ILP-based Instruction Scheduling for IA-64

Daniel Kästner Sebastian Winkel

Universität des Saarlandes / Fachrichtung 6.2. Informatik
Postfach 15 11 50 / D-66041 Saarbrücken / Germany
Phone: +49 681 302 5589 Fax: +49 681 302 3065
{kaestner,sewi}@cs.uni-sb.de

ABSTRACT

The IA-64 architecture has been designed as a synthesis of VLIW and superscalar design principles. It incorporates typical functionality known from embedded processors as multiply/accumulate units and SIMD operations for 3D graphics operations. In this paper we present an ILP formulation for the problem of instruction scheduling for IA-64. In order to obtain a feasible schedule it is necessary to model the data dependences, resource constraints as well as additional encoding restrictions—the bundling mechanism. These different aspects represent subproblems that are closely coupled which gives the motivation for a modeling based on integer linear programming. The presented approach is divided into two phases which allows us to compute mostly optimal solutions with acceptable computation time.

1. INTRODUCTION

The IA-64 architecture has been designed as a synthesis of VLIW and superscalar design principles. It incorporates typical functionality known from embedded processors as multiply/accumulate units and SIMD operations for 3D graphics operations. IA-64 is a statically scheduled architecture where the compiler is responsible for efficiently exploiting the available instruction-level parallelism and keep the execution units busy. Moreover the compiler has to explicitly take care of the density of the generated code. Due to the bundling mechanism of IA-64 this is a special resource allocation problem which has to be coupled with the scheduling proper in order to achieve good results.

Thus for IA-64 instruction scheduling, the phase coupling problem between the reordering of instructions, functional unit binding and bundle allocation (bundling) has to be solved. The efficiency of commonly used heuristic methods like list or trace scheduling is usually affected by such phase coupling effects. This gives the motivation for developing search-based methods, like the presented approach that is based on integer linear programming. Our ILP-modeling can be expected to produce optimal solutions. Currently the modeling is restricted to basic blocks, but acyclic control flow can be integrated via predication as well. Additionally, code motion between multiple basic blocks can be modeled as presented in [1]. Our scheduler can be adapted to every processor of the IA-64 architecture and is extendable to support upcoming EPIC architectures.

The concept of bundles employed in EPIC combines very simple issuing hardware with a high code density compared to VLIW. This makes this philosophy interesting for embedded systems, where simple hardware as well as the quality of the generated code is of high importance.

The paper is organized as follows: Sec. 1.1 gives a short introduction into integer linear programming and Sec. 1.2 presents the IA-64 architecture. After an overview of related work in Sec. 2, our ILP model is detailed in Sec. 3, Sec. 4 and Sec. 5. The results of our experimental evaluation are summarized in Sec. 6; Sec. 7 concludes and gives an outlook.

1.1 Integer Linear Programming

In the last decade, the use of integer programming models has increased significantly which is mostly due to the advances in algorithms for solving integer programs and the availability of reliable software packages [2]. Computing an optimal solution of an integer linear program is $\mathcal{NP}$-complete [3]. Nevertheless many large instances of such problems can be solved. Recent research has lead to an understanding of properties that make some ILP formulations easily solvable. Those formulations are called *structured* [4] since it is the structure of their constraints that permits efficient computations. Recent advances have also made it possible to improve the efficiency of ILP solving techniques by curtailing the necessary enumeration process [2, 5].

Let $P_F = \{x \mid Ax \geq b,\ x \in \mathbb{R}^n_+\}$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$. Then integer linear programming (ILP) is the following optimization problem:

$$\begin{aligned} \min \quad z_{IP} &= c^T x \\ x &\in P_F \cap \mathbb{Z}^n \end{aligned} \qquad (1)$$

The set P_F is called *feasible region.* If all integrality restrictions of equation (1) are removed the resulting linear program is called *LP-relaxation.* The feasible area P_F is called integral if it is equal to the convex hull P_I of the integer points ($P_I = conv(\{x \mid x \in P_F \cap \mathbb{Z}^n\})$; see Fig. 1). In this case, the optimal solution can be calculated in *polynomial time* by solving its LP-relaxation. Therefore, while formulating an integer linear program, one should attempt to find equality and inequation constraints such that P_F will be integral. It has been shown that for every bounded system of

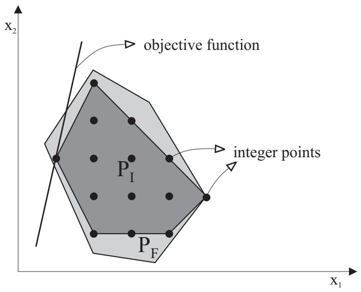

Figure 1: Feasible Areas.

rational inequalities there exists an integer polyhedron [6, 7]. Unfortunately for most problems it is not known how to formulate these additional inequalities—and there could be an exponential number of them [8].

In general, $P_I \subsetneq P_F$, and the LP-relaxation provides a lower bound on the objective function. The efficiency of many integer programming algorithms depends on the tightness of this bound. The better P_F approximates the feasible region P_I, the sharper is this bound. Thus for an efficient solution of an ILP-formulation, it is extremely important that P_F is close to P_I.

Depending on the choice of the main decision variables, integer linear programming models for the code generation problem can be classified as *time-indexed* formulations or *order-indexed* formulations [9]. In time-indexed approaches, the decision variables are based on the discrete point in time the modeled events are assigned to. The ordering of the events is derived from the assignment to points of time. In order-indexed approaches, the semantics of the decision variables reflects the ordering of the modeled events. Here the assignment of the events to points of time is derived from the computed ordering.

The implications of the modeling style are investigated in [1]. The study indicates that order-indexed modeling allows an efficient integration of the code generation problems instruction scheduling, functional unit allocation and register assignment. The order-indexed modeling is well suited for irregular architectures where the resource competition is high. Time-indexed modeling presents itself for architectures with a high degree of instruction-level parallelism where a large number of functional unit types is available for the execution of each microoperation.

1.2 The IA-64 Architecture

The IA-64 instruction set architecture [10, 11, 12] has been designed for high-performance applications, like processing multimedia data streams, 3D graphics, and executing numerically intensive computations. IA-64 is an EPIC architecture (*Explicitly Parallel Instruction Computing*) that relies on the compiler to exploit the available instruction-level parallelism.

The first implementation of the IA-64 architecture is the Intel Itanium processor [11, 12]. The Itanium provides a 10-stage in-order execution pipeline; all functional units are fully pipelined to achieve a high throughput. The Itanium contains four integer ALUs, four multimedia ALUs, two extended-precision floating-point ALUs, two single-precision floating-point units, two load/store units, and three branch units. The floating-point units are implemented as MAC-units that can execute one multiply/add operation per clock cycle. This is important e. g. for digital signal processing or 3D graphics applications. The functional units are classified as I (integer), F (floating-point), M (memory), or B-type (branch) units. The data paths include 128 general purpose registers, 128 floating-point registers, 64 predicate registers, and 8 branch registers. The Itanium supports branch prediction, predicated execution as well as control and data speculation.

Each IA-64 instruction is categorized into one of 6 types: Integer ALU (A-type), Non-ALU integer (I-type), Memory (M-type), Floating-point (F-type), Branch (B-type), Extended (L/X-type). Each instruction type may be executed on one or more execution unit types. A-type instructions in IA-64 can be scheduled to execute on either M- or I-units. The execution times of instructions range from 1 to 24 cycles, except for loads from memory missing the cache and the check instructions (used for speculation) which can take more than 50 cycles.

IA-64 uses a 128-bit encoding, called a *bundle*, that contains three 41-bit instructions and a 5-bit template field. The template bits help decode and route instructions and indicate the location of stops that mark the end of *groups* of instructions that can execute in parallel. The process of sending instructions to functional units is called *dispersal*; the dispersal windows can hold two bundles. Thus the Itanium can issue a maximum of six instructions per clock cycle. These six slots can contain at most 2 I-type, 2 M-type, 2 F-type, and 3 B-type instructions. The specific functional unit to which an instruction is sent is determined by its type and its position within the current set of instructions being issued.

In addition to the location of stops the template field specifies the mapping of instruction slots to execution unit types. Not all possible mappings of instruction slots to unit types are available. There are 12 basic template types (bundle types): MII, MI_I, MLX, MMI, M_MI, MFI, MMF, MIB, MBB, BBB, MMB, and MFB (stops are denoted by _). Each basic template has two versions: one with a stop after the third slot and one without. Bundle boundaries have no direct correlation with instruction group boundaries as instruction groups can extend over an arbitrary number of bundles. Instruction groups begin and end where stops are set in assembly code and dynamically whenever a branch is taken or a stop is encountered.

2. RELATED WORK

Search-based algorithms can be profitably used for code generation problems when a very high code quality is required, or when there are different phases that interact with each other. While heuristic methods typically require shorter computation times, search-based methods can usually produce a higher solution quality.

Early approaches for ILP-based code generation have been developed in the area of architectural synthesis. The ILP model used in the ALPS synthesizer [13, 14] is a time-indexed formulation for instruction scheduling and functional unit allocation. In [15, 6] the ALPS model is improved by exploiting results of polyhedral theory. The description of the feasible region is tightened and the solution efficiency increased. The resulting formulation has been implemented in the OASIC synthesizer and has been extended to incorporate the register assignment problem. In [1] the OASIC model is extended to the problem of code generation for irregular architectures. While all previously mentioned ILP formulations are time-indexed formulations, in [16] an order-indexed ILP formulation for instruction scheduling, functional unit allocation and register assignment is presented.

There have been only few approaches to incorporate ILP-based methods into the code generation process of a compiler. An early approach for local ILP-based instruction scheduling for vector processors has been presented in [17]. In the RECORD compiler [18], integer linear programming is used to model local instruction scheduling and a variant of functional unit allocation. Both ILP models are time-indexed formulations similar to the OASIC model [15, 6].

Wilson et al. [19] use an ILP-formulation for simultaneously performing code selection, scheduling, register allocation and assignment. The complexity of the resulting formulations however leads to prohibitive computation times. In [20] it is demonstrated that by using a well-structured ILP formulation, an optimal local instruction scheduling of large basic blocks for regular architectures can be computed in acceptable time.

The problem of phase-coupled code generation for irregular architectures is investigated in [1]. Extensions of the SILP- and OASIC models ([16, 15, 6]) are developed that allow a generic modeling of irregular hardware constraints and make the simultaneous optimization of several basic blocks, e. g. nested loops, possible. The extended modeling has been implemented in the PROPAN framework [1]. In [1] also ILP-based approximations are developed that allow to compute high-quality solutions in drastically reduced time compared to the exact solution. The modeling of [1] presents one of the starting points of our work.

The implementation of the global code scheduler in Intel's compiler for the IA-64 architecture is presented in [21].

3. THE ILP MODEL

The problem of optimal IA-64 instruction scheduling can be subdivided into two phases, as shown in Fig. 2. In the remainder of this paper we will refer to the first phase as *macro-scheduling* and to the second phase as *bundling*; the result of both phases represents the final schedule.

The goal of the optimization is to achieve a schedule that is time optimal, i. e. that requires minimal execution time, and to select among the time optimal schedules one that is space optimal, i. e. that consumes minimal program memory space.

In the macro-scheduling phase, a minimal-length preliminary schedule is computed where each instruction is assigned to an instruction group. The result is a sequence of instruction groups so that one group can be executed per clock cycle without violating data dependences or resource constraints.

Since no encoding restrictions are taken into account, this sequence does not correspond to a valid schedule yet. Therefore in the bundling phase an optimal bundle sequence is computed from the sequence of instruction groups which represents the final schedule.

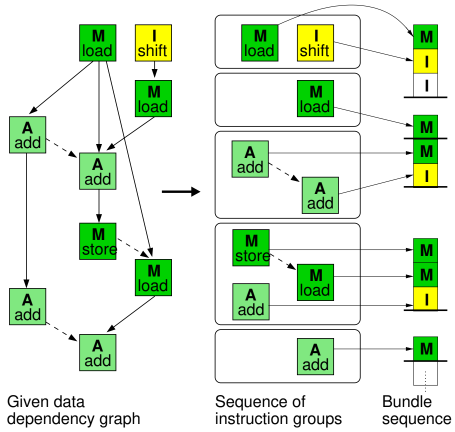

Figure 2: IA-64 instruction scheduling – overview.

Both phases could also be integrated into a single ILP formulation. For complexity reasons however such a unified formulation—a tentative modeling required 26 classes of constraints—cannot be considered promising. Moreover we will show in the following that the two-stage approach can be expected to produce an optimal solution wrt. the unified formulation in most cases. This allows efficient computations while usually achieving optimality.

4. MACRO-SCHEDULING

Macro-scheduling defines the order in which the instructions are executed. Each instruction is assigned to an *instruction group* so that all instructions from one group can be executed simultaneously in a single *clock cycle*. Thus in this section we treat both terms as synonymous. As a basis for our modeling we use the time-indexed OASIC formulation [15, 6] which is a structured model that is well suited for architectures with a high degree of instruction-level parallelism [1].

The main decision variables are binary variables called x_{nt}^k where a value of 1 means that instruction n is executed in clock cycle t on execution unit k. The index k of the decision variables x_{nt}^k is only relevant for A-type instructions since those can be executed on M- and I-units. For all other instructions the assignment to execution unit types is unique.

Let V denote the set of instructions from the input program. Before generating the integer linear programs, an interval is calculated for each instruction n containing all clock cycles in which the execution of n can be started in any feasible schedule. This interval is defined as $N(n) = \{asap(n), \ldots alap(n)\}$. The *asap* (*alap*) control step is the earliest (latest) control step in which n can be started without violating any data dependences. The interval $N(n)$ is refined using information about the availability of hardware resources [1].

The macro-scheduling polytope is composed of three types of constraints which will be presented in the following.

Let $R(n)$ denote the set of execution unit types instruction n can be assigned to. The *assignment constraints* ensure that every instruction is executed exactly once by exactly one execution resource:

$$\sum_{k \in R(n)} \sum_{t \in N(n)} x_{nt}^k = 1 \qquad \forall\, n \in V \tag{2}$$

The *precedence constraints* model the data dependences of the input program. The edges E_D of the data dependence graph $G_D = (V, E_D)$ can be classified into two categories: *weak* and *strong* dependences, i. e. $E_D = E_D^w \cup E_D^s$.

The weak dependences E_D^w include anti dependences (WAR) and all dependences with respect to memory.

The strong dependences E_D^s are composed of the true and output dependences with respect to register resources. Weak dependences are allowed inside an instruction group, strong dependences not. In the bundling phase, only the weak dependences have to be considered (see Sec. 5).

Let w_{mn} denote for each data dependence $(m, n) \in E_D$ the minimal number of cycles that must pass between the starting time of m and the starting time of n for the dependence to be respected. Then the following constraint is generated:

$$\sum_{k} \sum_{\substack{t_n \le t \\ t_n \in N(n)}} x_{nt_n}^k + \sum_{k} \sum_{\substack{t_m \ge t - w_{mn}+1 \\ t_m \in N(m)}} x_{mt_m}^k \le 1$$
$$\forall\, t \in \{t' + w_{mn} - 1 | t' \in N(m)\} \cap N(n), \qquad \forall\, (m, n) \in E_D \tag{3}$$

The precedence constraints exclude any ordering of instructions where data dependences are violated. In [1] it has been shown that any infeasible ordering is excluded but no feasible solution is discarded.

Equation (3) can be extended to deal with the case where the execution time of an instruction varies with the execution unit type it is scheduled to. For example, on the Itanium, an address generation in the ALU takes one cycle more if scheduled to an I-unit instead of a M-unit. Intuitively we model this restriction by adding an additional variable $x_{m(t-1)}^I$ to the left-hand side of equation (3).

The polytope developed so far is integral [4]. This changes, however, if we add the resource constraints: resource-constrained scheduling is $\mathcal{NP}$-complete.

Let R_k denote the number of execution units of type k available in the target processor and let U be a pre-calculated upper bound on the number of clock cycles required to execute the input program. Then the *resource constraints* prevent more than R_k instructions from being assigned to each functional unit type k at the same clock cycle.

$$\sum_{\substack{n \in V: \\ k \in R(n)}} x_{nt}^k \le R_k \qquad \forall\, k \,\wedge\, 1 \le t \le U \tag{4}$$

The execution units on the Itanium are asymmetric, i. e. some instructions like fixed shifts can only be executed on the first of several same-type units. We model this by generating additional instances of equation (4) for these instructions with the right-hand side set to one.

A dedicated resource constraint is added for the dispersal window. With a dispersal window width of d bundles at most $3d$ instructions can be started per clock cycle, e. g. $3 \cdot 2 = 6$ instructions on Itanium.

$$\sum_{k} \sum_{n \in V} x_{nt}^k \le 3d \qquad \forall\, 1 \le t \le U \tag{5}$$

Now every integer point saturating the constraints corresponds to a feasible solution of the macro-scheduling problem.

Our goal is to find a schedule of minimal length T; the value of T is defined by constraints (7), the objective function is given in equation (6).

$$\min \quad T \tag{6}$$

$$\sum_{k} \sum_{t \in N(n)} t x_{nt}^k \;\le\; T \qquad \forall\, n \in V \tag{7}$$

We extend this objective function to deal with a weakness of this time-based ILP model. This weakness concerns instructions with execution times spanning several clock cycles as, e. g. , load instructions. An example is shown in Fig. 3.

Cyc.	time-optimal	+space-optimal
0	ld8 r1=[r2] ;;	ld8 r1=[r2] ;;
1	I1 ;;	I1 I2 I3 ;;
2	I2 ;;	-
3	I3 ;;	-
4	add r3=r1,r3	add r3=r1,r3

Figure 3: A gap produced by a load instruction with 4-cycle execution time.

Often there are not enough candidates available to fill the gap between the definition and use completely with independent instructions. When placing independent instructions into a gap, care should be taken to produce dense code, i. e. to use as few instruction groups as possible. This is shown in both columns of the example with the independent instructions `I1`, `I2` and `I3`. There are several reasons for this:

In contrast to VLIW architectures it is not necessary to insert *nop*-instructions to ensure the correct distance between definitions and uses if there are not enough intermediate instructions. The IA-64 architectures automatically stalls if an operand is not yet available. In consequence empty instruction groups can be omitted and thus do not consume program memory space.

Second, our experiments indicate that the smaller the instruction groups are, the more likely it is that additional *nops* have to be inserted into the bundles computed later. Therefore fewer and thereby larger instruction groups are preferred over smaller ones.

A third reason is that the execution time of loads depends on where the access hits in the cache hierarchy. In our future work we will use a static cache analysis [22], possibly supplemented by profiling information, as a basis for deciding which access time should be assumed when generating the ILP. If during code generation a L2-hit is assumed but during actual program execution a L1-hit occurs this would not lead to stalls in the second case of Fig. 3 while the execution time of the first case would be the same as for a L1-miss.

So far our objective function (6) does not take the issue of generating dense code into account. Therefore we extend it to first minimize the execution time with high priority and then with lower priority the number of instruction groups used. For this we introduce dedicated binary variables u_t where a value of 1 indicates that at least one instruction is scheduled to clock cycle t. The variables are defined by the following constraint:

$$x_{nt}^k \leq u_t \qquad \forall n \in V \quad \forall t \in N(n) \; \forall k$$

The modified objective function (8) minimizes T as primary goal and the number of used instruction groups $\sum_{t=1}^{U} u_t$ as secondary goal. This is achieved by weighting T with the constant U (see Sec. 4).

$$\min \quad U \cdot T + \sum_{t=1}^{U} u_t \tag{8}$$

This completes our ILP model for macro-scheduling; it uses $\mathcal{O}(|V|^2)$ binary variables and requires $\mathcal{O}(|V|^3)$ constraints.

5. BUNDLING

The result of the preceding phase corresponds to an optimal schedule if no encoding restrictions (bundling restrictions) are taken into account and the assumptions about the cache behavior (see above) are fulfilled. Each instruction group is given as a set of instructions with their weak dependences. Now it is the task of the bundling phase to transform the sequence of instruction groups into an efficient sequence of feasible IA-64 bundles.

There are some cases where our bundling algorithm can be forced to introduce additional penalties that cannot be considered in the macro-scheduling phase:

1. For every cycle, the number of bundles processed is limited by the size of the dispersal window d. If the instruction group exceeds d bundles, a penalty cycle occurs which we call *dispersal split*. The ILP formulation presented in Sec. 4 guarantees that enough resources are available and that at most $3d$ instructions can be contained in an instruction group. However, due to the bundling restrictions it may be necessary to generate more than d bundles (see below).

2. Empty slots in a bundle have to be filled with *nops* that have to be assigned to a specific resource type. Choosing an inappropriate *nop* type may cause the number of available resources to be exceeded so that a one-cycle penalty called *resource split* occurs. This is illustrated in Fig. 4 for the Itanium with its two M- and I-units.

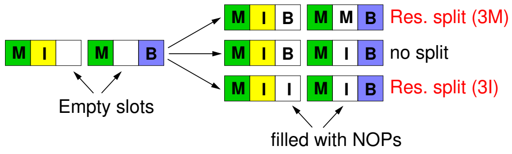

Figure 4: Choosing an inappropriate *nop* type may trigger a resource split (*nops* are depicted with a white background).

In the following we will present examples for instruction groups triggering a resource resp. dispersal split on the Itanium.

5.1 Resource Split

Fig. 5 shows an instruction group where the order `IIMM` is imposed by three weak dependences. Because in every bundle type an M-type slot precedes an I-type slot, an M-type *nop* must be placed before one of the first two I-type instructions. Then three M-type instructions trigger a resource split on the Itanium.

unit type	instruction
I	`shr r1=r2,r3`
I	`shr r2=r4,r5`
M	`ld8 r4=[r8]`
M	`st8 [r8]=r5`

Figure 5: Instruction group triggering a resource split.

5.2 Dispersal Split

A similar example is shown in Fig. 6. Because in no bundle type an M-type instruction can be placed after an I-type instruction, the first, second and fourth instruction have to be in different bundles – but more than two bundles trigger a dispersal split on the Itanium.

unit type	instruction
I	`shr r1=r2,r3`
M	`ld8 r2=[r8]`
I	`shr r8=r4,r5`
M	`ld8 r4=[r9]`

Figure 6: Instruction group triggering a dispersal split.

It is exactly here that the two-phase approach may lose optimality because a decision made in the first phase cannot be reversed (*phase coupling problem*). In an exact approach such unfavorable instruction groups could be detected and eventually excluded.

Instruction groups with weak dependences like in these two constructed examples can be expected to occur very rarely in practice. We did not encounter such a case in our experiments with more than 5000 instruction groups.

Now let us explain our approach to bundling. For complexity reasons we divide it into two subphases: *micro-scheduling* and *sequencing*. Micro-scheduling generates a separate bundle sequence for each single instruction group as depicted in

Fig. 7. Then sequencing combines these partial sequences into an optimal bundle sequence for the whole program.

5.3 Micro-Scheduling

Given is a partition of V into max_g instruction groups V_g, $\bigcup_{g=1}^{max_g} V_g = V$. For each of these groups we solve an integer linear program that minimizes the number of splits (see below). The micro-scheduling polytope is composed from 7 constraint types and three classes of binary variables. Due to the resource limitations the instruction groups computed in the macro-scheduling phase contain only few instructions so that the solution can be usually computed within less than one second. The main decision variables are binary variables similar to those of Sec. 4:

$$x_{ij}^n = 1 \Leftrightarrow \quad \text{Instruction } n \text{ is placed in slot } j \text{ of bundle } i.$$

We assume that $c = \left\lceil \frac{1}{3} |V_g| \right\rceil$ bundles are sufficient to incorporate all instructions of group g and generate a variable x_{ij}^n for each pair $(i,j) \in P := \{1, \ldots c\} \times \{1, \ldots, 3\}$. If the resulting ILP is infeasible, we increase c by one and iterate. Moreover we assume that no more than $2d$ bundles have to be generated for any instruction group, i. e. $c \leq 2d$.

Each instruction has to be placed into exactly one slot of exactly one bundle. This is ensured by the *assignment constraints*:

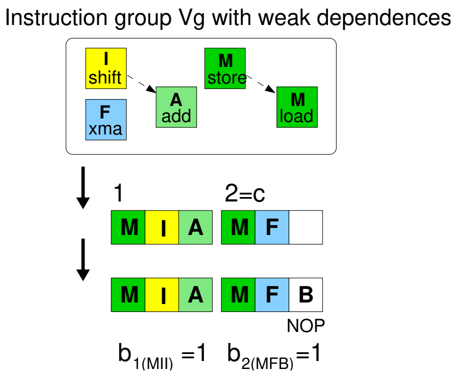

Figure 7: Micro-Scheduling in detail.

$$\sum_{(i,j) \in P} x_{ij}^n = 1 \qquad \forall\, n \in V_g \tag{9}$$

It is feasible to schedule two instructions between which a weak dependence exists into the same instruction group, i. e. $(m,n) \in E_D^w$. However it has to be ensured that m appears before n in the bundles of the group. In order to model this we introduce a relation $\prec \subset P^2$ that specifies the ordering of bundle positions.

$$(i',j') \prec (i,j) \Leftrightarrow (i' < i) \vee ((i' = i) \wedge (j' < j))$$

Then the following *precedence constraints* ensure that no weak dependence is violated:

$$\sum_{(i,j) \preceq s} x_{ij}^n \quad + \sum_{(i,j) \succeq s} x_{ij}^m \leq 1 \tag{10}$$
$$\forall \quad (m,n) \in (E_D^w \cap V_g^2) \quad \forall s \in P$$

Modified *resource constraints* ensure that no more than one instruction is placed in one slot:

$$\sum_{n \in V_g} x_{ij}^n \leq 1 \qquad \forall (i,j) \in P \tag{11}$$

Additional constraints are required for enforcing the bundle type restrictions. Let $\mathcal{U} = \{\mathtt{M}, \mathtt{I}, \mathtt{B}, \mathtt{F}\}$ be the set of functional unit types and $\mathcal{T} \subset \mathcal{U}^3$ be the set of feasible bundle types:

$$\mathcal{T} = \{\mathtt{MII}, \mathtt{MMI}, \mathtt{MIB}, \mathtt{MBB}, \mathtt{BBB}, \mathtt{MMB}, \mathtt{MFI}, \mathtt{MMF}, \mathtt{MFB}\}$$

We construct the complement of $\mathcal{T}$ as the set $\overline{\mathcal{T}}$ which contains all infeasible bundle types described by triples (t_1, t_2, t_3) of sets of instruction types $t_i \subseteq \mathcal{I} = \{\mathtt{M}, \mathtt{I}, \mathtt{B}, \mathtt{F}, \mathtt{A}\}$):

t_1	t_2	t_3
{I}	*	*
{F}	*	*
*	*	{M}
{B}	{A,M,I,F}	*
{B}	*	{A,M,I,F}
*	{I}	{F}
*	{B}	{A,M,I,F}
*	{F}	{F}

Figure 8: Elements $t = (t_1, t_2, t_3)$ of the set $\overline{\mathcal{T}}$ of forbidden instruction type combinations ($*$ stands for the set $\mathcal{I}$).

For every $t = (t_1, t_2, t_3) \in \overline{\mathcal{T}}$ let $D(t) = \{j \in \{1,2,3\} \mid t_j \neq *\}$ and let $type(n)$ denote the type of instruction n. Then the following constraint excludes any infeasible combinations of instruction types:

$$\sum_{j \in D(t)} \sum_{\substack{n \in V: \\ type(n) \in t_j}} x_{ij}^n \quad \leq \quad |D(t)| - 1$$
$$\forall\, 1 \leq i \leq c \quad \forall \quad t \in \overline{\mathcal{T}} \tag{12}$$

Every solution to the ILP as presented up to now corresponds to an instruction sequence which can be completed to a feasible bundle sequence by possibly filling some empty slots with *nops* (see Fig. 7). For each *nop* an appropriate resource type has to be determined (see Fig. 4). The *nop* type selection is integrated into the ILP by assigning a bundle type to each of the c bundles. This requires introducing an additional type of binary decision variables:

$$b_{it} = 1 \Leftrightarrow \quad \text{Bundle type } t \in \mathcal{T} \text{ is assigned to bundle } i \ (1 \leq i \leq c)$$

Instruction groups are delimited by stops. A stop can always be placed after the third slot of each bundle type. The two bundle types MII resp. MMI additionally allow a stop after the second (MI_I) resp. the first (M_MI) slot. Only the slots after resp. before the stop may be filled with instructions at the beginning resp. the end of the bundle sequence. This is achieved by extending the set $\mathcal{U}$ by a void slot type $-$ and introducing additional bundle types:

$$\mathcal{T}_E = \{\mathtt{MI-}, \mathtt{M--}\}, \mathcal{T}_B = \{\mathtt{--I}, \mathtt{-MI}\}$$

$$\mathcal{T}^- = \mathcal{T} \cup \mathcal{T}_E \cup \mathcal{T}_B$$

We create additional instances of the variable b_{it} which permit these bundle types at the beginning and the end of the instruction group:

$$b_{1t} \ \forall\ t \in T_B \qquad b_{ct} \ \forall\ t \in T_E$$

The *instruction type constraints* integrate the b_{it} variables into the ILP:

$$3 \cdot (1 - b_{it}) \quad \geq \quad \sum_{j=1}^{3} \sum_{\substack{n \in V_g: \\ t_j \notin R(n)}} x_{ij}^n$$

$$\forall\ 1 \leq i \leq c \quad \forall \quad t = (t_1, t_2, t_3) \in \mathcal{T}^- \tag{13}$$

If bundle type t is assigned to bundle type i, then the left-hand side of inequation (13) is equal to 0 and the constraint is activated. Then the right-hand side implies that in no slot of bundle i an instruction of inappropriate type is placed.

The *bundle type constraints* ensure that exactly one bundle type is assigned to each bundle:

$$\sum_{t \in T} b_{it} = 1 \qquad \forall 1 \leq i \leq c \tag{14}$$

Now the ILP model is almost complete – every integer point satisfying the constraints corresponds to a feasible bundle sequence for the instruction group. However, some of these possible sequences may trigger a split on the target processor. Therefore we must find a way how to measure if one or more dispersal / resource splits occur and avoid them.

We must consider that, in the cycle after a split, execution restarts immediately behind in the next slot and may lead to another split. This is represented by the following binary variables ($\forall (i,j) \in P$):

$s_{ij}^l{=}1 \Leftrightarrow$	At least one split occurs anywhere before (i,j) (excl.)
$s_{ij}^h{=}1 \Leftrightarrow$	At least one split occurs after (i,j) if execution (re)starts there (incl.)

The objective function of the micro-scheduling phase minimizes a variable S that measures the number of splits (using constraints (16)):

$$\min \quad S \tag{15}$$

$$s_{ij}^l + s_{ij}^h \quad \leq \quad S \qquad \forall (i,j) \in P \tag{16}$$

It remains to show that S corresponds to the number of splits σ. This follows directly from the two following propositions. The proofs are given in appendix A.

Proposition 1: Let $(i,j) \prec (i',j')$, then

a) $s_{ij}^l = 1 \quad \Rightarrow \quad s_{i'j'}^l = 1$
b) $s_{ij}^h = 0 \quad \Rightarrow \quad s_{i'j'}^h = 0$

Proposition 2: The following relations hold between S and σ:

a) $\sigma = 0 \quad \Leftrightarrow \quad S = 0$
b) $\sigma = 1 \quad \Leftrightarrow \quad S = 1$
c) $\sigma \geq 2 \quad \Leftrightarrow \quad S = 2$

A dispersal split must always occur after d bundles have been processed. Thus it is obvious that if $i \geq d+1$, then $s_{ij}^l = 1$ must hold, and that $i \leq c - d$ implies $s_{ij}^h = 1$. Thus in the ILP formulation these variables are replaced by 1.

The next constraints show how resource splits are detected ($\forall X \in U$):

$$\sum_{m=1}^{i-1} \left(\sum_{t \in T} |\{n \,|t_n = X\}|\, b_{mt} \right) +$$

$$\sum_{t \in T} |\{n < j \,|t_n = X\}|\, b_{it} - R_X \leq c_1 s_{ij}^l$$

$$\forall (i,j) \in P : (c-d, 3) \preceq (i,j) \preceq (d, 3) \tag{17}$$

We set $c_1 = i \max_{t \in T} \{|\{i|\, t_i = X\}|\} - R_X$ to obtain a tight inequation (and skip the constraint if $c_1 \leq 0$). The formula for s_{ij}^h is similar.

This completes our ILP model for micro-scheduling; it uses $\mathcal{O}(|V_g|^2)$ binary variables and requires $\mathcal{O}(|V_g|^3)$ constraints.

5.4 Sequencing

The result of the ILP for micro-scheduling as presented above is an optimal bundle sequence for each instruction group. These partial bundle sequences still have to be combined to form the final schedule. This cannot be done in a naive way since it is not guaranteed that the bundle sequences fit together. If one instruction group ends after slot i within a bundle ($i \leq 2$), the next instruction group must begin in slot $i+1$ of the same bundle. Due to the small size of the generated ILPs we can afford to compute the optimal bundle sequences for each possible start slot $i_s \in \{1,2,3\}$ and each possible end slot $i_e \in \{1,2,3\}$. Thereby we leave all possibilities for later sequencing open.

We achieve this by solving nine ILPs, each containing one of the following constraints for the beginning

$$b_{1(\mathtt{--I})} := 1,\ b_{1(\mathtt{-MI})} := 1,\ b_{1(\mathtt{--I})} + b_{1(\mathtt{-MI})} = 0$$

and one of the following for the ending:

$$b_{c(\mathtt{MI-})} := 1,\ b_{c(\mathtt{M--})} := 1,\ b_{c(\mathtt{MI-})} + b_{c(\mathtt{M--})} = 0$$

This produces nine partial sequences for each instruction group as shown in an example in Fig. 9. These fragments

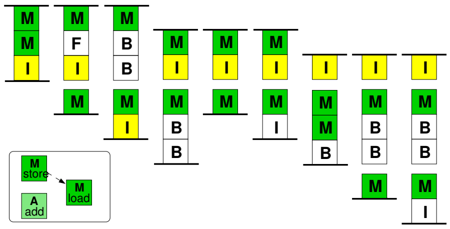

Figure 9: An example of the nine possible bundle sequences for an instruction group; *nops* are white.

can be chained together to form a global bundle sequence. We describe this by the following notation:

$S_{xy}(g,h) =$ The set of all possible bundle sequences from group g (including), starting before slot x, to group h (including), ending before slot y

We define $o_{xy}(g,h) \in S_{xy}(g,h)$ as the *optimal* bundle sequence in this set, where optimality is (constructively) defined below. For each sequence s we further use:

$|s| =$ The number of bundles in in this sequence

$\sigma(s) =$ The number of splits this sequence triggers

Let $s \oplus t$ denote the concatenation of sequences s and t, then the following equations hold:

$$\sigma(s \oplus t) = \sigma(s) + \sigma(t) \tag{18}$$

$$|s_{xu}(g,i) \oplus s_{uy}(i+1,h)| = \tag{19}$$

$$|s_{xu}(g,i)| + |s_{uy}(i+1,h)| - \delta_u$$

$$\delta_1 = \delta_2 = 1 \qquad \delta_0 = 0$$

The problem of sequencing now can be described as follows: Given from the previous phase are the optimal partial bundle sequences $o_{xy}(g,g), \forall x,y \in \{0,1,2\}, \forall g : 1 \leq g \leq max_g$. Needed is the optimal global bundle sequence $o_{00}(1, max_g)$.

We use *dynamic programming* to construct larger optimal sequences from smaller ones *bottom-up*.

To construct a sequence $o_{xy}(g,h)$ of the length $h - g + 1$ (instruction groups), we can assume that the smaller optimal sequences already have been computed. We use them to form a set of all possible candidates $O_{xy}(g,h) \subseteq S_{xy}(g,h)$:

$$O_{xy}(g,h) = \{o_{xu}(g,i) \oplus o_{uy}(i+1,h) \mid$$

$$g \leq i < h, u \in \{0,1,2\}\}$$

We collect the time-optimal ones of these candidates in the subset $O^T_{xy}(g,h) \subseteq O_{xy}(g,h)$:

$$O^T_{xy}(g,h) = \{s \in O_{xy}(g,h) \mid \sigma(s) = t_{min}\}$$

$$t_{min} = \min \{\sigma(s) \mid s \in O_{xy}(g,h)\}$$

We again form a subset $O^{TB}_{xy}(g,h) \subseteq O^T_{xy}(g,h)$ which contains these sequences with the fewest bundles:

$$O^{TB}_{xy}(g,h) = \left\{ s \in O^T_{xy}(g,h) \mid |s| = b_{min} \right\}$$

$$b_{min} = \min \left\{ |s| \mid s \in O^T_{xy}(g,h) \right\}$$

Then we chose a sequence from $O^{TB}_{xy}(g,h)$ as $o_{xy}(g,h)$ and repeat until we have obtained the optimal global bundle sequence $o_{00}(1, max_g)$ in $\mathcal{O}(max_g^2)$. In most cases, we also have $D(o_{00}(1, max_g)) = 0$ which indicates that no additional splits have been introduced in the second phase. This gives the proof of optimality.

6. EXPERIMENTAL EVALUATION

We have implemented and tested all ILP formulations presented in this paper; then we have compared the length and code size of the computed schedules with the results of heuristic methods.

For this we generated a pool of IA-64 assembly files using Intel's C++ Compiler 5.0 for Itanium (see Tab. 1). The first six files in the table are taken from SpecInt95, sim.asm is a benchmark program employing dynamic programming, layer.asm is taken from the MPEG audio decoder MPG123, and ccernel.asm contains the Livermore loops.

Program	BB	Instructions	Bundles	RP
g2.asm (go)	84	1165	481	0.84
g2s3.asm (go)	39	621	238	2.05
g2s2.asm (go)	140	2224	926	0.81
g29.asm (go)	244	3193	1338	0.84
g25.asm (go)	807	11407	4995	0.77
compress95.asm	45	419	808	0.94
sim.asm	164	2600	1046	0.93
layer.asm	154	1643	183	1.13
ccernel.asm	110	1543	803	1.29

Table 1: Input files for benchmarking. BB gives the number of basic blocks.

Then all basic blocks with more than two bundles were extracted from the assembly files, parsed with the PROPAN-generated assembly parser [1] and first scheduled with a heuristic method. We employed list-scheduling with highest level first heuristic. If the priority values of two instructions were equal, I-type and M-type instructions were preferred over A-type ones.

We computed two lower bounds for the schedule length: First, the minimum schedule length enforced only by the resource constraints, then the minimum length enforced only by the precedence constraints (i. e. critical path length). By dividing the first value by the second, we obtain a measure for the resource shortage when scheduling this basic block. We call this metric *resource pressure* and give average values in the last column of Tab. 1.

Using these lower bounds, we found out that 87% of all blocks were already optimally scheduled by list scheduling. The remaining blocks were passed to the ILP-solver, which could find a better schedule for 33% of them. So the total percentage of improved blocks is 4% on the average.

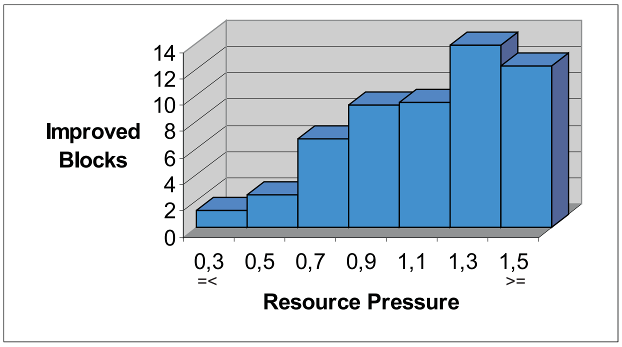

Figure 10: Percentage of improved blocks compared to resource pressure.

We analyzed how the result changes with increased parallelism. Therefore we duplicated the instructions in each basic block and predicated the copy and the original with complementary predicate registers. This process is similar to if-conversion [23, 24] and increases the available parallelism and thereby the resource pressure. After this change, the total percentage of improved blocks doubled to 10%. The distribution depicted in Fig. 10 shows how it grows with the resource pressure.

Although the generated integer linear programs often employed hundreds of binary variables, a solution could always be computed within seconds (see Fig. 11). These results show the strength of the well-structured ILP formulation we chose as a basis for our work.

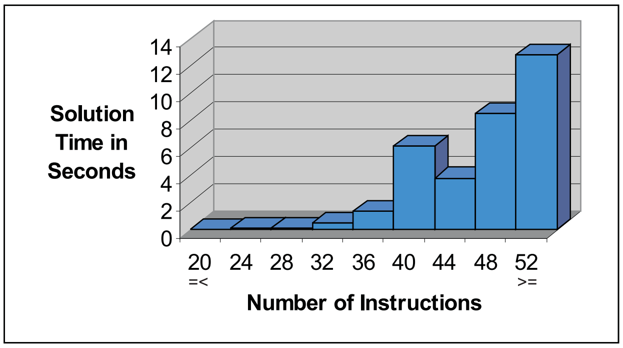

Figure 11: Average solution times of the macro-scheduling ILPs.

To evaluate the bundling phase, we compared the computed sequences directly with the code produced by Intel's compiler. The results in Fig. 12 show that our method produces sequences with ca. 25% fewer *nops*. The total code size shrinks by 6% on average. Note that some of the *nops* are inherent to the IA-64 architecture. For example, a basic block with n instructions contains at least n mod 3 *nops* because blocks always start and end at bundle boundaries. This effect alone contributes to 30% of the *nops* in the optimized bundle sequences.

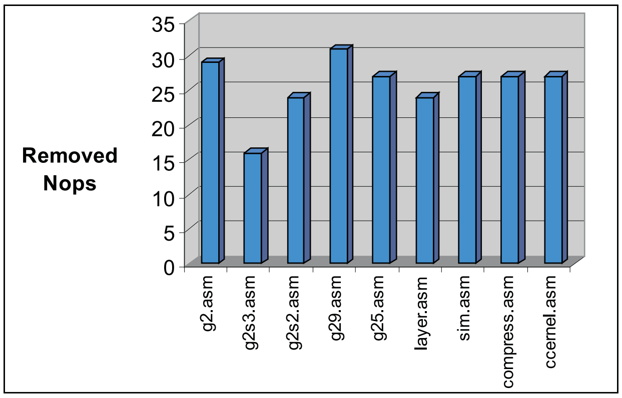

Figure 12: Percentage of removed *nops* from the output of Intel's compiler.

7. CONCLUSION AND OUTLOOK

We have presented an ILP formulation for the problem of IA-64 instruction scheduling. For complexity reasons the modeling is subdivided into two phases, macro-scheduling and bundling. This allows to compute mostly optimal solutions within seconds. The scheduler can be adapted to every processor of the IA-64 architecture and is extendable to support upcoming EPIC architectures. Experimental results indicate that our approach is suitable for small to medium sized code sequences as, e. g. , inner loops where small improvements already may have a major impact on performance.

For example, Intel proposes an integer programming framework to optimize the software library for computing transcendental functions on IA-64 [25].

The modeling currently is restricted to basic blocks, but control flow can be integrated using predication. If the cache behavior at certain program points can be statically predicted this knowledge can be used to generate more efficient code. For this we will employ the static cache behavior analysis of [22].

We see the presented ILP formulation as a basis which can be extended to incorporate additional compiler decisions crucial to IA-64 performance. Therefore we are currently integrating control and data speculation as well as code motion into the model.

APPENDIX

A. PROOFS

Proof: (Proposition 1)

a) If a split occurs when executing up to (i, j), then this split certainly also occurs when executing even more instructions. b) analogous. □

Proof: (Proposition 2)

a) $\sigma = 0 \Leftrightarrow s^h_{11} = 0 \wedge s^l_{c3} = 0 \overset{\text{Prop. 1}}{\Leftrightarrow} \forall(i,j) \in P : s^l_{ij} = s^h_{ij} = 0 \Leftrightarrow S = 0$

c) "$\Rightarrow$": $\sigma \geq 2 \Rightarrow$ Let (i,j) be the position of the first split. It is $s^l_{ij} = 1$ and, because another split follows, $s^h_{ij} = 1$. $\Rightarrow S \geq 2$. It is clear from the formulation that S cannot grow larger than 2, thus we have $S = 2$.

"$\Leftarrow$": $S = 2 \Rightarrow \exists(i,j) : s^l_{ij} = s^h_{ij} = 1 \Rightarrow \exists(i',j') \prec (i,j)$ with: a split occurs at (i',j'). Execution restarts behind, and since $s^h_{i'j'} = 1$ (following from Prop. 1 b), another split occurs $\Rightarrow \sigma \geq 2$

b) follows from a) and c)

□

B. REFERENCES

[1] D. Kästner, *Retargetable Code Optimisation by Integer Linear Programming.* PhD thesis, Saarland University, 2000.

[2] E. Johnson, G. Nemhauser, and M. Savelsbergh, "Progress in Integer Programming: An Exposition," Tech. Rep. LEC-97-02, Georgia Institute of Technology, School of Industrial and Systems Engineering, Atlanta, CA 30332-0205, Jan. 1997. http://tli.isye.gatech.edu/reports.html.

[3] M. Garey and D. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freemann and Company, 1979.

[4] S. Chaudhuri, R. Walker, and J. Mitchell, "Analyzing and Exploiting the Structure of the Constraints in the ILP-Approach to the Scheduling Problem," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, pp. 456–471, Dec. 1994.

[5] F. Eisenbrand, *Gomory-Chvatal Cutting Planes and the Elementary Closure of Polyhedra.* PhD thesis, Saarland University, 2000.

[6] C. Gebotys and M. Elmasry, "Global Optimization Approach for Architectural Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1266–1278, 1993.

[7] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization, Algorithms and Complexity.* Englewood Cliffs: Prentice-Hall, 1982.

[8] G. Nemhauser and L. Wolsey, "Integer Programming," in *Handbooks in Operations Research and Management Science* (G. Nemhauser, A. R. Kan, and M. Todd, eds.), ch. VI, pp. 447–527, Amsterdam; New York; Oxford: North-Holland, 1989.

[9] D. Kästner and R. Wilhelm, "Operations Research Methods in Compiler Backends," *Journal of Mathematical Communications*, 1999.

[10] Intel, http://www.intel.com/ia64, *IA-64 Architecture Software Developer's Manual, Volume 1: IA-64 Application Architecture, Revision 1.1*, July 2000.

[11] Intel, http://www.intel.com/ia64, *Itanium Processor Microarchitecture Reference for Software Optimization*, Aug. 2000.

[12] H. Sharangpani, "Itanium Processor Microarchitecture Overview," *Microprocessor Forum (Slides)*, Oct. 1999.

[13] C. Hwang, J. Lee, and Y. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, 1991.

[14] A. Bachmann, M. Schöbinger, and L. Thiele, "Synthesis of Domain Specific Multiprocessor Systems including Memory Design," in *VLSI Signal Processing VI*, (New York), pp. 417–425, IEEE Press, 1993.

[15] C. Gebotys and M. Elmasry, *Optimal VLSI Architectural Synthesis.* Kluwer Academic, 1992.

[16] L. Zhang, *SILP. Scheduling and Allocating with Integer Linear Programming.* PhD thesis, Saarland University, 1996.

[17] S. Arya, "An Optimal Instruction Scheduling Model for a Class of Vector Processors," *IEEE Transactions on Computers*, vol. C-34, Nov. 1985.

[18] R. Leupers, *Retargetable Code Generation for Digital Signal Processors.* Kluwer Academic Publishers, 1997.

[19] P. Marwedel and G. Goossens, *Code Generation for Embedded Processors.* Boston; London; Dortrecht: Kluwer, 1995.

[20] M. Heffernan, J. Liu, and K. Wilken, "Optimal Instruction Scheduling Using Integer Programming," *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 121–133, June 2000.

[21] J. Bharadwaj and C. McKinsey, "Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs," *Journal of Instruction-Level Parallelism*, vol. 1, no. 6, pp. 1–6, 2000.

[22] C. Ferdinand, *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.

[23] J. Park and M. Schlansker, "On Predicated Execution," Tech. Rep. HPL-91-58, Hewlett-Packard Laboratories, Palo Alto CA, May 1991.

[24] J. Dehnert and R. Towle, "Compiling for the Cydra 5," *The Journal of Supercomputing*, vol. 1/2, pp. 181–228, May 1993.

[25] J. Harrison, T. Kubaska, S. Story, and P. Tang, "The Computation of Transcendental Functions on the IA-64 Architecture," *Intel Technology Journal*, vol. Q4, 1999.