Integer Linear Programming versus Dynamic Programming for Optimal Integrated VLIW Code Generation

Andrzej Bednarski[1] and Christoph Kessler[1]

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

Abstract. To our knowledge there is only one Integer Linear Programming (ILP) formulation in the literature that fully integrates all steps of code generation, *i.e.* instruction selection, register allocation and instruction scheduling, on the basic block level. We give in this paper an improved version of this ILP formulation that also covers VLIW processors. Moreover, our ILP formulation does no longer require preprocessing the basic block's data flow graph to support instruction selection.
In earlier work, we proposed and implemented a dynamic programming (DP) based method for optimal integrated code generation, called OPTIMIST. In this paper we give first results to evaluate and compare our ILP formulation with our DP method on a VLIW processor.
We identify different code situations and shapes of data dependence graphs for which either ILP or DP is most suitable.

1 Introduction

We consider the problem of optimal integrated code generation for instruction-level parallel processor 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 [6], we developed a dynamic programming approach and implemented it in our retargetable framework called OPTIMIST [7]. 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).

While our original intention was just to obtain a quantitative comparison of our DP implementation to an equivalent ILP formulation, we see from the results reported in this paper that the two approaches actually appear to complement each other very well.

Integer linear programming (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. [9] 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. Zhang [12], Chang *et al.* [1] and Kästner [5] provide order-based and/or time-based ILP formulations for the combination of instruction scheduling with register allocation. Leupers and Marwedel [8] provide a time-based ILP formulation for code compaction of a given instruction sequence with alternative instruction encodings.

We know of only one ILP formulation in the literature that addressed all three tasks simultaneously, which was proposed by Wilson et al. [11, 10]. 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.

In this paper 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. Moreover, our formulation does no longer need preprocessing of the DFG.

The remainder of this paper is organized as follows: Section 2 provides 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 [6]. Section 3 evaluates the DP approach against the ILP approach, provides first results and draws some conclusions. Section 4 discusses further directions of ILP approach and Section 5 concludes the article.

2 The ILP formulation

In this section we first introduce various variables and parameters and then provide the ILP formulation for fully integrated code generation. However, first we introduce some notation that we use in the formulation.

2.1 Notation

In the remainder of this article, we use uppercase letters to denote parameters and constants provided to the ILP formulation (model). Lowercase letters denote solution variables and indexes. For example, F represents an index set of functional unit types, which is given as parameter to the ILP model.

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 G. 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.

2.2 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 $p \in B'$.
- $s_{p,t}$ 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.

We search for a schedule that minimizes the total execution time of a basic block. That is, we minimize τ.

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

2.3 Parameters to the ILP model

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

Data flow graph:

- G index set of DFG nodes
- E_G index set of DFG edges
- OPG_i operation identifier of node i. Each DFG node is associated with an integer value that represents a given 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.
- $ODP_{p,k}$ 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 value 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.

The rest of the section provides the ILP model for fully integrated code generation for VLIW architectures. First, we 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. Currently, 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 (true) data dependences are considered. Finally, we assure that, at any time, the schedule does not exceed available resources, and that the instructions issued simultaneously fit into a long instruction word, i.e., do not exceed the issue width.

2.4 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) forces 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 eace 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.

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

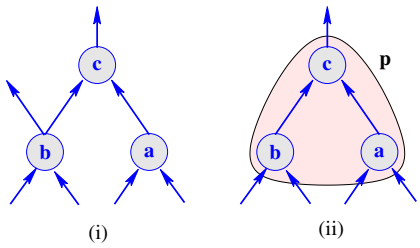

Fig. 1. Pattern coverage restrictions: (i) Pattern p cannot cover the set of nodes since there is an outgoing edge from b, (ii) pattern p covers the set of nodes $\{a, b, c\}$.

Our model forbids, as is the case for almost all architectures, to access a partial result that flows along a covered edge and thus appears inside a pattern. Only a value flowing out of a node matched by a root node of the matching pattern is accessible (and will be allocated some register). This situation is illustrated in Figure 1. A possible candidate pattern p that covers nodes `a`, `b`, and `c` cannot be selected in case (i) because the value of `b` is used by another node (outgoing edge from `b`). On the other hand, the pattern might be selected in case (ii) since the value represented by `b` is only internal to pattern p.

For that, 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'.

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

2.5 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(s) is active.

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

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.

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

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

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

2.6 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 assuring full coverage, Equation (1) constraints 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, see Equation (12), and the case of edges outgoing from singletons, see Equation (13).

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

Active edges leaving a node covered by a singleton pattern p carry always the latency L_p of p. Equation (13) assures that the schedule meets the latency constraint also for these cases.

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

2.7 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.

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

2.8 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 can be 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

$$\min \tau \tag{17}$$

3 Experimental results

First, we provide a theoretical VLIW architecture 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. Finally we summarize first results.

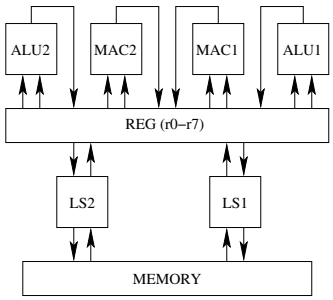

Fig. 2. Theoretical load/store VLIW target architecture used for the evaluation.

3.1 Target architecture

In order to compare OPTIMIST's DP technique to the ILP formulation of Section 2, we use a theoretical VLIW target platform (see Figure 2) with the following characteristics. The issue width is a maximum of three instructions per clock cycle. The architecture has two arithmetic and logical units (`ALU1` and `ALU2`). 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 two ALUs, the architecture has two multiply-and-accumulate units (`MAC1` and `MAC2`) that take two clock cycles to perform a multiply-and-accumulate operation. There are eight general purpose registers accessible from any unit. We assume a single memory bank with unlimited size. To store and load data there are two load/store units (`LS1` and `LS2`). The latency for a load or store operation is four clock cycles.

OPTIMIST is a retargetable framework, *i.e.* it can produce code for different target processors. A hardware description language (xADML), based on XML, parametrizes the code generator. An xADML document is divided into two parts: One part consists in declaring resources such as registers, memories modules and functional units. The other (and largest) part provides the instruction set for the specified target processor. The instruction set specification is (optionally) subdivided into two parts: (i) pattern definitions, and (ii) associations (mappings) between a pattern and a target processor instruction. An xADML specification contains thus a structural and behavioral description of a target processor.

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 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 [2]. As assumed in Section 2, we use for the ILP formulation the simpler model with unit occupation time and latency for each instruction. An extension of the ILP formulation to use general reservation tables is left to future work.

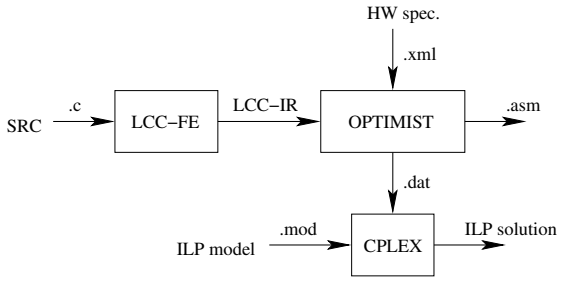

Fig. 3. Experimental setup.

3.2 Experimental setup

Figure 3 shows our experimental platform. We provide a plain C code sequence as input to OPTIMIST. We use LCC [3] (within OPTIMIST) as C front-end. Besides the source code we provide the description of the target architecture in xADML language (HW Spec). 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 2. The data file contains hardware specifications, such as the issue width of the processor, the set of functional units, patterns, etc. that are extracted from the hardware description document. It generates all parameters introduced in Section 2.3. Finally we use the CPLEX solver [4] to solve the set of equations.

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

3.3 First results

We generated code for a set of various basic blocks of different sizes. Most of them perform simple arithmetic computations taken from various digital signal processing filter algorithms. We run the evaluation of the dynamic programming approach on a Linux machine with an Athlon processor of 1.5GHz, with 1.5GB of RAM and 1.5GB of swap. The ILP evaluation was performed using CPLEX 6.5.1[1] on a 300 MHz UltraSparc 9 with 640MB of RAM and 1GB of swap space.

Table 1 reports our first results. For each case, the second column reports the number of nodes in the DFG for that basic block. The third and fourth column give the height of the DAG and the number of edges, respectively. Observe that

[1] By the time of writing, our first priority has been to test and evaluate the new ILP formulation. Since we do not (yet) have a CPLEX license at our department, the mathematical department kindly let us run our evaluation on their SPARC machines, running an old CPLEX version. We should expect better results with more modern versions of CPLEX and hardware.

Table 1. First results that compares ILP against DP approach to fully integrated code generation. An X in the table is set when the computation time exceeded 6 hours of equivalent computation time on machine running CPLEX solver.

				ILP		DP		
Case	$\|G\|$	Height	$\|E_G\|$	Time (h:mm'ss)	cc	Time (h:mm'ss)	TimeX5	cc
(a)	8	3	6	5'33	8	'01	'05	8
(b)	13	4	12	34'44	10	'20	1'40	10
(c)	14	6	14	1:04'58	13	'08	40	13
(d)	15	4	6	28'43	10	1'24	7'00	10
(e)	18	3	16	5'15	10	X	X	X
(f)	18	5	18	X	X	38'52	3:14'20	11
(g)	19	4	18	3:27'08	11	X	X	X
(h)	22	6	27	— out of memory —		1:16'17	6:21'25	17

the height corresponds to the longest path of the DFG in terms of number of DFG nodes, and not to its critical path length, whose calculation is unfeasible since the instruction selection is not yet known. The fifth column reports the computation time for finding an ILP solution, and in the sixth column we display the amount of clock cycles required for the basic block. We report the results for OPTIMIST in columns seven to nine. Since the Athlon processor is approximately five times faster than the Sparc processor available to us for running the ILP solver, we report in the TimeX5 column the theoretical computation time if OPTIMIST were run on the Ultra Sparc 9 machine. In the case when the equivalent computation time of the machine running the CPLEX solver exceeded 6 hours, the computation was abandoned and is represented with an X.

We should mention a factor that contributes in favor of the ILP formulation. In the OPTIMIST framework we enhanced the intermediate representation with extended basic blocks (which is not standard in LCC). As consequence, we introduced data dependence edges for resolving memory write/read precedence constraints. These are implicitly preserved in LCC since it splits basic blocks at writes and processes each basic block in sequence. In the current ILP formulation we consider only data flow dependences edges. Thus, we instrumented OPTIMIST to remove edges introduced by building extended basic blocks. As a result, the DAGs have a larger base, *i.e.* with larger number of leaves, and in general a lower height. We are aware that the DP approach suffers from DFGs with a large number of leaves, as OPTIMIST early generates a large number of partial solutions.[2]

First, for all cases where it was possible to check, we found optimal solutions with the same number of clock cycles, which was of course expected. We can see that for DFGs that are of rather vertical shape, DP performs much faster than

[2] For the test cases where we removed memory dependence edges, the resulting DFG may no longer be equivalent to the original C source code. It is however still valid to compare the ILP and DP techniques, since both formulations operate on exactly the same intermediate representation.

ILP, see cases (c), (f) and (h). A surprising result comes from case (e), where ILP produced a solution within several minutes while DP did not find a solution within 6 hours of equivalent CPU time of the CPLEX machine. The shape of the DFG in case (e) is a "flat" DAG where more than 50% of the total number of DFG nodes are leaf nodes. We can observe a similar shape for the case (g), where seven out of 19 nodes are leaf nodes. In case (h) we reach the limit of CPLEX (more precisely, of the AMPL preprocessor generating the ILP system). We are aware that the version of CPLEX that we currently have access to is rather old, and thus the computation time should be slightly better with a newer version.

It was unexpected to see the ILP formulation perform quite well for cases where the DP approach had problems. Hence, for flat-shaped DAGs we would prefer to use ILP, whereas for DFGs with larger height, we should opt for DP. For uncertain cases, we may consider to spawn simultaneously both variants and wait for the first answer.

4 Future work

The current ILP formulation lacks several features available in OPTIMIST framework. In this paper we considered and provided a target architecture that suits the ILP model. We will consider extending the formulation to handle cluster 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.

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.

5 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.* [11] 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 are not aware of any other ILP formulation in the literature that integrates all code generation phases into a single ILP model.

We compared the ILP formulation with our research framework for integrated code generation, OPTIMIST, which uses dynamic programming. We evaluated

both methods on a theoretical architecture that fitted the ILP model restrictions. Our first results show that both methods performs well in distinct cases: The dynamic programming approach of OPTIMIST is more suitable for DFGs with vertical shape and narrow bases (a small number of leaves). In contrast, ILP seems to perform better in the case of "flat" DFGs, with low height and large bases.

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 should evaluate the formulation on a more recent version of CPLEX than we did in this paper.

Acknowledgments We thank the mathematical department of Linköpings universitet for letting us use their CPLEX installation. This research was partially funded by the Ceniit programme of Linköpings universitet and by SSF RISE.

References

1. 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.
2. 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.
3. C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison Wesley, 1995.
4. I. Inc. CPLEX homepage. http://www.ilog.com/products/cplex/, 2005.
5. D. Kästner. *Retargetable Postpass Optimisations by Integer Linear Programming.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2000.
6. C. Kessler and A. Bednarski. Optimal integrated code generation for VLIW architectures. Accepted for publication in *Concurrency and Computation: Practice and Experience*, 2005.
7. C. Kessler and A. Bednarski. OPTIMIST. www.ida.liu.se/~chrke/optimist, 2005.
8. R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. *IEEE Transactions on VLSI Systems*, 5(1):112–122, 1997.
9. 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.
10. 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.
11. 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.
12. L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming.* PhD thesis, Technische Fakultät der Universität des Saarlandes, Saarbrücken (Germany), 1996.