Using Speculative Computation and Parallelizing techniques to improve Scheduling of Control based Designs

Roberto Cordone[‡] Fabrizio Ferrandi[#] Gianluca Palermo[#]
Marco D. Santambrogio[#] Donatella Sciuto[#]

[‡]Università Statale di Milano - DTI
via Bramante, 65
26013, Crema, ITALY
Tel: +39-0373-898-054
Fax: +39-0373-898-010
e-mail: cordone@dti.unimi.it

[#]Politecnico di Milano - DEI
pza Leonardo da Vinci, 32
20133, MILANO, ITALY
Tel: +39-02-2399-3479
Fax: +39-02-2399-3411
e-mail: {ferrandi—gpalermo—santambr—sciuto}@elet.polimi.it

November 18, 2005

ABSTRACT

Recent research results have seen the application of parallelizing techniques to high-level synthesis. In particular, the effect of speculative code transformations on mixed control-data flow designs has demonstrated effective results on schedule lengths. In this paper we first analyze the use of the control and data dependence graph as an intermediate representation that provides the possibility of extracting the maximum parallelism. Then we analyze the scheduling problem by formulating an approach based on Integer Linear Programming (ILP) to minimize the number of control steps given the amount of resources. We improve the already proposed ILP scheduling approaches by introducing a new conditional resource sharing constraint which is then extended to the case of speculative computation. The ILP formulation has been solved by using a Branch and Cut framework which provides better results than standard branch and bound techniques.

I. INTRODUCTION

Today high-level synthesis systems need to deal with designs much more complex than a few years ago. Synthesis results were improved by applying standard optimizations such as re-timing and algebraic transformations, as shown in [21], while nowadays speculative code motion techniques demonstrate their effectiveness on scheduling lengths.

Those *new* techniques are well known in the software compilers area [7, 5], and their application to the high level synthesis problem has been proposed by Santos et al. [4] and Rim et al. [23]. Several works such as the Waveschedule approach [12] have introduced the speculative execution as an efficient method to achieve the goal of the minimization of the expected number of cycles. With the exception of [22], no exact methods provide support to speculation of control-dependent specifications. More recently, Gupta et al.[10] have defined a methodology based on code motion techniques developed for parallelizing compilers for high-level synthesis of C-based specifications.

Starting from previous works on parallelizing compilers [8, 9], this paper presents a methodology that extracts from the control and data flow graph of a sequential program a data structure that exposes the parallelism inherent in the specification. The analysis of this data structure commonly used by parallelizing compilers, represents the starting point for the definition of a scheduling technique. In this paper we do not address the scheduling of specifications with loop control structures. Therefore, the proposed algorithm works on specifications obtained by removing all feedback control edges.

We analyze the problem by formulating an approach based on Integer Linear Programming (ILP) to minimize the number of control steps given the amount of resources. We improve the already proposed ILP scheduling approaches by introducing a new conditional resource sharing constraint which is then extended to the case of speculative computation.

Section II presents a compared analysis of the proposed data structure with respect to the state of the art for speculative code transformations on mixed control-data flow designs. Section III describes the integer linear programming (ILP) model that computes a scheduling taking into account the speculative computation issue. The ILP has been solved by a Branch and Cut framework [15]. Section IV presents the results obtained by the proposed approach, while section V concludes by giving an overview of the future directions of the proposed approach.

II. INTERMEDIATE REPRESENTATION

Language based specifications are usually translated into intermediate representations to efficiently manage and analyze the design specification. Several types of intermediate representations have been proposed in literature, each one targeting different types of applications: data flow graph, control flow graph, hierarchical task graph (HTG) [8].

HTGs have been defined as intermediate parallel program representations that encapsulate minimal data and control dependences, and can be used to extract and exploit functional and task-level parallelism. In particular, the hierarchical task graph, as defined in [8], is a directed graph HTG whose vertices can be: simple, representing a task with no subtasks,

0-7803-9451-8/06/$20.00

compound, representing a task that consists of other tasks in an HTG (e.g., higher level structures such as subroutines or loops), loop, representing a task that is a loop whose iteration body is an HTG.

The hierarchical task graph can be extracted from the control flow graph of a sequential program, by identifying the edges through data and control dependences analysis [8, 9].

Let us first consider *control dependences*. A node B is control dependent on A if A can control whether or not B will be executed. Ferrante in [6] defines how control dependences can be identified:

A node *B* is control dependent on *A* if, and only if, *A* is not post-dominated by *B* in the CFG, and there exists a directed path from *A* to *B* in the CFG such that every node other than *A* on the path is postdominated by *B*.

Postdominance [14] is the relation defined as follows: in a directed graph with a distinguished node *STOP*, a node *V* is *postdominated* by another node *W* if, and only if, every directed path from *V* to *STOP* contains *W*.

On the other hand, we can define *data dependence* edges in this way: a node B is data dependent from node A when a data transfer from A towards B exists. They can be further subdivided into three main types: flow dependences (*RAW* dependences), anti dependences (*WAR* dependences) and output dependences (*WAW* dependences).

In [8, 9] control and data analysis are used to define the notion of *precedence* giving some conditions on when a node precedes another, with the aim of maximizing the overall parallelism. Moreover, the precedence notion can be used during the scheduling of the operations since it only considers the actual constraints on the execution order of the operations whatever is the considered granularity (i.e, instruction, functional or task level parallelism).

Gupta et al. [10] reconsider these works on parallelism extraction by using HTG as intermediate representation for high-level synthesis. In particular, they exploit the structural nature of the HTG to perform the scheduling of the operations with code motion and speculation. They consider a particular level of granularity, basic-block, and they slightly modify the definition of the compound node by adding to that node the following control structures: if-then-else, switch-case and sequence of HTGs. With these modifications they loose the power of the control dependence graph, the edges between the nodes are the same of the control flow graph (CFG), but they gain the ability to perform code motion transformations that improve the synthesis results in control intensive designs.

To better understand the differences between control flow graph and its corresponding control dependence graph (CDG), let us consider the example reported in Figure 1, assuming that no data dependences are present. Note that, the CFG imposes more constraints on the order of nodes than the CDG. In fact, without data dependences, the edges of the CFG A-E, B-E, C-D and D-E do not express true precedences between the operations of the specification.

[10] defines several transformations which can be classified into the following four types: *Across Hierarchical Blocks*, *Speculation*, *Reverse Speculation* and *Conditional Speculation*. *Across Hierarchical Blocks*: movement of operations across entire hierarchical blocks, *Speculation*: unconditional

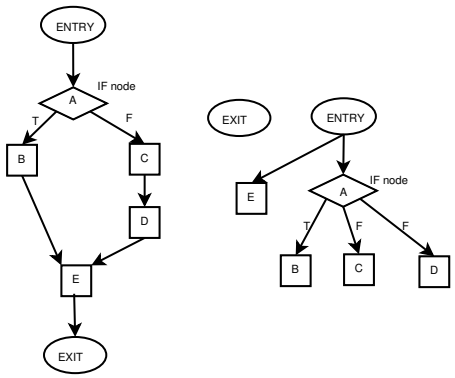

Fig. 1. Comparing control flow graph (a) and control dependence graph (b).

execution of operations that were originally supposed to have executed conditionally, *Reverse Speculation*: where operations before conditionals are moved into subsequent conditional blocks and executed conditionally, *Conditional Speculation*: in which an operation is moved up and duplicated into preceding conditional branches and executed conditionally.

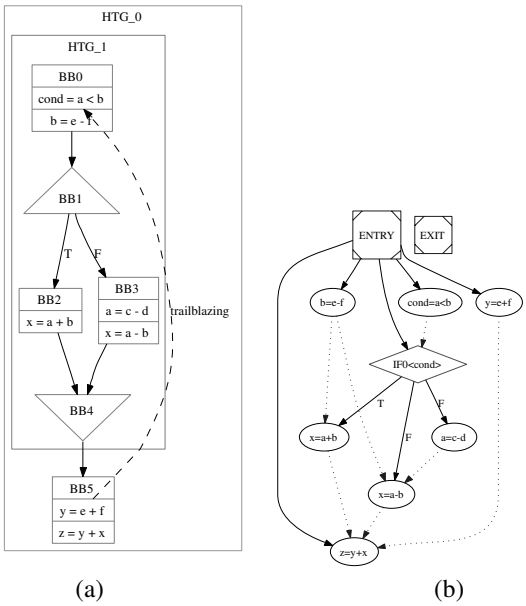

Fig. 2. (a) Code transformation of type 1, starting from the HTG of Figure 8 of [10]. (b) Corresponding control and data dependences graph.

The first transformation considered (i.e., code transformation of type 1) moves blocks following the Trailblazing code motion technique [18] across the nodes of the HTG. Consider for example the operation $y = e + f$ of basic block BB5 reported in Figure 2(a). This operation does not have any data dependence with any node of the if-HTG node, therefore exploiting the hierarchy of the HTG it can be easily moved from BB5 to BB0.

Let us now analyze the same example but considering a different data structure. Starting from the control flow graph we build the CDG and from the flow dependences of the specification, we build the data dependence graph (DDG). The graph built by joining the CDG and the DDG for the example of Fig-

ure 2(a) is reported in Figure 2(b). On this graph, the identification of the control step of the operation $y = e + f$ does not require any move across the hierarchy. The graph imposes only two precedence constraints: one specifying that the node must be executed after the *ENTRY* node and the other requiring that the node must be executed before the $z = y + x$ operation. As shown by [10] anti and output data dependences are required to correctly build the data-path after the scheduling step. [10] performs dynamic variable renaming during the scheduling. We perform this step after the scheduling has been performed. Therefore, if the operation $b = e - f$ has been scheduled before the operation $cond = a < b$, the anti dependence edge will require a renaming of variable b.

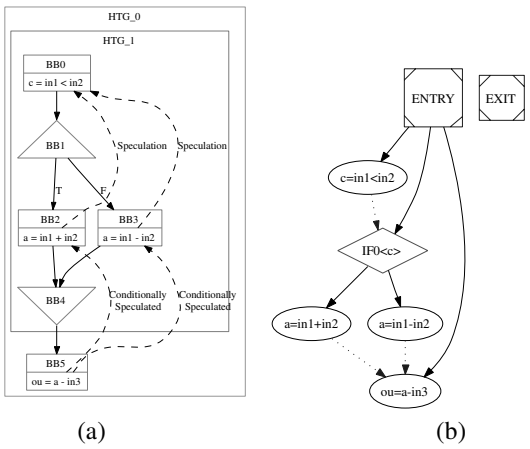

Fig. 3. (a) Code transformation of type 2 and 4, starting from the HTG of Figure 6 of [10]. (b) Corresponding control and data dependences graph.

Speculation (code transformation of type 2) is shown through the example of Figure 3. The HTG based speculation solves the problem of executing an operation before the branch condition by performing a transformation and then the scheduling of the graph. In our approach, we remove the control edges from the graph and then we change the scheduling algorithm with respect to [3]. In particular, if an operation is scheduled before the branch condition no sharing of resources is allowed, while a resource sharing can be exploited if the operation is scheduled after the branch condition. Next section details the ILP-formulation of the resource sharing and of the speculation constraint. Figure 3(a) shows also how the transformation of type 4 can be performed on the HTG. The graph corresponding to the combined CDG and DDG of Figure 3(b) imposes only relative constraints on the operation $ou = a - in3$. Therefore, if speculation moves the operations $a = in1 + in2$ and $a = in1 - in2$ one step earlier, the same can be done on operation $ou = a - in3$. Similar conclusions can be drawn also for reverse speculation. Unfortunately, there are some cases in which parallel execution of an operation does not give the same result as performing conditional or reverse speculation.

In general, transformations of type 3 and 4 may require CFG transformations not easily manageable by an ILP-formulation or by a standard scheduling algorithm. Therefore the methodology proposed in this paper performs a scheduling on the CDG+DDG as extracted from the sequential specification performing code transformations of type 1 and 2. In any case the proposed methodology can take advantage of transformations of type 3 and 4 by coupling the proposed scheduling approach with a transformation toolbox as the one proposed in [10].

III. Scheduling model

This section presents an integer linear programming model for the scheduling problem. Gebotys *et al.* in [3] analyzed the classical ILP model of the scheduling problem, which consists of assignment, precedence and capacity constraints. They also developed a family of additional constraints that, though redundant for the ILP formulation, remove a large subset of fractional solutions when the integrality constraints on the decision variables are relaxed. As a consequence, the continuous relaxation provides a much tighter bound and a nearly integer, if not even integer, solution. This information can be exploited by an ILP solver to compute an optimal solution in shorter time. Next section presents a new way to express the capacity constraints in order to better support the scheduling of control intensive designs. In fact, the new formulation also allows the transformation of type 2.

A code is modeled as a directed acyclic graph, whose nodes represent single operations (or blocks of operations), while the arcs represent precedence constraints due to data or control dependences between nodes. A *branching block* is defined as a condition statement and a number of alternative paths P, one of which is performed, according to the outcome of the condition statement. Each path is a set of code operations and, possibly, branching blocks.

Since only one of the alternative paths actually needs to be followed (based on the outcome of the condition statement), operations belonging to different alternative paths can be assigned to the same functional unit in a given control step. Notice that the operations included in a branching block could also be performed before the condition statement, if no precedence constraint forbids it. In that case, however, no pair of operations can share the same functional unit in a control step.

Let I denote the set of all functional unit types available, K the set of operations and J the set of control steps available (from 0 to the length of a heuristic schedule). L_{ik} is the number of control steps that operation k, when mapped on a functional unit of type i, takes to return ready to accept successive data inputs after a previous execution. C_{ik} is the number of control steps that operation k, when mapped on a functional unit of type i, needs to be executed. Note that $L_{ik} \leq C_{ik}$. N_i is the number of functional units of type i available.

The Boolean variables x_{ijk} model the assignment of code operations to control steps and functional units: when $x_{ijk} = 1$ operation k starts executing at control step j and it is assigned to a functional unit of type i; otherwise $x_{ijk} = 0$. All variables x_{ijk} concerning functional units or control steps incompatible with operation i are undefined.

The integer variables z_{ijB} provide the number of resources of type i occupied in the control step j by branching block B. The integer variable w is the makespan, that is the last control step in which a code operation is performed.

The aim of the scheduling problem is to minimize the

makespan:

$$\min(w)$$

subject to

$$w \geq \sum_{ij} (j + C_{k,i} - 1) x_{i,j,k} \qquad k \in K$$

Each operation is assigned to a specific control step and functional unit type:

$$\sum_{ij} x_{ijk} = 1 \qquad k \in K$$

For each precedence relation $k \prec k'$, operation k cannot be scheduled after operation k'.

$$\sum_{i} \left(\sum_{j \leq j_c} x_{ijk'} + \sum_{j \geq j_c - C_{ik} + 1} x_{ijk} \right) \leq 1$$

where

$$k \prec k', j_c \in \Gamma_{kk'}$$

The first sum states whether operation k' starts before step j_c, while the second sum states whether operation k ends after it. These two events are mutually exclusive. The condition needs to be checked only in the time interval $\Gamma_{kk'}$ in which both events are feasible:

$$\Gamma_{kk'} = [\mathcal{L}_{kk'}; \mathcal{R}_{kk'}]$$

where

$$\mathcal{L}_{kk'} = \max \left(\text{asap}\left(k'\right), \text{asap}\left(k\right) + \min_i C_{ik} - 1 \right)$$

$$\mathcal{R}_{kk'} = \min \left(\text{alap}\left(k'\right), \text{alap}\left(k\right) + \max_i C_{ik} - 1 \right)$$

As in [3], the elementary precedence constraints are combined through a node packing approach, in order to restrict the search space.

The maximum number of functional units used by the whole specification is known:

$$z_{ijB_0} \leq N_i \qquad i \in I, j \in J$$

For each branching block B, each alternative path P of B employs at most z_{ijB} functional units of type i in control step j.

$$\sum_{k \in P} \sum_{j - L_{ki} + 1 \leq j' \leq j} x_{ij'k} + \sum_{B' \in P} z_{ijB'} \leq z_{ijB} \qquad i \in I, j \in J, P \in B, B \in \mathcal{B}$$

where $\mathcal{B}$ is the set of all branching blocks, and the sum over j' takes care of the fact that operation k could occupy functional unit i in control step j, even if it starts before, due to its latency.

Previous ILP approaches ([3]) support conditional branches descriptions generating a similar capacity constraint for each set of mutually exclusive code operations or code operations from each possible path generated by conditional branches. Since the proposed constraint is local to the branching block B, the number of constraints required by the model is not exponential but linear in the number of conditional branches.

The previous capacity constraints operate separately on each alternative path in block B. However, if the operations considered are performed before the condition statement which defines block B, they cannot share the same resources. Therefore, in that case the left-hand-side term must be summed over all paths:

$$\sum_{P \in B} \left(\sum_{k \in P} \sum_{j'=j}^{j - L_{ki} + 1} x_{ij'k} + \sum_{B' \in P} z_{ijB'} \right) \leq z_{ijB} + M \left(1 - \sum_{i' \in I_B} \sum_{j' \geq j - C_{k_B i'} + 1} x_{ij'k_B} \right)$$

$$i \in I, j \in J, B \in \mathcal{B} \setminus \{B_0\}$$

where k_B is the condition statement associated with block B and I_B the subset of unit types which can perform k_B. The sum on the right hand side states whether the condition statement terminates after control step j or not. In the second case, M is a constant large enough to make the constraint redundant: a sufficient value is $M = N_i$.

IV. Experimental Results

The approach presented in this paper has been implemented in a high-level synthesis framework called *PandA*. The framework takes as input C-code and generates optimized scheduled code. *PandA* uses as front-end a customized interface to the GNU *GCC* compiler [1]. Starting from version 3.4, *GCC* provides the possibility of dumping on file the syntax tree structure representing the compiled source code. The combined CDG+DDG data structure is built starting from this syntax tree structure. The use of *GCC* allows the introduction of several compiler optimization techniques into a high-level synthesis framework, such as loop unrolling, constant propagation, dead code elimination, common subexpression elimination, etc. Moreover, the GCC front-end provides an internal representation (i.e., GIMPLE [16]) which is a language-independent tree representation, thus allowing future partial support of languages such as C++ and Java. The ILP formulation has been solved by using a Branch and Cut framework. Branch and cut is a refinement of the standard linear programming based branch and bound approach[17]. The branch and cut approach, with respect to the branch and bound technique, looks for linear inequalities which are violated by the current fractional optimal solution but are respected by all feasible integer solutions of the problem. By adding these inequalities (named cuts or valid inequalities) the continuous relaxation achieves a tight bound and a less fractional solution. There are several standard techniques to generate valid inequalities, both for general ILPs and for specific families of problems. The node packing approach of Gebotys is one of the latter. The open source package *COIN-OR* [15] provides a set of tools among which an ILP solver with the capability of generating the most important families of valid inequalities. Those which are proved effective to solve the scheduling problem are the *Probing*, *Gomory* and *Clique* inequalities.

The validation of the presented approach has been performed using the results obtained by the *Spark* [10] framework as a comparison. All the computational times reported in the result tables refer to a 1.7GHz Pentium IV Linux Workstation.

To produce the experimental results, we have chosen a set of 6 well known standard benchmarks for the problem of high

level synthesis. A first subset is composed of small benchmarks derived from [19] (*sehwa*), [20] (*maha*) and [11] (*kim*), while the second one is composed of a set of multimedia applications extracted from the Mediabench suite [13] (*adpcm encode*, *adpcm decode*, *motion vector* for Mpeg2) taken from [2]. Table I reports the number of operations and branching blocks for the above mentioned set of benchmarks (upper part of the table) as well as a further set used in a following analysis on ILP complexity (lower part of the table).

The experimental results compare three different scheduling techniques: *SPARK*, *LIST* and *ILP*. *SPARK*: shows the results obtained by the *Spark* framework enabling all its features, e.g. code motion, speculation; *LIST*: represents the results obtained by applying the *List-Based scheduling* techniques with the presented intermediate representation[1]; *ILP*: represents the results obtained by applying the *ILP* formulation presented in section III using our intermediate representation;

The results for each technique and each target benchmark are shown in terms of the computational time needed to generate the schedule and the number of control steps. The computational times are average values obtained by applying the scheduling processes for ten times.

The table II shows the obtained results with the following different configurations:

- *ARCH-1 - 1-Add, 1-Sub, 1-Mul, 1-Cmp, 1-Sh, 2-[]*
- *ARCH-2 - 1-Add, 1-Sub, 1-Mul, 2-Cmp, 1-Sh, 2-[]*
- *ARCH-3 - 2-Add, 2-Sub, 1-Mul, 2-Cmp, 1-Sh, 2-[]*

where $X- < res >$ means that X resources for $< res >$ are allocated. *Add*, *Sub*, *Mul*, *Cmp*, *Sh* and *[]* stand for adder, subtractor, multiplier, comparator, shifter and array address decoder, respectively.

Three scheduling techniques are applied using three different architectures. With the first architecture (*ARCH-1*) the number of control steps obtained using the *Spark* framework is always greater or equal to the value obtained by the two methods that use the intermediate representation presented in the paper. Moreoever, our approach increases its effectiveness on applications of increasing complexity. The ILP approach improved the results obtained by LIST, in terms of control steps for *Motion Vector* (11 w.r.t. 12) and *Sehwa* (8 w.r.t. 9).

Similar results are shown with the other two configurations *ARCH-2* and *ARCH-3*, where the advantage obtained by the proposed scheduling over SPARK is up to 27% for *ARCH-2* and up to 33% for *ARCH-3*, both for the *AdpcmEncode* benchmark.

Experimental results that take into account the time needed for the scheduling show that this value is comparable for SPARK and LIST. On the other side, the ILP approach requires a time which is one order of magnitude greater with respect to SPARK and LIST, excluding the *Motion Vector* benchmark where for the architectures *ARCH-1* and *ARCH-2* it is up to two order of magnitude larger. As we expected, by increasing the number of functional units available for the scheduling, the problem becomes easier to solve and the time needed for the scheduling decreases.

[1]List-based scheduling can be directly adapted to the combined CDG+DDG data structure

To better understand the power of the branch and cut approach with respect to the branch and bound technique we performed further experiments on a larger set of benchmarks. In particular, we have enriched the set with some well known data-intensive high level synthesis benchmarks.

Table III compares the results obtained by the LIST approach with a branch and bound (*B&B*) and branch and cut (*B&C*) applied to the ILP formulation described in section III, reporting the number of control steps obtained and the computational time required. Note that, the LIST approach is heuristic while the other two also proved the optimality of the solution if the required time is less than the time limit of 1000 seconds.

Table III clearly shows that branch and cut is more scalable than the standard branch and bound, solving several problems in a reasonable computation time with the exception of two cases. In fact, on small benchmarks the overhead due to the generation of valid inequalities makes branch and cut slower. On the other hand, the branch and bound is not able to solve 11 of the 26 scheduling problems while branch and cut solves all apart 2 benchmarks.

Benchmark	#Operations	Branching Blocks
Kim	33	3
Sehwa	29	6
Maha	29	6
MotionVector	100	11
AdpcmDecode	87	11
AdpcmEncode	108	15
Chemical	38	1
Dct_wang	57	1
Ewf	39	1
Paulin	15	1
Pr1	51	1
Tseng	13	1
Wdf	44	1

TABLE I
OPERATIONS AND BRANCHING BLOCKS.

V. CONCLUDING REMARKS

The complexity of design for modern applications has extremely grown in recent years. This means that the standard techniques for high level synthesis can be considered obsolete for a certain number of new designs. To cope with this problem, recent research results have demonstrated, for example, the effectiveness of speculative code transformations on mixed control-data flow design to reduce the length of the resulting schedules. Our work proposes an approach based on a new data structure, the control and data dependence graph, that allows a better exploitation of parallelism present in the original specification. Moreover, this work introduces an Integer Linear Programming formulation of the scheduling problem, which minimizes the number of control steps given the amount of resources. The capacity constraint introduced has been verified to be well suited to manage resource sharing constraints in case of speculative computations. The experimental results show the validity of the proposed methodology. In general, the quality of the solution provided by the heuristic approach is nearly optimal showing that the data structure based on the

ARCH-1

	SPARK		LIST		ILP	
	Control Steps	Time [sec]	Control Steps	Time[Sec]	Control Steps	Time[sec]
Kim	10	0.030	10	0.013	10	0.169
Sehwa	8	0.046	9	0.015	8	0.332
Maha	9	0.049	9	0.013	9	0.125
MotionVector	15	0.309	12	0.173	11	28.2
AdpcmDecode	18	0.110	13	0.212	13	3.357
AdpcmEncode	18	0.144	14	0.210	14	2.011

ARCH-2

	SPARK		LIST		ILP	
	Control Steps	Time [sec]	Control Steps	Time[Sec]	Control Steps	Time[sec]
Kim	10	0.054	10	0.012	9	0.163
Sehwa	7	0.045	7	0.014	7	0.132
Maha	9	0.054	9	0.012	9	0.129
MotionVector	13	0.274	12	0.177	11	26.2
AdpcmDecode	15	0.122	11	0.190	11	0.807
AdpcmEncode	18	0.145	13	0.519	13	1.237

ARCH-3

	SPARK		LIST		ILP	
	Control Steps	Time [sec]	Control Steps	Time[Sec]	Control Steps	Time[sec]
Kim	8	0.036	8	0.010	8	0.074
Sehwa	6	0.049	6	0.013	6	0.085
Maha	9	0.054	9	0.011	9	0.123
MotionVector	10	0.323	10	0.162	9	0.968
AdpcmDecode	15	0.114	10	0.164	10	0.474
AdpcmEncode	18	0.148	13	0.367	13	0.843

TABLE II
EXPERIMENTAL RESULTS OBTAINED USING DIFFERENT ARCHITECTURES.

ARCH-1

	LIST		ILP-B&B		ILP-B&C	
	Control Steps	Time [sec]	Control Steps	Time[Sec]	Control Steps	Time[sec]
Kim	10	0.013	10	0.099	10	0.169
Sehwa	9	0.015	8	0.201	8	0.332
Maha	9	0.013	9	0.080	9	0.125
MotionVector	12	0.173	12	>1000	11	28.2
AdpcmDecode	13	0.212	13	>1000	13	3.357
AdpcmEncode	14	0.210	14	2.583	14	2.011
Chemical	19	0.021	19	>1000	19	56.4
Dct_wang	25	0.031	25	>1000	25	>1000
Ewf	21	0.021	21	>1000	21	>1000
Paulin	8	0.005	8	0.044	8	0.137
Pr1	19	0.022	19	>1000	19	48.56
Tseng	6	0.006	6	0.021	6	0.057
Wdf	28	0.020	28	>1000	28	84.21

ARCH-3

	LIST		ILP-B&B		ILP-B&C	
	Control Steps	Time [sec]	Control Steps	Time[Sec]	Control Steps	Time[sec]
Kim	8	0.010	8	0.056	8	0.074
Sehwa	6	0.013	6	0.059	6	0.085
Maha	9	0.011	9	0.078	9	0.123
MotionVector	10	0.162	9	5.507	9	0.968
AdpcmDecode	10	0.164	10	0.317	10	0.474
AdpcmEncode	13	0.367	13	0.783	13	0.843
Chemical	19	0.020	19	>1000	19	49.03
Dct_wang	24	0.031	24	>1000	23	342.7
Ewf	19	0.020	19	>1000	19	182.7
Paulin	8	0.005	8	0.047	8	0.129
Pr1	18	0.022	18	>1000	18	36.717
Tseng	5	0.005	5	0.014	5	0.033
Wdf	17	0.016	17	0.145	17	0.620

TABLE III
COMPARISON OF RESULTS OBTAINED WITH LIST BASED SCHEDULING, STANDARD BRANCH AND BOUND AND BRANCH AND CUT.

combined CDG+DDG is better than other intermediate representations previously proposed in literature. Therefore, future work will consider the introduction of speculative computation into the List based algorithm. The ILP approach has shown reasonable execution time and can be fruitfully used to optimize kernel functions, where a larger computation time can be afforded.

VI. Acknowledgments

This publication has been part funded by the European Commission's Sixth Framework Programme.

References

[1] GCC - GNU Compiler Collection. http://gcc.gnu.org.

[2] Spark synthesis benchmarks ftp site. ftp://ftp.ics.uci.edu/pub/spark/benchmarks.

[3] M. I. H. Catherine H. Gebotys. Global optimization approach for architectural synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(9):1266–1278, September 1993.

[4] L. dos Santos and J. Jess. A reordering technique for efficient code motion. In *Design Automation Conference*, 1999.

[5] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *3rd International Conference on Supercomputing*, 1989.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Language and Systems*, 9(3):319–349, 1987.

[7] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, July 1981.

[8] M. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):166–178, March 1992.

[9] M. Girkar and C. Polychronopoulos. Extracting task-level parallelism. *ACM Trans. on Programming Language and Systems*, 17(4):600–634, July 1995.

[10] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Using global code motions to improve the quality of results for high-level synthesis. *IEEE Transactions on CAD*, 23(2), February 2003.

[11] T. Kim, N. Yonezawa, J. Liu, and C. Liu. A scheduling algorithm for conditional resource sharing - a hierarchical reduction approach. *IEEE Transactions on CAD*, April 1994.

[12] G. Lakshminarayana, A. Raghunathan, and N. Jha. Wavesched: a novel scheduling technique for control-flow intensive designs. *IEEE Transactions on CAD*, May 1999.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, 1997.

[14] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. on Programming Language and Systems*, 1(1):121–141, 1979.

[15] R. Lougee-Heimer, F. Barahona, B. Dietrich, J. P. Fasano, J. Forrest, R. Harder, L. Ladanyi, T. Pfender, T. Ralphs, M. Saltzman, and K. Schienberger. The coin-or initiative: Open-source software accelerates operations research progress. *ORMS Today*, 28(5):20–22, October 2001 2001.

[16] J. Merril. Generic and gimple: A new tree representation for entire functions. In *Proceedings of GCC Developers Summit*, pages 171 – 180, 2003.

[17] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, 1988.

[18] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *International Conference on Parallel Processing*, 1993.

[19] N. Park and A. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on Computer-Aided Design*, March 1988.

[20] A. Parker, J. Pizarro, and M. Mlinar. MAHA: A program for datapath synthesis. In *Design Automation Conference*, 1986.

[21] M. Potkonjak and J. Rabaey. Optimizing resource utlization using tranformations. *IEEE Trans. on CAD*, March 1994.

[22] I. Radivojevic and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE Transactions on CAD*, January 1996.

[23] M. Rim, Y. Fann, and R. Jain. Global scheduling with code-motions for high-level synthesis applications. *IEEE Transactions on VLSI Systems*, September 1995.