace Scheduling and Percolation Scheduling, can be found in [2]. Code motion is captured by recent scheduling methods [1] [10] [11]. Some HLS methods [7] [13] cope with conditional code. Path–based Scheduling [3] optimizes execution paths as fast as possible, but speculation is not allowed. Speculation is usually addressed for speeding up execution [6] [9]. Little work is reported on code motion for worst–case execution. A survey of state equivalence techniques for sequential synthesis is given in [4]. The criterion for pipelining detection in [1] relies on equivalence classes, yet state equivalence is not addressed. To our knowledge, no other method actually checks state equivalence, while applying code motion and speculation during scheduling. Usually, information on conditional execution is not properly maintained as to be efficiently recovered on the fly. In our method, this is encoded in Boolean form at the beginning, it is updated after each code motion and is efficiently retrieved. Also, built–in scheduler heuristics make it expensive to predict future scheduler decisions. Our approach is free from built–in heuristics, as explained in Section 3.

This paper is organized as follows. Section 2 explains our modeling and Section 3 summarizes our approach. Section 4 formulates our criterion for on–the–fly detection of state equivalence, whose implementation is discussed in Section 5. Experimental results are shown in Section 6 and our conclusions in Section 7.

* On leave from Federal University of Santa Catarina, Florianópolis, Brazil, Computer Science Department (INE). Partially supported by CNPq (Brazil) under fellowship award n. 200283/94–4.

2. Background

Definition 1. A *data flow graph* DFG = (V, E) is a directed graph, where V is the set of nodes, representing operations, and E ⊆ V × V is the set of edges, representing dependences between operations.

An example of DFG is shown in Figure 1b for the description in Figure 1a. Circles represent operations. Pentagons denote either *branch* (B) or *merge* (M) nodes controlled by a a *conditional* (c_1). See [5] for an explanation on DFG semantics. To keep track of code motion, we use a condensation of the DFG, as follows.

Definition 2. A *basic–block control flow graph* BBCG = (U, F) is a directed graph where U is the set of nodes, representing BBs or junctions, and F ⊆ U × U is the set of edges, representing the flow of control.

In building the BBCG, all operations in the DFG enclosed by a pair of branch, merge, input or output nodes are condensed into a BB in the BBCG. All branch (merge) nodes in the DFG controlled by the same conditional become a single branch (merge) node. All inputs are contracted to a single *source* node; all outputs, to a *sink* node. Given the DFG in Figure 1b, its BBCG appears in Figure 1c, where circles represent BBs. A pentagon denotes a *junction*: either a *branch* (B) or a *merge* (M).

The relation between a DFG and its BBCG is kept by means of so–called *links*. A link connects an operation o_n in the DFG with a basic block BB_i in the BBCG. To denote that operation o_n is connected to basic block BB_i by means of link λ, we write $o_n \overset{\lambda}{\Rightarrow} BB_i$. For instance, in Figure 1c, each arrow represents a link.

Definition 3. A *state machine graph* SMG = (S, T) is a directed graph, where S is set of states, and T ⊆ S × S is the set of transitions.

The SMG is a prototype for the state transition diagram of the actual FSM [4]. Given a DFG and a set of resource constraints, different SMGs can be derived. Figures 1d and 1e show alternative SMGs for the DFG in Figure 1b, assuming 1 adder, 1 subtracter and 1 comparator. In deriving the SMG in Figure 1d, exploitation of ILP is limited to BBs, whereas code motion is used for the SMG in Figure 1e. Note that if a time constraint of 4 cycles is specified, the former SMG has to be ruled out.

We say that node v_i *reaches* v_j via p, written $v_i \xrightarrow{p} v_j$, if there is a path p from v_i to v_j. To mean that there is some path from v_i to v_j, we write $v_i \xrightarrow{*} v_j$.

In this paper, we assume a set of resource constraints for the data path. Our goal is to obtain a SMG for the control unit that complies with the design constraints.

3. Our constructive approach

Our approach is sketched in Figure 2. Solutions are encoded by priority encodings Π of the operations in the DFG. An *explorer* creates priority encodings and a *constructor* builds a solution for each Π and evaluates its cost. The explorer uses a local search algorithm to select the solution with lowest cost. While building a solution, the constructor checks properties of conditional execution, which are modeled as Boolean queries and are directed to a so–called *Boolean oracle*.

The constructor consists of a *scheduler* and a so–called *parallelizer*. The parallelizer manages code motion and speculation and assigns operations to states while the SMG is generated on the fly. It appoints a *current state* s_k to be scheduled. The parallelizer keeps in the set A_k all the *available operations* [1] for scheduling in state s_k (ready operations). From A_k, the scheduler selects an operation v_i for executing in state s_k. Then, the parallelizer updates the set A_k accordingly. It also updates the *set of free resources* at state s_k, written as R_k. This interaction proceeds until $R_k = \emptyset$ or $A_k = \emptyset$. After scheduling s_k, *next states* are

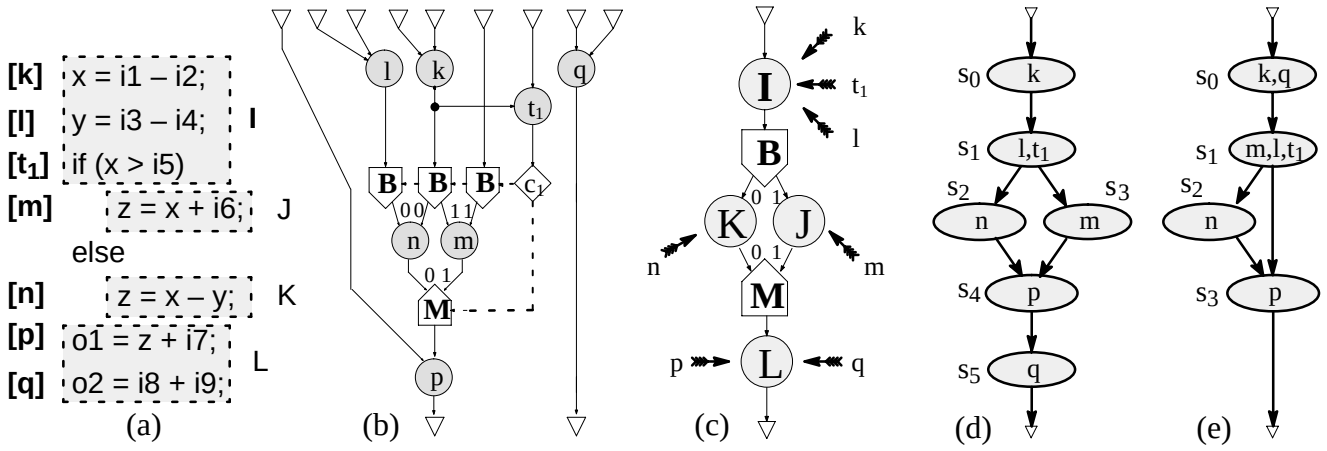

Figure 1. A behavioral description, its DFG, BBCG and resulting SMGs

determined and scheduled. The BBCG is used as a frame for building a SMG. During construction, it is as if each BB is split on the fly into a sequence of successive states. As opposed to most global schedulers [1], our approach removes heuristics *out of the scheduler* (and places them in the explorer). This grants the scheduler a predictability that can be used for state optimization. Every set A_k is ordered by the *same* priority encoding Π. Given a state s_k and an ordered set A_k, the scheduler selects the first operation $v_i \in A_k$ satisfying resource constraints.

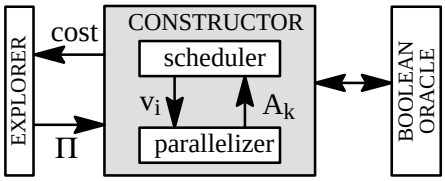

Figure 2. An outline of the approach

4. Exploiting state equivalence

Let's illustrate some notions with Figure 3. To predict state equivalence, we use a Boolean encoding for conditional execution. The key idea is to associate a Boolean variable with each conditional, which is called a *guard* [10], and to define Boolean expressions, which are called *predicates*. For instance, the execution condition of the operations enclosed by a basic block BB_i is represented by a predicate $G(BB_i)$, as illustrated in Figure 3a. Predicates can be efficiently obtained as explained in [12].

Let $\mathcal{C}_i = \{c_1, c_2, \ldots, c_n\}$ be the set of conditionals scheduled in a state s_i (as pointed out by the arrows in the Figure 3b). During execution, a *truth assignment to the conditionals* in $\mathcal{C}_i$ determines their Boolean–valued outcome and is represented by a predicate. Every transition (s_i, s_j) owns an *enabling predicate* $G((s_i, s_j))$, whose value is determined at execution time by a truth assignment to the conditionals in $\mathcal{C}_i$. For instance, $G(s_1, s_4) = \bar{c}_1 \cdot c_2$ in Figure 3b. If $\mathcal{C}_i = \emptyset$, there is a single transition leaving s_i and $G((s_i, s_j)) = 1$. For simplicity, we omit constant predicates in the figures.

4.1. A reformulation for the notion of state equivalence

Classical state equivalence relies on a FSM model. Two states, say s_n and s_m, are equivalent if the output sequences of two instances of the FSM, one initialized in s_n and the other in s_m, match for any input sequence [4]. However, the HLS model for the control unit is more abstract, typically a symbolic description of a FSM. In this model, an output pattern is associated with the set of *operations executing in a given state*, which is called a *bundle*. An input pattern of the FSM is associated with the *predicate* representing a truth assignment to the conditionals scheduled in the previously executed state. Hence, this model requires a more abstract notion of state equivalence. Let OP_n be the bundle of operations in state s_n. Given a path in the SMG, say $p = \langle s_n, s_{n+1}, \ldots, s_{n+k} \rangle$, let $\langle OP_n, OP_{n+1}, \ldots, OP_{n+k} \rangle$ be the sequence of bundles associated with p.

Definition 4. Let $\langle (s_n, s_{n+1}), \ldots, (s_{n+k-1}, s_{n+k}) \rangle$ be a sequence of k transitions starting at state s_n. Given a sequence of predicates $\mathcal{G} = \langle G_1, G_2, \ldots, G_k \rangle$ with $G_i = G((s_n, s_{n+i}))$ and $1 \le i \le k$, the *sequence of bundles induced by* $\mathcal{G}$, written $OP(s_n, G_1, G_2, \ldots, G_k)$, is $\langle OP_n, OP_{n+1}, \ldots OP_{n+k} \rangle$.

In Figure 3b, for instance, the sequence $\langle \bar{c}_1 \cdot \bar{c}_2, 1, c_3 \rangle$ induces the sequence of bundles $\langle OP_1, OP_2, OP_5, OP_8 \rangle$.

Definition 5. States s_n and s_m are *schedule equivalent*, written $s_n \overset{\Phi}{=} s_m$, if and only if the equality $OP(s_n, G_1, G_2, \ldots, G_k) = OP(s_m, G_1, G_2, \ldots, G_k)$ holds for every possible sequence $\langle G_1, G_2, \ldots, G_k \rangle$.

For equivalence, not only the bundles of operations scheduled in states s_n and s_m must coincide, but also *the bundles of every state reachable from them under a same sequence of enabling predicates*. This is illustrated in Figures 4a and 4b. Note that duplication of conditionals has occurred, as a result of code motion. Many states in Figure 4a are equivalent. For instance, note that not only $OP(s_6, \bar{c}_3, 1, 1, 1) = OP(s_{13}, \bar{c}_3, 1, 1, 1)$ holds, but also $OP(s_6, c_3, 1, 1, 1) = OP(s_{13}, c_3, 1, 1, 1)$, i.e. $s_6 \overset{\Phi}{=} s_{13}$. Each shaded state in Figure 4a is redundant and can be merged with its equivalent, as shown in Figure 4b. Our goal is to avoid building a solution like the one in Figure 4a, *without restricting code motion and speculation*. The formulation of state equivalence assumes a completely defined SMG. However, in the course of scheduling some states and transitions are *not yet defined* (after scheduling the current state, the next states are still unscheduled). Therefore, we have to *predict* state equivalence while building the SMG.

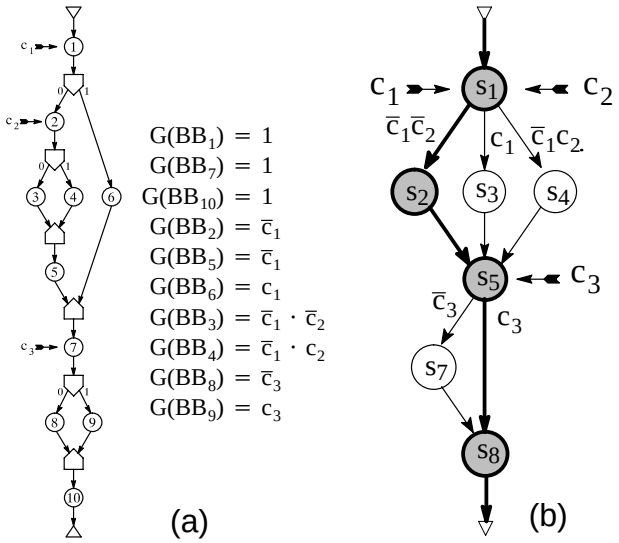

Figure 3. The relation between BBCG and SMG

4.2. On–the–fly detection of state equivalence

Though incompletely defined, a sequence of transitions can be captured by a predicate. In Figures 4c and 4d, let s_n be a state within a basic block BB_i. Suppose that an operation o_z is to be executed in some state, say s_x, reachable from s_n. Hence, o_z must be linked to some BB_k such that $BB_i \xrightarrow{p} BB_k$. Assume that $G(BB_i) = \bar{c}_1 \cdot c_2$ and $G(BB_k) = \bar{c}_1 \cdot c_2 \cdot c_3 \cdot \bar{c}_4$. Since BB_i precedes BB_k on path p, the guards c_3 and c_4 are due to branches occurring after BB_i on path p. This shows that a predicate determining a sequence of transitions from s_n to s_x can be obtained by removing from $G(BB_k)$ the guards in $G(BB_i)$. This is implemented by the smoothing operator [4]. The *smoothing* of a predicate G with respect to guard c, written $\mathcal{S}_c(G)$, is obtained by omitting all the occurrences of c in G. For instance, in Figure 4c, the predicate $\Gamma = c_3 \cdot \bar{c}_4$ is derived by smoothing the guards c_1 and c_2 in $G(BB_k)$. Note that Γ determines the sequence of transitions highlighted in Figure 4d, as formalized below.

Definition 6. A predicate Γ *induces a sequence of transitions* $\langle \mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_k \rangle$, with $\mathcal{T}_i \in T$, iff $\Gamma \cdot G(\mathcal{T}_i)$ is satisfiable for every transition $\mathcal{T}_i$ in the sequence.

Given an already scheduled state s_n within BB_i, an unscheduled state s_m within BB_j, and the pairs (A_n, R_n) and (A_m, R_m), we want to predict if the process of scheduling, starting at s_n and s_m, will result on equivalent sequences of bundles. Our detection of state equivalence relies on the following three properties:

i. Scheduler predictability

The operations scheduled in state s_n depend only on the resource occupation R_n, on the available operations A_n and on a given priority encoding Π. Therefore, if $(A_n, R_n) = (A_m, R_m)$ holds, then $OP_n = OP_m$.

ii. Reachability from available operations

Let $\mathcal{R}(o_y)$ denote the *set of operations reachable from operation* o_y in the DFG excluding branch and merge nodes. Given a state s_n within BB_i, let's find the set of all operations executed on some path from BB_i to the sink. This set, written as $\mathcal{R}_i(A_n)$, can be found by applying the concept of reachability above to each operation available at state s_n, as follows:

$$\mathcal{R}_i(A_n) = \{ o_z \in \bigcup_{o_y \in A_n} \mathcal{R}(o_y) \mid o_z \xrightarrow{\lambda} BB_k \wedge BB_i \xrightarrow{*} BB_k \}.$$

As a consequence, if it is known that $\mathcal{R}_i(A_n) = \mathcal{R}_j(A_m)$, we conclude that the same set of operations is bound to be executed in states reachable either from s_n or s_m. However, this does not guarantee that a given operation is executed on different paths under exactly the same condition, which motivates the ensuing analysis.

iii. Execution under a same sequence of predicates

An operation o_z may be linked to many BBs reachable from BB_i, due to compensation code. To capture the joint effect of all "copies" of o_z, we first find the set of all links emanating from o_z, written $\Lambda(o_z)$, and we select those linked to BBs reachable from BB_i. This subset, written as $\Lambda_i(o_z)$, is obtained as follows:

$$\Lambda_i(o_z) = \{ \lambda \in \Lambda(o_z) \mid o_z \xrightarrow{\lambda} BB_k \wedge BB_i \xrightarrow{*} BB_k \}.$$

The *joint execution predicate* of operation o_z on all paths starting at BB_i, written $G_i(o_z)$, is expressed as:

$$G_i(o_z) = \sum_{\lambda \in \Lambda_i(o_z)} G(\lambda).$$

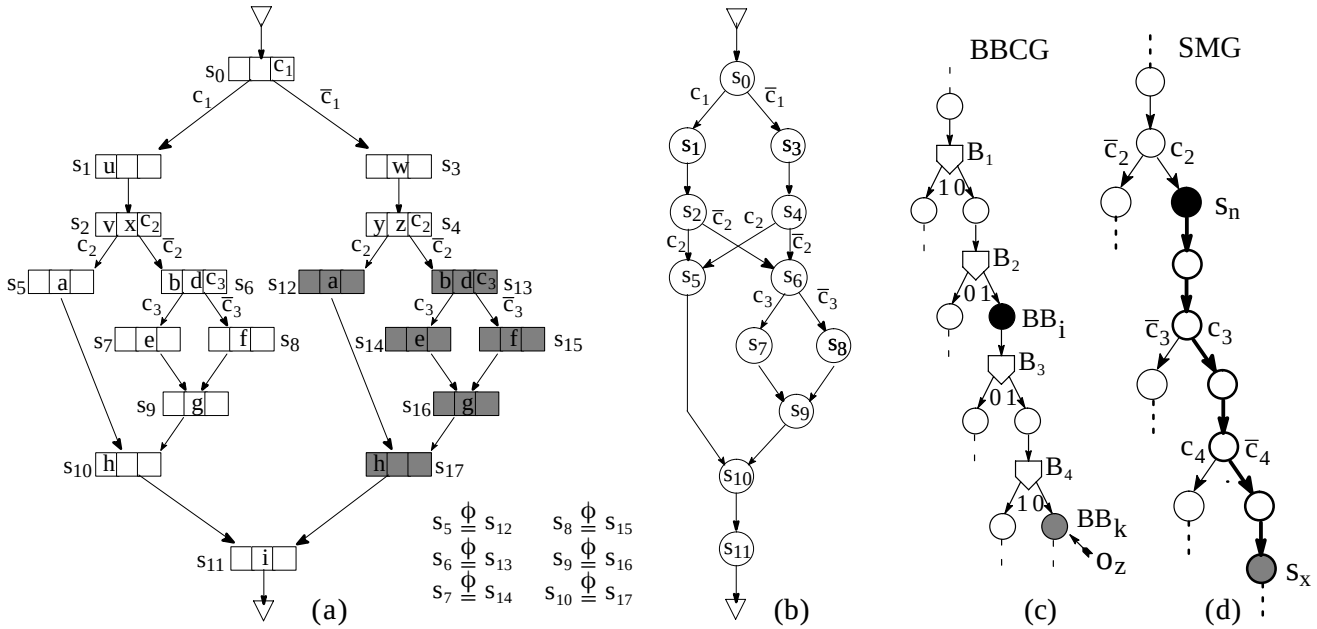

Figure 4. Illustrative examples for state equivalence

Assume that operation o_z will be scheduled in a state s_x reachable from s_n (recall example in Figures 4c and 4d). The predicate inducing a sequence of transitions to s_x, written $\Gamma_i(o_z)$, is obtained by Algorithm 1.

Algorithm 1. Algorithm for determining $\Gamma_i(o_z)$

$\Gamma_i(o_z) = G_i(o_z)$;
foreach $c \in$ support($G(BB_j)$)
  $\Gamma_i(o_z) = \mathcal{Y}_c(\Gamma_i(o_z))$;

Therefore, given the states s_n and s_m within BB_i and BB_j, respectively, if $\Gamma_i(o_z) = \Gamma_j(o_z) = \Gamma$ holds, then o_z will be executed, on *different* paths starting from s_n and s_m, but under sequences of transitions induced by a *same* predicate Γ. Now, we formalize our criterion for on–the–fly detection of state equivalence.

Theorem 1. Let s_n and s_m denote states within basic blocks BB_i and BB_j, respectively. Assume that all availability sets are ordered according to a given priority encoding Π. The equivalence $s_n \overset{\Phi}{=} s_m$ holds for a given Π, iff all the following conditions hold:

- $A_n = A_m$ and $R_n = R_m$, (1.1)
- $\mathcal{R}_i(A_n) = \mathcal{R}_j(A_m) = \mathcal{R}$, (1.2)
- $\forall o_z \in \mathcal{R} : \Gamma_i(o_z) = \Gamma_j(o_z)$. (1.3)

A proof for this theorem can be found in [12].

5. Implementation

Algorithm 2 illustrates an efficient implementation. The pair (A_n, R_n) is stored in a table for every scheduled state s_n. For a given "empty" state s_m about to be scheduled, condition 1.1 is checked via hashing. Only if a hit occurs, the other conditions are tested.

Algorithm 2. Exploiting state equivalence

```
procedure equivalent_state(s_m)
  if (∃s_n ∈ S | (A_n, R_n) = (A_m, R_m))
    if (s_n ≙ s_m)                    /* Theorem 1 */
      return (s_n);
  return(none);
procedure handle_current_state(s_m)
  s_n = equivalent_state(s_m);
  if (s_n ≠ none)
    merge s_m with s_n;
  else
    schedule s_m;
```

Condition 1.2 is checked efficiently by keeping the sets ordered by the priority encoding. Checking condition 1.3 is fast, as it relies on predicates whose number of guards is bounded by the depth of conditional nesting, typically a small fraction of the total number of tests. The technique is part of our constructive approach, which is implemented in the so–called NEAT System. The Boolean oracle currently relies on a BDD package.

6. Experimental results

We performed experiments under largely unrestricted code motion and speculation. A random–generated sequence of priority encodings was used to induce many solutions, from which statistics were derived. This allows us to evaluate the impact of our technique for an *arbitrary priority encoding*. Table 1 compares the quality of the solutions *with* and *without* exploitation of state equivalence for several examples. L_i denotes the mean value for the schedule length of the longest path in the SMG. Both the mean value and the standard deviation (σ) are given for the number of states. The average time (avg time) to build one SMG is given in seconds on a HP9000/735 workstation.

The values of L_i coincided in both cases and for every example (i.e. without exploiting equivalence, we are paying a higher price for the same schedule quality). The shaded columns, indicate that, without exploiting equivalence, the size of the SMG is unpractical for DFGs with complex control flow. To overcome this, most methods either restrict code motion (e.g. by disallowing duplication of tests) or rely on heuristics to alleviate the problem [1]. The last column shows the state expansion without state equivalence. It indicates that restrictions usually imposed on code motion can be relaxed when our technique is applied, since state expansion is controlled by merging equivalent states. Note that σ grows when state equivalence is exploited. This shows that the size of the SMG is actually more sensitive to the priority encoding than we could tell if the technique was not applied. This means that not merging equivalent states during exploration hampers further phases of the design flow (solutions apparently similar during exploration may end up in very different SMG sizes after sequential synthesis). The results also show that our technique accelerates the construction of solutions, since the time spent on equivalence checking is less than the time to schedule all redundant states.

7. Conclusions

Unrestricted code motion may increase the number of states, yet we have shown that it can be supported without inserting redundant states. Results indicate that if a HLS tool is required to make use of flexible code motions to face tight time–constraints, the size of the SMG is unpractical without on–the–fly exploitation of state equivalence. Besides, our technique speeds up scheduling via an efficient state equivalence checking.

Table 1. The impact of on–the fly exploitation of state equivalence

example	nodes DFG	BBs	case	resource constraints					L_i	without			with			exp
										#states		avg time	#states		avg time	
				alu	add	sub	mul	cmp		mean	σ[%]		mean	σ[%]		
waka	46	10	A	0	1	1	0	1	7.8	14.5	4.5	0.06	11.6	7.6	0.06	1.2
[13]			B	2	0	0	0	1	7.9	14.7	5.7		11.8	8.9		1.2
kim1	48	10	B	0	1	1	0	1	8.8	21.8	4.4	0.08	19.8	6.2	0.08	1.1
[7]			C	0	2	1	0	1	6.9	15.7	5.8		14.9	5.2		1.1
rotor	66	10	A	1	0	0	0	0	11.0	35.0	0.0	0.09	21.4	8.5	0.08	1.6
[10]			B	2	0	0	0	0	8.0	21.3	2.1		17.3	2.6		1.2
			C	3	0	0	0	0	7.0	20.0	0.0		14.6	3.3		1.4
			E	1	0	0	2	0	9.8	29.8	1.2		19.5	4.9		1.5
			F	2	0	0	2	0	8.0	24.0	0.0		15.6	3.2		1.5
			G	3	0	0	2	0	8.0	24.0	0.0		15.6	3.2		1.5
s2r	122	22	A	1	0	0	0	0	14.7	127.5	3.2	0.51	71.9	14	0.46	1.8
[10]			B	2	0	0	0	0	9.5	77.0	4.4		59.2	6.9		1.3
			C	3	0	0	0	0	8.9	73.2	4.3		54.2	8.0		1.3
			E	1	0	0	2	0	13.1	95.5	3.8		73.0	6.9		1.3
			F	2	0	0	2	0	10.0	78.9	5.1		58.0	8.9		1.4
			G	3	0	0	2	0	9.4	75.8	5.1		56.9	7.1		1.3
kim2	464	52	A	0	1	1	1	1	59	2406	16	27	495	17	11	4.9
[7]			C	0	1	2	1	1	59	2395	15		476	15		5.0
			D	0	1	1	2	1	58	2317	16		437	20		5.3

References

[1] A. Aiken et al., "Resource–Constrained Software Pipelining", IEEE Trans. Parallel and Distributed Syst., vol. 6(12), pp. 1248–1270, Dec. 1995.

[2] U. Banerjee et al., "Automatic Program Parallelization", Proc. of the IEEE, vol. 81(2), pp. 211–243, Feb. 1993.

[3] R. Bergamaschi et al., "Control–Flow Versus Data–Flow Based Scheduling: Combinining Both Approaches in an Adaptive Scheduling System", IEEE Trans.VLSI Syst., vol. 5(1), pp.82–100, Mar. 1997.

[4] G. DeMicheli, "Synthesis and Optimization of Digital Circuits", Mc Graw–Hill, 1994.

[5] J. Eijndhoven and L. Stok, "A Data Flow Exchange Standard", Proc. Europ. Conf. Design Automation, pp. 193–199, 1992.

[6] U. Holtmann and R. Ernst, "Combining MBP–Speculative Computation and Loop Pipelining in High–Level Synthesis", Proc. European Design and Test Conf., pp.550–555, 1995.

[7] T. Kim et al., "A Scheduling Algorithm for Conditional Resource Sharing, A Hierarchical Reduction Approach", IEEE Trans. CAD, vol. 13(4), pp. 425–438, 1994.

[8] R. P. Kleihorst et al., "MPEG2 Video Encoding in Consumer Electronics", J. VLSI Signal Proc., vol. 17, pp. 241–253, 1997.

[9] G. Lakshminarayana et al., "Incorporating Speculative Execution into Scheduling of Control–flow Intensive Behavioral Descriptions", Proc. Design Automation Conf., 1998.

[10] I.Radivojevic and F.Brewer, "A New Symbolic Technique for Control Dependent Scheduling", IEEE Trans. CAD, vol.15(1), pp. 45–57, 1996.

[11] M. Rim et al.,"Global Scheduling with Code–Motions for High–Level Synthesis Applications", IEEE Trans. VLSI Systems, vol. 3, n. 3, pp. 379–392, Sept. 1995.

[12] L.C.V. dos Santos, "Exploiting instruction–level parallelism : a constructive approach", PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, Nov. 1998.

[13] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors", Proc. Design Automation Conf., pp.112–115, 1992.