Combining code motion and scheduling

Luiz C. V. dos Santos *, J.T.J. van Eijndhoven and J.A.G. Jess
Design Automation Section, Eindhoven University of Technology, The Netherlands

Abstract— **This work addresses a resource-constrained optimization problem which arises in the context of the high-level synthesis of an ASIC or in the code generation for an ASIP. For a given behavioral description containing conditional constructs, scheduling and code motion are combined and encoded in the form of a unified optimization problem. As taking code motion into account may lead to a larger search space, a code-motion pruning technique has been developed. Optimal solutions are kept in the search space and a local-search method is used to seek for potential solutions. We show that our technique can cope with issues like speculative execution and code duplication. Moreover, it can be extended to tackle constraints imposed by the advance choice of a controller, such as pipelined-control delay and limited branch capabilities. For all cases tested so far, our experimental results have reached the best published results.**

Keywords— **high-level synthesis, code generation, scheduling, code motion, speculative execution**

I. Introduction

SCHEDULING is an important problem in the context of high-level synthesis of digital systems and in the domain of code-generation for embedded processors. Scheduling is control-dependent when the behavioral description has conditionals and loops [1].

A straightforward way of addressing control-dependent scheduling is by applying classical scheduling techniques to all operations which execute under the same condition, the so-called *basic-block* (BB).

As such a local approach may lead to poor quality results for control-flow dominated designs, operations can be moved up and down after BB scheduling [2].

Other approaches have focused on scheduling operations directly across BB boundaries [4], [3], [5], [6], [7]. However, an optimal solution might be missed either due to the applied heuristics [3], [5], [6], [7] or due to a fixed order chosen in advance [4].

A symbolic technique was presented to cope with control-dependent scheduling [1]. Instead of generating a single representative solution (like the mentioned methods), the technique encapsulates all feasible schedules in a BDD form. However, this exact method is unlikely to be used in early (more iterative) phases of a design flow.

We have been developing a method where several representative solutions are generated by a solution constructor and are explored by a local-search algorithm [9]. The basic idea is to keep at least one optimal solution in the search space when advanced issues like code motion and speculative execution are taken into account. With high-quality solutions kept in the search space and by using a local-search approach, one can tradeoff *accuracy* and *search time*. As code motion may lead to a larger search space, a code-motion pruning technique is used to reduce search time.

In this paper we will focus on the techniques to support the improved execution model (code motion, speculative execution, effects due to the advance choice of a controller). In order to make the paper self-contained, we revisit in section II the formulation presented in [9] and we give an outline of the approach in section III. Section IV presents the support for the improved execution model. In section V we emphasize on the idea of code-motion pruning. Finally, experimental results are shown in section VI.

II. Problem formulation and representation

Optimization problem: Given a number K of functional units and an acyclic control data flow graph, find a control sequence represented by a state machine graph, in which precedence constraints are satisfied for each functional unit type, such that a cost function C is minimized.

Definition 1: A *control data flow graph* DFG = (U, E) is a directed graph where the nodes represent operations and the edges represent their dependencies.

Definition 2: A *guard* g_k is a boolean variable associated with the control-flow decision of a conditional c_k [1].

Definition 3: A *predicate* is a boolean function on the set of guard variables.

Definition 4: A *basic block control flow graph* BBCG = (V, F) is a directed graph where the nodes represent

* On leave from INE, Federal University of Santa Catarina, Brazil and partially supported by CNPq (Brazil) under fellowship award n. 200283/94-4

BBs and the edges represent the flow of control. An operation initially associated with a given BB may move to another BB; this is called *code motion.*

Definition 5: Each path in the BBCG defines a sequence of BBs which enclose a set of operations of the DFG. This set is called an *execution instance* (EXI).

III. The constructive approach

Figure 1 shows an outline of our approach. Solutions are encoded by a permutation Π of the operations in the DFG. A solution *explorer* handles encoded solutions and uses a permutation-driven solution *constructor* to evaluate their cost. The explorer is based on a local search algorithm [11]. Conditional execution is modeled by means of boolean functions and queries about whether operations execute under the same conditions are directed to a so-called *boolean oracle* (the term was coined in [10]) which allows us to abstract from the way the queries are implemented.

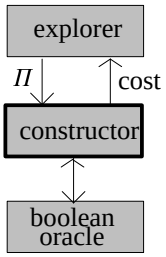


Fig. 1. An outline of the approach.

A. Requirements of a solution

Due to the presence of conditionals in a description, different execution instances are possible, as conditional decisions are data-dependent. A solution is said to be *complete* only if a valid schedule exists for every possible execution instance.

Since conditional resource sharing is affected by the timely availability of test results, a solution is said to be *causal* when the result of a test is not used before the time when it is available.

B. Criterion of optimality

We call *solution space* the set of all feasible solutions. Depending on the scheduling method, not all solutions in the solution space might be reachable. We call *search space* the set of all solutions that can potentially be generated by a given method.

Even though our method can not ensure that an optimal solution will always be reached as a consequence of the local-search formulation, we claim that at least one optimal solution is in the search space for all cost functions which are monotonically increasing in terms of schedule lengths [9].

IV. Support for control-dependent scheduling

A. Initial links

In order to capture the freedom for code motions we introduce the notion of a link. A *link* connects an operation u in the DFG with a BB v in the BBCG. Its interpretation is that u may be executed under the predicate which defines the execution of operations in v. One operation can be linked to several mutually exclusive BBs. Figure 2 illustrates the link concept.

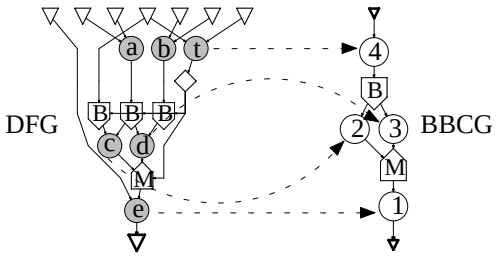


Fig. 2. Links, paths and execution instances

We encode the freedom for code motion by using a set of *initial links* (see [9] for an explanation on how they are obtained). In figure 2 some initial links are shown, but others are omitted for clarity: a is initially linked to both BB2 and BB3 and b only to BB3. Each link points to the latest BB in a given path where the respective operation can still be executed. This means that each operation is **free** to be executed inside any preceding BB on the same path as soon as **data** precedence and resource constraints allow (the only *control* dependency to be satisfied is the need to execute the operation at the latest inside the BB pointed by the initial link). The underlying idea is to traverse the BBCG in topological order trying to schedule operations in traversed BBs. If operation u is given an initial link to BB v and v is reached in the traversal, then u must be scheduled inside it. We say that the assignment of u to BB v is then *compulsory.*

B. Modeling conditional execution

We use predicates to model conditional execution of operations. They are used as attributes of the objects manipulated in our method.

Based on the timely availability of the result of a conditional, we will make a distinction on the predicates. A conditional is said to be *resolved* at a given

time step if its result is available to influence execution at that step. A predicate is said to be *static* and will be labeled as **G** when it represents the execution condition of an operation when all conditionals are assumed as being resolved. A *dynamic* predicate is labeled as Γ and represents the execution condition of an operation when some conditionals may not be resolved at a given time step.

C. Support for code motion and speculative execution

When each operation is executed inside the BB to which it was initially linked, all conditionals are resolved prior to the operations whose execution condition they affect, as a consequence of the BBCG structure. Hence, static predicates represent their execution.

Assume, however, that an operation will move from one BB to another. Let $G_{initial}$ be its predicate prior to code motion. Assume that the operation is scheduled into another BB where a different predicate $G_{current}$ holds. The static predicate after code motion is given by the product: $G = G_{initial}.G_{current}$.

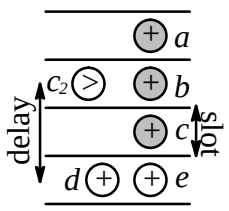
As code motion may lead to speculative execution, G does not represent anymore the effective execution condition and Γ is computed by smoothing all guards whose respective conditionals are unresolved.

Algorithm 1
dynamicPredicate(G, step, slot)
$\Gamma = G$
foreach $g_k \in support(\Gamma)$
  if $end(c_k) + slot > step$
    $\Gamma = smooth(\Gamma, g_k)$

D. Support for pipelined-control delay

When the target controller is pipelined, the result of a conditional may not be available but after a given delay [14]. As a consequence, the conditional is unresolved within a time *slot* after its completion. This effect is considered when evaluating the dynamic predicate, as shown in algorithm 1, where $end(c_k)$ stands for the completion time of conditional c_k

Figure 3 illustrates the effect of pipelined-control delay. Conditional c_2 is associated with guard g_2 and the conditional associated with guard g_1 is considered resolved. A 2-cycle pipelined-control delay is assumed and a single adder is available. In figure 3b the static and dynamic predicates are shown. Only operations d and e can conditionally share the adder. Note also that operations in grey are speculative executed with respect to conditional c_2.



$G_a = G_b = G_d = g_1 g_2$
$G_c = G_e = g_1 g_2'$

$\Gamma_a = \Gamma_b = \Gamma_c = g_1$
$\Gamma_d = g_1 g_2$
$\Gamma_e = g_1 g_2'$

(b)

Fig. 3. Effect of pipelined-control delay

E. Limited branch capability

When simple controllers are targeted (e.g. for the sake of retargetable microcode), state transitions in the underlying FSM are limited by the branch capability of the chosen controller.

$\Pi = [a, b, i, j, k, l, m, n, c_1, c_2]$

(a)

$G_a = 1$	$G_n = g_1'$	$G_i = g_1'$	$G_k = g_2'$
$G_b = 1$	$G_m = g_2'$	$G_j = g_1$	$G_l = g_2$

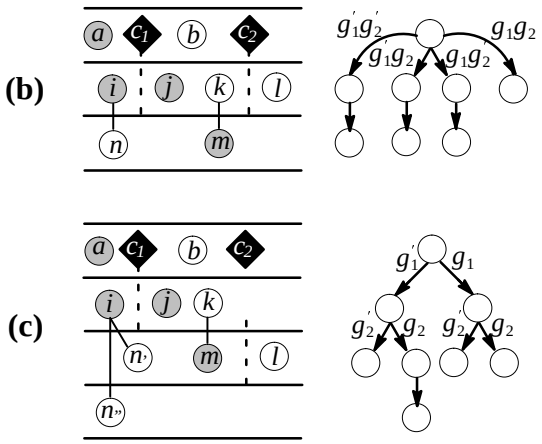


Fig. 4. Effect of limited branch capability

Figure 4 illustrates the problem. In figure 4a we show a permutation Π and the static predicates for the operations in the example. Two different schedules are presented in figures 4b and 4c (1 comparator, 1 "white" resource and 1 "grey" resource). In figure 4c, n' and n'' represent the duplication of operation n.

The schedule in figure 4b implicitly assume a 4-way branch capability, as shown by the state machine graph. For the same Π, if we delay the execution of conditional c_2 of one cycle , we will obtain the schedule in figure 4c, which requires only 2-way branch capability. However, this schedule needs an extra cycle.

As suggested by the example, our method can handle limited branch capabilities during solution construction. If the controller limits the branch capability to a value k, where $k = 2^n$, the constructor will

allow at most n conditionals at the same step. This is similar to the technique presented in [14].

Our treatment for controller-imposed constraints exempts a further rescheduling (like in [15]) into the target controller.

F. Conditional resource sharing

During solution construction, we need to check if two operations can share a same resource under different execution conditions. Let i and j denote two operations. These operations can share a resource at a given step only when the identity $\Gamma_i\Gamma_j \equiv 0$ holds.

V. An Outline of the Method

The algorithm of our solution constructor is summarized in [9], where an illustrative example can also be found. We present here a brief overview and we emphasize on the idea of code-motion pruning.

A. The topological-sorted scheduler engine

The solution constructor takes a permutation Π and generates a solution. Techniques borrowed from the so-called constructive topological sorted scheduler [12] are used, because it has the important property that there always exists a permutation which results in an optimal solution. A schedule is constructed out of a permutation as follows. An operation to be scheduled is selected among ready operations (unscheduled operations whose predecessors are all scheduled) following the order in the permutation. Each selected operation is attempted to be scheduled at the as early as possible time where a free resource is available. Our constructor obeys the criterion below:

Criterion 1: All operations which may execute under the predicate **G** of a BB are scheduled by following the principle of topological-sorted construction.

Note that any ready operation under predicate **G** may be scheduled at the given BB, even if does not belong to it.

B. Traversing the BBCG

The solution constructor follows the flow of tokens in the DFG while the BBCG is traversed in topological order. An operation can be assigned to any traversed BB, as soon as data dependencies and resource constraints allow. If more than one operation satisfies these constraints, an operation will be chosen based on the order in the permutation. Such an assignment is not compulsory as long as the BB to which the operation was initially linked is not reached. As a result, an initial link $u \rightarrow v$ might become a final assignment, but it will be revoked if u succeeds to be scheduled inside any ancestor of v, inducing a code motion.

C. Splitting the linear-time sequence

Our constructor uses a criterion to split the linear-time sequence in order to expose a flow of control:

Criterion 2: Let o be an operation which is ready under the predicate **G** of a BB i, but whose assignment to i is not compulsory. If the schedule of o inside BB i would require the allocation of exactly $\delta = \lceil delay(o) \rceil$ time steps, then operation o is *not* allowed to be scheduled in BB i.

We claim that criterion 2 does not discard any better solution (see proof in [9]).

D. The underlying code-motion pruning

We will show here how the application of criterion 2 represents a code-motion pruning. From an original solution S_m induced by a permutation Π, we will try to construct a better solution S_n for the *same* Π.

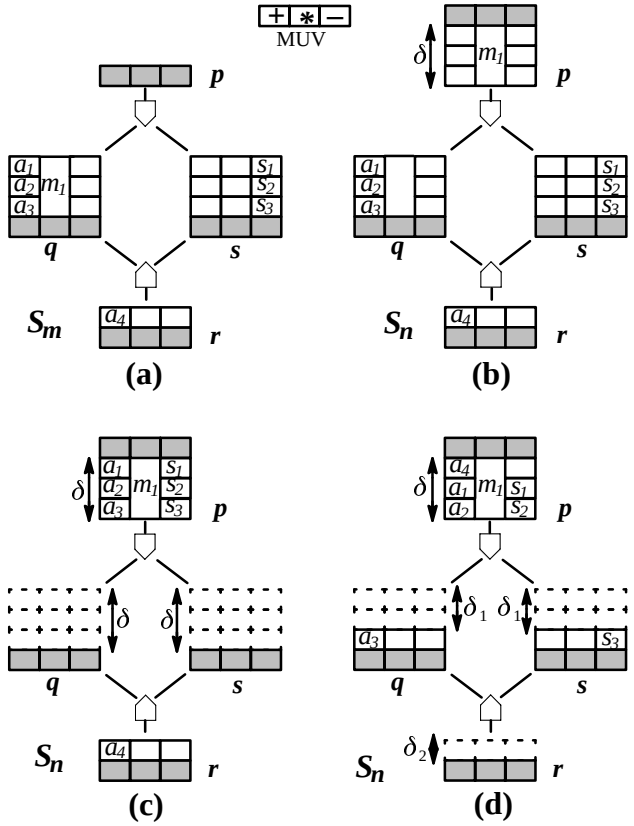


Fig. 5. The idea of code-motion pruning

In figure 5, operations a_1 to a_4 are additions, s_1 to s_3 are subtractions and m_1 is a multiplication (assume one resource of each type). A grey field in the utilization vector (MUV) means either that a resource was

occupied by some other operation or was not occupied due to a data dependency.

In figure 5a we show a solution S_m, generated by our constructor. This means that S_m was generated following both criteria 1 and 2. For example, operation *m1* was not scheduled inside BB p, because it must have been recognized as non-compulsory in p. Also, empty fields in S_m mean that other operations could not be scheduled in the idle modules due to data dependencies. We will show that if m_1 was allowed to boost into BB p, no better result would be reached.

In figure 5b, we start to construct a new solution S_n where *m1* is boosted into BB p and it allocates exactly $\delta = 3$ steps. This will make room for operations from other BBs to move up. We will consider two different scenarios for code motion.

In a first hypothesis, we assume that no operation in BB r can be scheduled in the allocated steps,but operations in BB q and BB s may move up. In figure 5c, the boosted operations cause BBs q and s to shrink by the same number δ of steps allocated in BB p.

Figure 5d illustrates a second hypothesis. We assume that operation a_4 can move up from BB r into the allocated steps. As a consequence, *a3* could not be moved into BB p. Besides, *s1* can not be scheduled at the same step with a_4, because this was not possible in the original solution S_m, what means that the topological-sorted engine must have detected a data dependency between them. As a result, *s3* stays in BB s and path $p \rightarrow s \rightarrow r$ can not be shortened.

Note that in figures 5c and d, even though we have optimistically assumed that the boosted operations have completely freed the steps from which they have moved, no better solution could be reached.

The underlying idea illustrate here is that, instead of allowing any *arbitrary* code motions generated by the permutation-driven scheduler engine, we only allow those which obey criterion 2. This leads to the notion of code-motion pruning. As the application of criterion 2 do not prune any better solutions and the application of criterion 1 guarantees that at least one permutation returns the optimal schedule length, we conclude that this code-motion pruning keeps at least one optimal solution in the search space.

E. Causality and completeness by construction

In path-based approaches [4], completeness is guaranteed by finding a schedule for each control path and overlapping them into a single-representative solution. The method enumerates all control paths, whose number may grow exponentially in the number of conditionals. Causality is guaranteed by preventing an operation from being executed prior to the respective branch test, leading to a limitation of the execution model, as speculative execution is not allowed. Besides, the method may generate infeasible solutions when constraints are imposed by the controller.

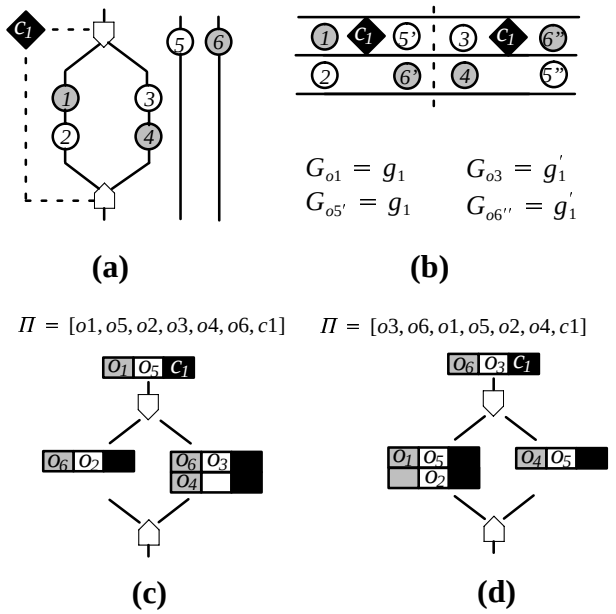


Fig. 6. Causality by construction

The method in [1] has to accommodate the overhead of a trace validation algorithm, required to ensure both completeness and causality. Traces are selected such that they can co-exist, without any conflict, in a same executable solution.

In our method completeness is guaranteed by the traversal of BBs, because all predicates G associated with the BBs are traversed, which makes sure that all possible execution conditions are covered (without the need to enumerate paths). Causality is guaranteed by the *dynamic* evaluation of predicates (see Algorithm 1) during the construction of each solution. To illustrate this fact, we will revisit an example from [1].

Figure 6b shows a solution for the DFG in 6a. A resource of each type is assumed. The solution is complete (both EXIs are scheduled) but it is not causal, because it can not be decided whether o_3 or $o_{5'}$, o_1 or $o_{6''}$ will be executed as conditional c_1 is unresolved at the first step. Note that $G_{o3}G_{o5'} \equiv 0$, but $\Gamma_{o3}\Gamma_{o5'} \neq 0$; and also that $G_{o1}G_{o6''} \equiv 0$, but $\Gamma_{o1}\Gamma_{o6''} \neq 0$.

In figures 6c and d, two solutions were constructed for different permutations. Operations o_3 and o_5 are prevented to be scheduled at the same step (fig. 6c), as well as o_1 and o_6 (fig. 6d). Our *dynamic* evaluation of predicates prevents the construction of non-causal

solutions (like in fig. 6b).

VI. Experimental results

The method has been implemented in the NEAT System [13]. We have been using the BDD package developed by Geert Janssen as boolean oracle and a genetic algorithm as explorer. Search was performed for several randomly chosen seeds. In the following tables, CPU means the average search time in seconds, using an HP9000/735 workstation.

TABLE I
Benchmarks without controller constraints

	kim	maha		parker	
		(a)	(b)	(a)	(b)
add	2	1	2	1	2
sub	1	1	3	1	3
alu	0	0	0	0	0
cmp	1	-	-	-	-
CPU	0.6	3.6	1.8	1.7	0.9
ours	6(5.75)	5(3.31)	4(2.25)	5(3.31)	4(2.00)
ST[1]	6(5.75)	5(3.31)	4(2.25)	-	4(2.13)
TBS[5]	-	5(3.31)	-	-	-
CVLS[3]	6(5.75)	5(3.31)	4(2.38)	5(3.31)	4(2.38)
HRA[8]	7(6.25)	8(4.62)	-	-	-
PBS[4]	-	5(-)	-	-	-

TABLE II
Benchmarks with pipelined-control delay

	rotor							
	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
alu	1	2	3	4	1	2	3	4
mul	0	0	0	0	2	2	2	2
lat	12	7	7	6	10	8	8	8
CPU	0.4	3.3	0.4	0.9	0.6	0.7	0.1	0.1
alu: 1-cycle ALU; mul: 2-cycle pipel. multiplier 1 single-port look-up table; pip.-control delay = 2 cycles speculative execution allowed								

In table I we compare our results with other methods without constraints imposed by the controller. At the top of the table we show the resource constraints for each benchmark. Our results are shown at the middle and results collected from other methods at the bottom. For each result, the schedule length of the longest control path is shown and the average schedule length (assuming equal branch probabilities) is indicated between parenthesis. Note that our method can reach the best published results.

In table II we show our results for benchmarks with pipelined-control delay. Our approach can reach the same latencies obtained by the exact method in [1]. Note that, even though our local-search method can not guarantee optimality, optimal solutions were reached for all cases within competitive CPU times.

VII. Conclusions

These preliminary results suggest that our techniques successfully support the improved execution model when issues like code motion and speculative execution are dealt with during scheduling. Besides, they show that constraints imposed by the controller can be tackled by the constructive approach. As future work, the treatment for loops is to be addressed.

Acknowledgments

The authors would like to acknowledge the suggestions of Marc Heijligers and Koen van Eijk.

References

[1] I. Radivojevic and F. Brewer, "A New Symbolic Technique for Control-Dependent Scheduling," *IEEE Trans. on CAD*, vol. 15, n. 1, pp. 45-57, Jan. 1996.

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

[3] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors," in *Proc. ACM/IEEE DAC*, pp. 112-115, 1992.

[4] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, n. 1, pp. 85-93, Jan. 1991.

[5] S. Huang et al., "A tree-based scheduling algorithm for control dominated circuits," in *Proc. ACM/IEEE DAC*, pp. 578-582, 1993.

[6] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. on Computers*, vol. C-30, n. 7, pp. 478-490 , Jul. 1981.

[7] R. Potasman et al., "Percolation Based Synthesis," in *Proc. ACM/IEEE DAC*, pp. 444-449, 1990.

[8] T. Kim et al., "A Scheduling Algorithm for Conditional Resource Sharing – A Hierarchical Reduction Approach," *IEEE Trans. on CAD*, vol. 13, n. 4, pp. 425-438, Apr. 1994.

[9] L.C.V. dos Santos et al., "A Constructive Method for Exploiting Code Motion," in *Proc. ACM/IEEE ISSS*, 1996.

[10] M. Berkelaar and L. van Ginneken, "Efficient Orthonormality Testing for Synthesis with Pass-Transistor Selectors," in *Proc. ACM/IEEE ICCAD*, pp. 256-263, 1995.

[11] C. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity,* Prentice Hall, 1992.

[12] M. Heijligers and J. Jess, "High-Level Synthesis Scheduling and Allocation using Genetic Algorithms based on Constructive Topological Scheduling Techniques," in *Proc. International Conference on Evolutive Computation*, 1994.

[13] M. Heijligers et al., "NEAT: an Object Oriented High Level Synthesis Interface," in *Proc. IEEE ISCAS*, 1994.

[14] A. Kifli et al. "A Unified Scheduling Model for High-Level Synthesis and Code Generation," in *Proc. European Design and Test Conference*, pp. 234-238, 1995.

[15] S.-Z. Lin et al., "Efficient Microcode Arrangement and Controller Synthesis for Application Specific Integrated Circuits," in *Proc. ACM/IEEE ICCAD*, pp. 38-41, 1991.