Optimal Integrated VLIW Code Generation with Integer Linear Programming

Andrzej Bednarski and Christoph Kessler

PELAB, Department of Computer and Information Science,
Linköpings universitet, S-58183 Linköping, Sweden
andbe@ida.liu.se, chrke@ida.liu.se

Abstract. We give an Integer Linear Programming (ILP) solution that fully integrates all steps of code generation, *i.e.* instruction selection, register allocation and instruction scheduling, on the basic block level for VLIW processors.
In earlier work, we contributed a dynamic programming (DP) based method for optimal integrated code generation, implemented in our retargetable code generator OPTIMIST. In this paper we give first results to evaluate and compare our ILP formulation with our DP method on a VLIW processor. We also demonstrate how to precondition the ILP model by a heuristic relaxation of the DP method to improve ILP optimization time.

1 Introduction

We consider the problem of optimal integrated code generation for instruction-level parallel architectures such as VLIW processors. Integrated code generation solves simultaneously, in a single optimization pass, the tasks of instruction selection, instruction scheduling including resource allocation and code compaction, and register allocation.

In previous work [8], we developed a dynamic programming approach and implemented it in our retargetable framework called OPTIMIST [9]. However, there may be further general problem solving strategies that could likewise be applied to the integrated code generation problem. In this paper, we consider the most promising of these, *integer linear programming* (ILP).

ILP is a general-purpose optimization method that gained much popularity in the past 15 years due to the arrival of efficient commercial solvers and effective modeling tools. In the domain of compiler back ends, it has been used successfully for various tasks in code generation, most notably for instruction scheduling.

Wilken et al. [12] use ILP for instruction scheduling of basic blocks which allows, after preprocessing the basic block's data flow graph, to derive optimal solutions for basic blocks with up to 1000 instructions within reasonable time.

ILP formulations integrating instruction scheduling and resource allocation are either *time-based* or *order-based*. In time-based formulations the main decision variables indicate the time slot when an operation is to be started. In order-based formulations the decision variables represent the flow of the hardware resources among operations.

Gebotys *et al.* [5] give a time-based formulation that integrates instruction scheduling and resource allocation and computes time optimal schedules. Leupers and Marwedel [10] provide a time-based ILP formulation for code compaction of a given instruction sequence with alternative instruction encodings.

Zhang [16], Chang *et al*. [2] and Kästner [7] provide order-based and/or time-based ILP formulations for the combination of instruction scheduling with register allocation. Winkel [15] formulates an ILP model for post-pass optimization that can be solved efficiently for global instruction scheduling, including code motion and predication.

We know of only one ILP formulation in the literature that addressed all three tasks simultaneously, which was proposed by Wilson *et al*. [13, 14]. However, their formulation is for single-issue architectures only. Furthermore, their proposed model assumes that the alternatives for pattern matching in instruction selection be exposed explicitly for each node and edge of the basic block's data flow graph (DFG), which would require a preprocessing of the DFG before the ILP problem instance can be generated.

We provide an ILP formulation that fully integrates all three phases of code generation and extends the machine model used by Wilson *et al*. by including VLIW architectures with homogeneous register file. Moreover, our formulation does no longer need preprocessing of the DFG.

The remainder of this paper is organized as follows: After introducing some notation, we provide in Section 3 the ILP formulation for fully integrated code generation for VLIW processors. For a description of the DP approach of OPTIMIST, we refer to a recent article [8]. Section 4 evaluates the DP approach against the ILP approach, and draws some conclusions. Section 5 discusses further directions of ILP approach and Section 6 concludes the article.

2 Notation

We use uppercase letters to denote model parameters and constants provided to the ILP formulation. Lowercase letters denote solution variables and indexes.

Indexes i and j denote nodes of the DFG. We reserve indexes k and l for instances of nodes composing a given pattern. t is used for time index. We use the common notation $|X|$ to denote the cardinality of a set (or pattern) X.

As usual, instruction selection is modeled as a general pattern matching problem, covering the DFG with instances of patterns that correspond to instructions of the target processor. The set of patterns B is subdivided into patterns that consist of a single node, called *singletons* (B''), and patterns consisting of more than one node, with or without edges (B'). That is, $B = B' \cup B''$ such that $\forall p \in B', |p| > 0$ and $\forall p \in B'', |p| = 1$.

In the ILP formulation that follows, we provide several instances of each non-singleton pattern. For example, if there are two locations in the DFG where a multiply-accumulate pattern (MAC) is matched, these will be associated with two different instances of the MAC pattern, one for each possible location. We require that each pattern instance be matched at most once in the final solution. As a consequence, the model requires to specify a sufficient number of pattern instances to cover the DFG. For singleton patterns, we only need a single instance. This will become clearer once we have introduced the coverage equations where the edges of a pattern must correspond to some DFG edges.

2.1 Solution variables

The ILP formulation uses the following solution variables:

- $c_{i,p,k,t}$ a binary variable that is equal to 1, if a DAG node i is covered by instance node k of pattern p at time t. Otherwise the variable is 0.
- $w_{i,j,p,k,l}$ a binary variable that is equal to 1 if DFG edge (i,j) is covered by a pattern edge (k,l) of pattern $p \in B'$ (see Figure 1).
- $s_{p,t}$ a binary variable that is set to 1 if a pattern $p \in B'$ is selected and the corresponding instruction issued at time t, and to 0 otherwise.
- $r_{i,t}$ a binary variable that is set to 1 if DFG node i must reside in some register at time t, and 0 otherwise.
- τ an integer variable that represents the execution time of the final schedule.

In the equations that follow, we use the abbreviation $c_{i,p,k}$ for the following expression $\sum_{\forall t \in 0..T_{max}} c_{i,p,k,t}$, and s_p for $\sum_{\forall t \in 0..T_{max}} s_{p,t}$.

2.2 Parameters to the ILP model

The model we provide is sufficiently generic to be used for various instruction-level parallel processor architectures. Our ILP model requires the following parameters:

Data flow graph:
- G index set of DFG nodes
- E_G index set of DFG edges
- OP_i operation identifier of node i, representing a given DFG operation.
- OUT_i indicates the out-degree of DFG node i.

Patterns and instruction set:
- B' index set of instances of non-singleton patterns
- B'' index set of singletons (instances)
- E_p set of edges for pattern $p \in B'$
- $OP_{p,k}$ operator for an instance node k of pattern instance p. This relates to the operation identifier of the DFG nodes.
- $OUT_{p,k}$ is the out-degree of a node k of pattern instance p.
- L_p is an integer value representing the latency for a given pattern p. In our notation, each pattern is mapped to a unique target instruction, resulting in unique latency value for that pattern.

Resources:
- F is an index set of functional unit types.
- M_f represents the amount of functional units of type f, where $f \in F$.
- $U_{p,f}$ is a binary value representing the connection between the target instruction corresponding to a pattern (instance) p and a functional unit f that this instruction uses. It is 1 if p requires f, otherwise 0.
- W, is a positive integer representing the issue width of the target processor, *i.e.*, the maximum number of instructions that can be issued per clock cycle.
- R denotes the number of available registers.
- T_{max} is a parameter that represents the maximum execution time budget for a basic block. The value of T_{max} is only required for limiting the search space, and has no impact on the final result. Observe that T_{max} must be greater (or equal) than the time required for an optimal solution, otherwise the ILP problem instance has no solution.

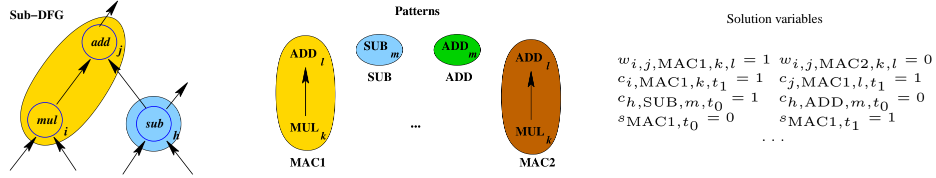

Fig. 1. Example of pattern matching.

3 ILP formulation

To provide the ILP model for fully integrated code generation for VLIW architectures, we first give equations for covering the DFG G with a set of patterns, *i.e.* the instruction selection. Secondly, we specify the set of equations for register allocation. Here we address regular architectures with general purpose registers, and thus only check that the register need does not exceed the amount of physical registers at any time. Next, we address scheduling issues. Since we are working on the basic block level, only flow dependences are considered. We assure that the schedule never exceeds available resources, and that instructions issued simultaneously fit into a long instruction word.

3.1 Instruction selection

Our instruction selection model is suitable for tree-based and directed acyclic graph (DAG) data flow graphs. Also, it handles patterns in the form of tree, forest, and DAG patterns. The goal of instruction selection is to cover all nodes of DFG G with a set of patterns. For each DFG node i there must be exactly one matching node k in a pattern instance p. Equation (1) enforces this full-coverage property. Solution variable $c_{i,p,k,t}$ records for each node i which pattern instance node covers it, and at what time. Beside full coverage, Equation (1) also assures a requirement for scheduling, namely that for each DFG node i, the instruction corresponding to the pattern instance p covering it is scheduled (issued) at some time slot t.

$$\forall i \in G, \sum_{p \in B} \sum_{k \in p} c_{i,p,k} = 1 \tag{1}$$

Equation (2) records the set of pattern instances being selected for DFG coverage. If a pattern instance p is selected, all its nodes should be mapped to distinct nodes of G. Additionally, the solution variable $s_{p,t}$ carries the information at what time t a selected pattern instance p is issued.

$$\forall p \in B', \forall t \in 0..T_{max}, \sum_{i \in G} \sum_{k \in p} c_{i,p,k,t} = |p| s_{p,t} \tag{2}$$

If a pattern instance p is selected, each pattern instance node k maps to exactly one DFG node i. Equation (3) considers this unique mapping only for selected patterns, as recorded by the solution variables s.

$$\forall p \in B', \forall k \in p, \sum_{i \in G} c_{i,p,k} = s_p \tag{3}$$

Equation (4) implies that all edges composing a pattern must coincide with exactly the same amount of edges in G. Thus, if a pattern instance p is selected, it should cover exactly $|E_p|$ edges of G. Unselected pattern instances do not cover any edge of G. Remark that in our model each pattern instance is distinct, and that we further assume that there are enough pattern instances available to fully cover a particular DFG.

$$\forall p \in B', \sum_{(i,j)\in E_G} \sum_{(k,l)\in E_p} w_{i,j,p,k,l} = |E_p| s_p \tag{4}$$

Equation (5) assures that a pair of nodes constituting a DFG edge covered by a pattern instance p corresponds to a pair of pattern instance nodes. If we have a match ($w_{i,j,p,k,l} = 1$) then we must map DFG node i to pattern instance node k and node j to pattern instance node l of pattern instance p.

$$\forall (i,j) \in E_G, \forall p \in B', \forall (k,l) \in E_p,\ 2w_{i,j,p,k,l} \leq c_{i,p,k} + c_{j,p,l} \tag{5}$$

Equation (6) imposes that instructions corresponding to a non-singleton pattern (instance) p are issued at most once at some time t (namely, if p was selected), or not at all (if p was not selected).

$$\forall p \in B',\ s_p \leq 1 \tag{6}$$

Equation (7) checks that the IR operators of DFG (OP_i) corresponds to the operator $OP_{p,k}$ of node k in the matched pattern instance p.

$$\forall i \in G, \forall p \in B, \forall k \in p, \forall t \in 0..T_{max}, c_{i,p,k,t}(OP_i - OP_{p,k}) = 0 \tag{7}$$

Equation (8) simply checks if the out-degree $OUT_{p,k}$ of node k of a pattern instance p equals the out-degree OUT_i of the covered DFG node i. As nodes in singleton patterns are always pattern root nodes, we only need to consider non-singleton patterns, *i.e.* the set B'.

$$\forall p \in B', \forall (i,j) \in E_G, \forall (k,l) \in p,\ w_{i,j,p,k,l}(OUT_i - OUT_{p,k}) = 0 \tag{8}$$

3.2 Register allocation

Currently we address (regular) architectures with general-purpose register set. We leave modeling of clustered architectures for future work. Thus, a value carried by an edge not entirely covered by a pattern (active edge), requires a register to store that value. Equation (9) forces a node i to be in a register if at least one of its outgoing edge is active, where N is a large number considered to be infinity.

$$\forall t \in 0..T_{max}, \forall i \in G, \sum_{t_t=0}^{t} \sum_{(i,j)\in E_G} \sum_{p\in B} \left(\sum_{k\in p} c_{i,p,k,t_t} - \sum_{l\in p} c_{j,p,l,t_t} \right) \leq N r_{i,t} \tag{9}$$

If all outgoing edges from a node i are covered by a pattern instance p, there is no need to store the value represented by i in a register. Equation (10) requires solution variable $r_{i,t}$ to be set to 0 if all outgoing edges from i are inactive at time t.

$$\forall t \in 0..T_{max}, \forall i \in G, \sum_{t_t=0}^{t} \sum_{(i,j)\in E_G} \sum_{p\in B} \left(\sum_{k\in p} c_{i,p,k,t_t} - \sum_{l\in p} c_{j,p,l,t_t} \right) \geq r_{i,t} \tag{10}$$

Finally, Equation (11) checks that register pressure does not exceed the number R of available registers at any time.

$$\forall t \in 0..T_{max},\ \sum_{i \in G} r_{i,t} \leq R \tag{11}$$

3.3 Instruction scheduling

The scheduling is complete when each node has been allocated to a time slot in the schedule such that there is no violation of precedence constraints and resources are not oversubscribed. Since we are working on the basic block level, we only need to model the true data dependences, represented by DFG edges. Data dependences can only be verified once pattern instances have been selected, covering the whole DFG. The knowledge of the covered nodes with their respective covering pattern (*i.e.*, the corresponding target instruction) provides the necessary latency information for scheduling.

Besides full coverage, Equation (1) constrains each node to be scheduled at some time t in the final solution. We need additionally to check that all precedence constraints (data flow dependences) are satisfied. There are two cases: First, if an edge is entirely covered by a pattern p (inactive edge), the latency of that edge must be 0, which means that for all inactive edges (i, j), DFG nodes i and j are "issued" at the same time. Secondly, edges (i, j) between DFG nodes matched by different pattern instances (active edges) should carry the latency L_p of the instruction whose pattern instance p covers i. Equations (12) and (13) guarantee the flow data dependences of the final schedule. We distinguish between edges leaving nodes matched by a multi-node pattern, Equation (12), and the case of edges outgoing from singletons, Equation (13). Active edges leaving a node covered by a singleton pattern p carry always the latency L_p of p.

$$\forall p \in B',\ \forall (i,j) \in E_G,\ \forall t \in 0..T_{max} - L_p + 1,$$
$$\sum_{k \in p} c_{i,p,k,t} + \sum_{\substack{q \in P \\ q \neq p}} \sum_{t_t=0}^{t+L_p-1} \sum_{k \in q} c_{j,q,k,t_t} \leq 1 \tag{12}$$

$$\forall p \in B'',\ \forall (i,j) \in E_G,\ \forall t \in 0..T_{max} - L_p + 1,$$
$$\sum_{k \in p} c_{i,p,k,t} + \sum_{q \in B} \sum_{t_t=0}^{t+L_p-1} \sum_{k \in q} c_{j,q,k,t_t} \leq 1 \tag{13}$$

3.4 Resource allocation

A schedule is valid if it respects data dependences and its resource usage does not exceed the available resources (functional units, registers) at any time. Equation (14) verifies that there are no more resources required by the final solution than available on the target architecture. In this paper we assume fully pipelined functional units with an occupation time of one for each unit, *i.e.* a new instruction can be issued to a unit every new clock cycle. The first summation counts the number of resources of type f required by instructions corresponding to selected multi-node pattern instances p at time t. The

second part records resource instances of type f required for singletons (scheduled at time t).

$$\forall t \in 0..T_{max},\ \forall f \in F,\ \sum_{\substack{p \in B' \\ U_{p,f}=1}} s_{p,t} + \sum_{\substack{p \in B'' \\ U_{p,f}=1}} \sum_{i \in G} \sum_{k \in p} c_{i,p,k,t} \leq M_f \tag{14}$$

Finally Equation (15) assures that the issue width W is not exceeded. For each issue time slot t, the first summation of the equation counts for multi-node pattern instances the number of instructions composing the long instruction word issued at t, and the second summation for the singletons. The total amount of instructions should not exceed the issue width W, *i.e.*, the number of available slots in a VLIW instruction word.

$$\forall t \in 0..T_{max},\ \sum_{p \in B'} s_{p,t} + \sum_{p \in B''} \sum_{i \in G} \sum_{k \in p} c_{i,p,k,t} \leq W \tag{15}$$

3.5 Optimization goal

In this paper we are looking for a time-optimal schedule for a given basic block. The formulation however allows us not only to optimize for time but can be easily adapted for other objective functions. For instance, we might look for the minimum register usage or code length.

In the case of time optimization goal, the total execution time of a valid schedule is derived from the solution variables c as illustrated in Equation (16).

$$\forall i \in G,\ \forall p \in P,\ \forall k \in p,\ \forall t \in 0..T_{max},\ c_{i,p,k,t} * (t + L_p) \leq \tau \tag{16}$$

The total execution time is less or equal to the solution variable τ. Looking for a time optimal schedule, our objective function is to minimize τ.

4 Evaluation

First, we provide two theoretical VLIW architectures for which we generate target code. Secondly we describe the experimental setup that we used to evaluate our ILP formulation against our previous DP approach and summarize the results.

4.1 Target Architectures

In order to compare OPTIMIST's DP technique to the ILP formulation of Section 3, we use two theoretical VLIW target platforms (Case I and Case II) with the following characteristics.

Case I: The issue width is a maximum of two instructions per clock cycle. The architecture has an arithmetic-logical unit (`ALU`). Most ALU operations require a single clock cycle to compute (occupation time and latency are one). Multiplication and division operations have a latency of two clock cycles. Besides the ALU, the architecture has a multiply-and-accumulate unit (`MAC`) that takes two clock cycles to perform a multiply-and-accumulate operation. There are eight general purpose registers accessible from

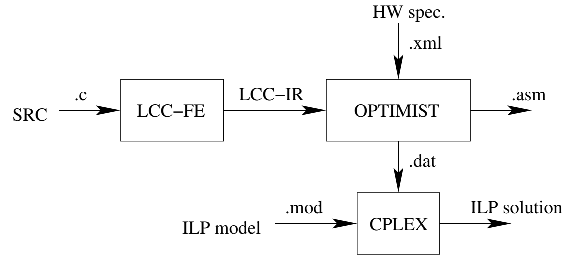

Fig. 2. Experimental setup.

any unit. We assume a single memory bank with unlimited size. A load/store unit (`LS`) stores and loads data in four clock cycles.
Case II: The issue width is of maximum four instructions per clock cycle. The architecture has twice as many resources as in Case I, *i.e.* two arithmetic-logical units, two multiply-and-accumulate units, and two load/store units with the same characteristics.

4.2 Experimental Setup

We implemented the ILP data generation module within the OPTIMIST framework. Currently our ILP model addresses VLIW architectures with regular pipeline, *i.e.* functional units are pipelined, but no pipeline stall occurs. We adapted hardware specifications in xADML [1, Chap. 8] such that they fit current limitations of the ILP model. In fact, the OPTIMIST framework accepts more complex resource usage patterns and pipeline descriptions expressible in xADML, which uses the general mechanism of reservation tables [3]. As assumed in Section 3, we use for the ILP formulation the simpler model with unit occupation time and a latency for each instruction. An extension of the ILP formulation to use general reservation tables is left to future work.

Figure 2 shows our experimental platform. We provide a plain C code sequence as input to OPTIMIST. We use LCC [4] (within OPTIMIST) as C front-end. Besides the source code we provide the description of the target architecture in xADML language For each basic block, OPTIMIST outputs the assembly code as result. If specified, the framework also outputs the data file for the ILP model of Section 3. The data file contains architecture specifications, such as the issue width of the processor, the set of functional units, patterns, *etc*. that are extracted from the architecture description document. It generates all parameters introduced in Section 2.2. Finally we use the CPLEX solver [6] to solve the set of equations.

Observe that for the ILP data we need to provide the upper bound for the maximum execution time (T_{max}). For that, we first run a heuristic variant of DP that still considers full integration of code generation phases, and provide its execution time (computed in a fraction of a second) as the T_{max} parameter to the ILP data.

4.3 Results

We generated code for basic blocks taken from various digital signal processing benchmark programs. We run the evaluation of the DP approach on a Linux (kernel 2.6.13)

PC with Athlon 1.6GHz CPU and 1.5GB RAM. The ILP solver runs on a Linux (kernel 2.6.12) PC with Athlon 2.4GHz CPU, 512MB RAM using CPLEX 9.

We should mention a factor that contributes in favor of the ILP formulation. In the OPTIMIST framework we use LCC [4] as C front-end. Within our framework we enhanced the intermediate representation with extended basic blocks [11] (which is not standard in LCC). As consequence, we introduced data dependence edges for resolving memory write/read precedence constraints. In the current ILP formulation we consider only data flow dependences. Thus, we instrumented OPTIMIST to remove edges introduced by building extended basic blocks. Removing dependence edges results in DAGs with larger base, *i.e.* with larger number of leaves, and in general a lower height. We are aware that the DP approach suffers from DAGs with a large number of leaves, as OPTIMIST early generates a large number of partial solutions. Further, removing those edges builds DFGs that may no longer be equivalent to the original C source code. However, it is still valid to compare the ILP and DP techniques, since both formulations operate on the same intermediate representation.

Table 1 reports our results for the Case I architecture. The first column indicates the name of the basic block. The second column reports the number of nodes in the DAG for that basic block. The third and fourth columns give the height of the DAG and the number of edges, respectively. Observe that the height corresponds to the longest path of the DAG in terms of number of DAG nodes, and not to its critical path length,

Table 1. Evaluation of ILP and DP fully integrated code generation approaches for the Case I architecture.

	Basic block	$\lvert G \rvert$	Height	$\lvert E_G \rvert$	DP		ILP	
					τ (cc)	t (sec)	τ (cc)	t (sec)
1)	iir filter bb9	10	4	10	10	0.3	10	0.9
2)	vec_max bb8	12	4	12	11	0.6	11	1.3
3)	dijkstra bb19	16	7	15	14	6.6	14	5.6
4)	fir filter bb9	16	3	14	15	61.3	15	7.8
5)	cubic bb16	17	6	16	14	15.0	14	5.7
6)	fir_vselp bb10	17	9	17	16	3.4	16	8.2
7)	matrix_sum loop bb4	17	8	17	16	4.0	16	8.8
8)	scalarprod bb2	17	8	18	17	1.2	17	15.8
9)	vec_sum bb3	17	8	18	16	1.4	16	11.8
10)	matrix_copy bb4	18	7	19	16	4.3	16	12.5
11)	cubic bb4	21	8	23	17	69.8	17	277.7
12)	iir filter bb4	21	6	17	20	3696.4	20	46.5
13)	fir filter bb11	22	6	27	19	89.7	CPLEX	
14)	codebk_srch bb20	23	7	22	17	548.8	17	63.1
15)	fir_vselp bb6	23	9	25	19	40.6	CPLEX	
16)	summatrix_un1 bb4	24	10	28	20	25.4	CPLEX	
17)	scalarprod_un1 bb2	25	10	30	19	14.9	CPLEX	
18)	matrixmult bb6	30	9	35	23	2037.7	AMPL	
19)	vec_sum unrolled bb2	32	10	40	24	810.9	AMPL	
20)	scalarprod_un2 bb2	33	12	42	23	703.1	AMPL	

whose calculation is unfeasible since the instruction selection is not yet known. The fifth column reports the amount of clock cycles required for the basic block, and in the sixth column we display the computation time (in seconds) for finding a DP solution. Columns seven and eight report the results for ILP. The computation time for the ILP formulation does not include the time for CPLEX-presolve that optimizes the equations.

In the tables we use three additional notations: CPLEX indicates that the ILP solver ran out of memory and did not compute a result. AMPL means that CPLEX-presolve failed to generate an equation system, because it ran out of memory. Where the DP ran out of memory we indicate the entry as MEM.

For all cases that we could check both techniques report the same execution time (τ). It was unexpected to see that the ILP formulation performs quite well and in several cases with an order of magnitude faster than DP. For cases 4), 12) and 14) in Table 1 the DP takes almost eight times, eighty times and nine times respectively longer than the ILP solver to compute an optimal solution. Since we removed the memory data dependence edges (as mentioned earlier) the resulting test cases present two, four and two unrelated DAGs for case 4), 12) and 14) respectively. We know that DP suffers from DAGs with a large number of leaves because a large number of selection nodes is generated already at the first step. For the rest of the test cases, DP outperforms the ILP formulation or has similar computation times. Observe that we reported for cases 3) and 5) that ILP takes shorter time to compute an optimal solution. But if we include the time of CPLEX-presolve, which runs for 7.1s in case 3) and 8.3s in case 5), the ILP times are worse or equivalent. For problems larger than 22 nodes, the ILP formulation fails to compute a solution. For problem instances over 30 nodes, the CPLEX-presolve does not generate equations because it runs out of memory.

Table 2. Evaluation of ILP and DP fully integrated code generation approaches for the Case II architecture.

	Basic block	$\lvert G \rvert$	Height	$\lvert E_G \rvert$	DP τ (cc)	DP t (sec)	ILP τ (cc)	ILP t (sec)	ILP t′ (sec)
1)	iir filter bb9	10	4	10	9	0.6	9	1.5	0.4
2)	vec_max bb8	12	4	12	10	2.4	10	1.6	0.7
3)	dijkstra bb19	16	7	15	14	73.5	14	10.7	4.2
4)	fir filter bb9	16	3	14	9	2738.9	9	9.1	2.5
5)	cubic bb16	17	6	16	12	1143.3	12	CPLEX	3.8
6)	fir_vselp bb10	17	9	17	14	62.1	14	CPLEX	4.9
7)	matrix_sum loop bb4	17	8	17	15	90.2	15	CPLEX	10.2
8)	scalarprod bb2	17	8	18	15	10.0	—	CPLEX	CPLEX
9)	vec_sum bb3	17	8	18	13	11.4	13	CPLEX	4.6
10)	matrix_copy bb4	18	7	19	14	89.4	14	AMPL	4.1
11)	cubic bb4	21	8	23	16	8568.7	—	AMPL	CPLEX
12)	iir filter bb4	21	6	17	—	MEM	12	AMPL	7.4
13)	fir filter bb11	22	6	27	—	MEM	—	AMPL	CPLEX
14)	codebk_srch bb20	23	7	22	—	MEM	—	AMPL	CPLEX
15)	fir_vselp bb6	23	9	25	16	7193.9	—	AMPL	AMPL

Table 2 shows the results for the Case II architecture. The notations are the same as for Case I. We added an additional column in the ILP part, denoted t′, that reports the ILP computation time when the upper bound T_{max} is derived from a run of a heuristically pruned DP algorithm [1, Chap. 4] (this decreases the number of generated equations by providing a value of T_{max} closer to an optimal solution). The time for this DP run for preconditioning the ILP (within a fraction of a second) is not included in t′.

For the cases 4) and 12) in Table 2, DP performs worse than ILP. For the case 12) DP runs out of memory, whereas the ILP could compute a solution within 7.4s if T_{max} is close enough to the optimum. The case II results show that it is beneficial to spend time on minimizing T_{max}. We could gain four additional nodes in ILP problem size. For Case II, if the ILP computes a solution it outperforms the DP.

5 Future work

The current ILP formulation lacks several features of the OPTIMIST framework. In this paper we considered target architectures that suit the ILP model. We plan to extend the formulation to handle clustered VLIW architectures, such as Veloci-TI DSP variants. For that, we will need to model operand residences (*i.e.*, in which cluster or register set a value is located). This will certainly increase the amount of generated variables and equations and affect ILP performance.

Also, we need to formulate the insertion of spill code. The current ILP formulation assumes a sufficient number of registers, which is not generally the case.

We also mentioned that the current ILP formulation is based on a simpler resource usage model that is limited to unit occupation times per functional unit and a variable latency per target instruction. It would be of interest to have a more general model using reservation tables for specifying arbitrary resource usage patterns and complex pipelines, which is already implemented in OPTIMIST's DP framework.

Finally, we will extend the scope of the optimization beyond the basic block level, in particular to integrated software pipelining of loops.

6 Conclusions

In this paper we provided an integer linear programming formulation for fully integrated code generation for VLIW architectures that includes instruction selection, instruction scheduling and register allocation. We extended the formulation by Wilson *et al.* [14] for VLIW architectures. In contrast to their formulation, we do no longer need to preprocess the DFG to expose instruction selection alternatives. Moreover, we have a working implementation where ILP instances are generated automatically from the OPTIMIST intermediate representation and a formal architecture description in xADML.

We compared the ILP formulation with our research framework for integrated code generation, OPTIMIST, which uses dynamic programming. We evaluated both methods on theoretical architectures that fit the ILP model restrictions. Where the ILP solver terminates successfully, the ILP-based optimizer mostly works faster than the dynamic programming approach; on the other hand, it fails for several larger examples where

dynamic programming still provides a solution. Hence, the two approaches complement each other. Moreover, the ILP approach profits from preconditioning by a heuristic variant of DP.

Currently, our ILP formulation lacks support for memory dependences and for irregular architecture characteristics, such as clustered register files, complex pipelines, *etc*. We intend to complete the formulation as part of future work. Further we need to address insertion of spill code.

Acknowledgments We thank Petru Eles and Alexandru Andrei from ESLAB of Linköpings universitet for letting us using their CPLEX installation. This research was partially funded by the Ceniit program of Linköpings universitet and by SSF RISE.

References

1. A. Bednarski. *Integrated Optimal Code Generation for Digital Signal Processors*. PhD thesis, Linköpings universitet, Linköping, Sweden, June 2006.
2. C.-M. Chang, C.-M. Chen, and C.-T. King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers Mathematics and Applications*, 34(9):1–14, 1997.
3. E. S. Davidson, L. E. Shar, A. T. Thomas, and J. H. Patel. Effective control for pipelined computers. In *Proc. Spring COMPCON75 Digest of Papers*, pages 181–184. IEEE Computer Society Press, Feb. 1975.
4. C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Welsey Publishing Company, 1995.
5. C. H. Gebotys and M. I. Elmasry. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 2–7, New York, NY, USA, 1991. ACM Press.
6. I. Inc. CPLEX homepage. http://www.ilog.com/products/cplex/, 2005.
7. D. Kästner. *Retargetable Postpass Optimisations by Integer Linear Programming*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2000.
8. C. Kessler and A. Bednarski. Optimal integrated code generation for VLIW architectures. To appear in *Concurrency and Computation: Practice and Experience*, 2006.
9. C. Kessler and A. Bednarski. OPTIMIST. www.ida.liu.se/∼chrke/optimist, 2005.
10. R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. *IEEE Transactions on VLSI Systems*, 5(1):112–122, 1997.
11. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
12. K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 121–133, 2000.
13. T. Wilson, G. Grewal, B. Halley, and D. Banerji. An integrated approach to retargetable code generation. In *Proc. 7th international symposium on High-level synthesis (ISSS'94)*, pages 70–75. IEEE Computer Society Press, 1994.
14. T. C. Wilson, N. Mukherjee, M. Garg, and D. K. Banerji. An integrated and accelerated ILP solution for scheduling, module allocation, and binding in datapath synthesis. In *The Sixth Int. Conference on VLSI Design*, pages 192–197, Jan. 1993.
15. S. Winkel. *Optimal Global Instruction Scheduling for the Itanium® Processor Architecture*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, Sept. 2004.
16. L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming*. PhD thesis, Technische Fakultät der Universität des Saarlandes, Saarbrücken (Germany), 1996.