

# Optimal Global Scheduling for Itanium™ Processor Family

Sebastian Winkel  
Saarland University  
Saarbrücken, Germany  
sewi@cs.uni-sb.de

## Abstract

Global scheduling with integrated decisions about speculation and predication for Itanium™ Processor Family (IPF) is widely known as a complex and challenging task.

Compilers find it especially difficult to use the proper amount of speculation and code motion, as both techniques increase the demand for execution resources. If applied too conservatively, free execution slots are wasted, contrary to the EPIC philosophy. If applied too aggressively, resource shortage can spoil the benefit. It is unknown how well state-of-the-art scheduling heuristics perform here.

We take a very precise approach to this problem: we reformulate it as a combinatorial optimization problem and apply integer linear programming (ILP) to obtain provably optimal and correct solutions. We integrate code motion with automated generation of compensation code and control speculation into the ILP model.

Since the performance of IPF is highly compiler-dependent, optimal schedules promise a speedup for compute-intensive applications, as well as some theoretically funded insights into the potential of the architecture.

Early experiments with several functions from the SPEC benchmarks show substantial improvements: Our post-pass optimizer reduces the schedule lengths produced by Intel's compiler by 20-30%.

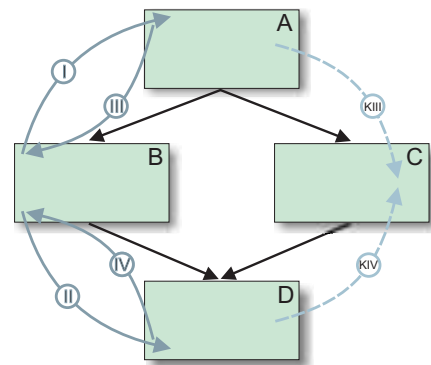
## 1 Introduction

One of the major challenges of EPIC code generation is to find a proper balance between (speculative) code motion and resource demand. Code motion is applied to decrease the schedule length, but it can also increase the resource demand in several manners:

First, as Fig. 1 shows, a speculative upward movement of an instruction like from block B to A (I) has the effect that this instruction occupies an execution slot unnecessarily on the path A-C-D.

Second, an upward movement across a join like from D to B (IV) enforces the placement of a *compensation copy*

of the instruction in block C (KIV). The resource demand does not increase for a single path, but for the total schedule. Moreover, control speculative loads require additional check instructions.



**Figure 1: Code motion: upward (I-IV), downward (III+II), speculative (I+II), non-speculative (III+IV).**

One might think that, with the plenty of execution units available in EPIC processors, resource shortage in general is not an issue. However, there is a two-fold effect how resource pressure increases as these transformations are applied:

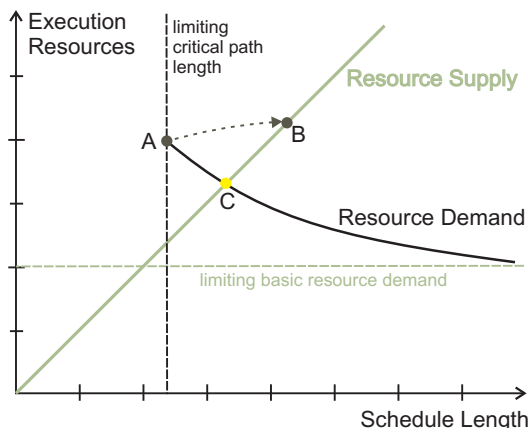
Not only does the demand for execution slots increase as described, but also the supply of execution slots decreases with the schedule length. Fig. 2 gives an idealized illustration of this effect, with the optimal schedule located at the intersection point of the supply and demand curves.

While a “hesitant” scheduler could apply code motion and speculation too conservatively and waste opportunities, an overeager code generator could decide too aggressively and thereby exceed the available resources (point A). This could spoil the benefit of these techniques as the final schedule is formed (to point B).

Besides from these difficult trade-offs, even simple local instruction scheduling is an NP-complete problem where heuristics only deliver approximations.

We use integer linear programming to obtain *globally optimal* and *provably correct* solutions to this problem.

“Optimal” means schedules with minimal length here, as defined precisely later in Sec. 4.2. It is not guaranteed that



**Figure 2: Trade-off between schedule length reduction and resource demand.**

these minimal schedules necessarily deliver maximal performance in practice – there are many other factors influencing performance – however, on a statically scheduled architecture, there should be a strong correlation between schedule length and performance.

Optimal solutions need their time – up to a few minutes –, and so we do not intend to replace the scheduling heuristics of general purpose compilers. We have two different objectives:

- We see an application as an optimization tool for performance-critical software components like encryption routines, probably as a postpass solution.
- We expect answers to some very basic questions that are still partly open for EPIC architectures:  
How much parallelism can be statically extracted by using code motion, predication and speculation?  
How well perform scheduling heuristics like wave-front scheduling [BM00]?

The rest of the paper is organized as follows: Section 2 gives a short introduction to integer programming. The basics of the ILP model and extensions are described in Sections 4 and 5, respectively. It is assumed here that the reader is familiar with fundamentals of IPF like static scheduling and control speculation. Section 6 presents the experimental results and Section 7 concludes the paper.

## 2 Integer Linear Programming

Since the invention of the simplex algorithm by George B. Dantzig over fifty years ago [Dan51], *linear programming* has developed to an indispensable tool for the formulation and solution of optimization problems.

This applies especially to the unequally more powerful – and unequally more difficult to solve – *integer* linear programming (ILP), whose potential was almost immediately

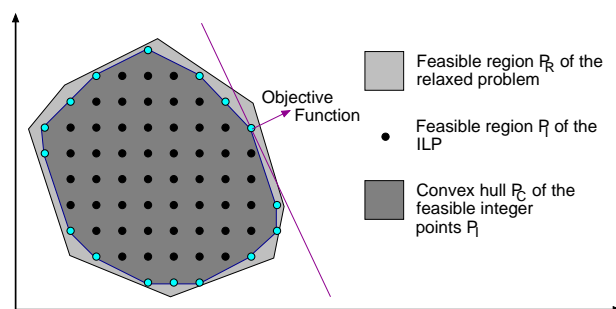
recognized after its discovery in the fifties [BFG<sup>+</sup>00]. But insufficient hardware and software have soon led to some disillusionment and to the perception that ILP has very limited practical applicability.

In the last years, however, this situation has changed “dramatically” due to advances in solution algorithms and ILP formulations [BFG<sup>+</sup>00]. This is also confirmed by our own experiences.

Integer linear programming (ILP) minimizes a linear objective function subject to a system of linear constraints given by  $P_R = \{x \mid Ax \leq b, x \in \mathbb{R}_+^n\}$  with  $c \in \mathbb{R}^n$ ,  $b \in \mathbb{R}^m$  and  $A \in \mathbb{R}^{m \times n}$ :

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

The integer points  $P_I = P_R \cap \mathbb{Z}^n$  form the *feasible solutions* or the *search space*. Computing an optimal solution is NP-complete, but the *relaxed problem* without the integrality restriction of equation (1) can be solved in *polynomial time* [NW88].



**Figure 3: Feasible regions.**

Note that if the polyhedron  $P_R$  would be made equal to the convex hull of the feasible integer points, then also the integer problem could be solved in polynomial time (see Fig. 3). Though equality usually cannot be achieved in practice, it is important for the solution efficiency to find a *tight* ILP formulation where  $P_R$  is close to this convex hull.

## 3 Related Work

Besides from many heuristics like [BM00, BR91], there exist only few ILP-based exact approaches to instruction scheduling:

Wilson et al. [WGB94] and Chang et al. [CCK97] simultaneously perform scheduling and register allocation; the former also include code selection. Both works show experimental results only for very small examples, with solution times of several seconds.

Wilken et al. [HLW00] show that by using a tight ILP formulation and clever precomputations, solution times of

less than 0.1 seconds can be achieved for scheduling basic blocks of a hundred and more instructions. Kästner and Winkel [KW01] combine scheduling and bundling for the Itanium architecture in a two-phase approach.

All works mentioned only deal with local scheduling – the only ILP model known to us that tackles acyclic global scheduling is used by the postpass optimizer PROPAN for DSPs [Käs00]. However, it allows no disjoint control flow paths in the ILP and code motion only between control equivalent basic blocks.

A local scheduler based on optimal approaches without ILP is presented by Haga and Barua [HB01]. In contrast to most earlier work, they integrate template selection into the scheduling process to minimize the number of NOPs on EPIC architectures.

## 4 The Basic ILP Model

Our goal is to formulate an ILP model for the given input program where every integer point inside the polyhedron corresponds to a possible schedule and vice versa. We say that a schedule is *feasible* if the corresponding point is a feasible solution of the ILP model.

We first collect some basic requirements for the ILP formulation. It should be kept in mind that, for search-based methods like ILP, solution efficiency is an important issue, and the last three points in the list take this into account.

**Remark 1** The ILP model should be:

1. **correct:** no incorrect schedule is feasible
2. **complete:** at least one optimal schedule must be feasible
3. **compact:** as many non-optimal schedules as possible are excluded from the search space
4. **simple:** as much abstraction and unification should be used to have as few variables and constraints as possible
5. **efficient:** the inequalities should describe a tight polyhedron (see. Sec. 2) □

During the following introduction of the basic structure of the model, the emphasis lies on the correctness.

### 4.1 Correctness

At first we assume that the scheduling region is acyclic; we will expand on loops later in section 5.2. Let  $G_B = (\mathcal{B}, E_C, \mathcal{B}_{entry}, \mathcal{B}_{exits})$  be the basic block graph of the scheduling region with the set of basic blocks  $\mathcal{B}$  and the control flow edges  $E_C$ . Entry and exit blocks are given

by  $\mathcal{B}_{entry}$  and  $\mathcal{B}_{exits}$ , respectively. We call block  $D$  a (direct) successor of  $C$  if there is a path from  $C$  to  $D$  in  $G_B$  (consisting of one edge); the definition of predecessor is analogical.

Global data dependences are given by the acyclic data dependence graph  $G_D = (V, E_D)$ . Each edge  $e \in E_D$  has a latency  $lat_e$  associated with it, where false and memory dependences<sup>1</sup> have the latency zero. On the Itanium architecture, execution units generally have a throughput of one instruction per cycle.

We can view global scheduling as a *transformation* between global schedules which rearranges instructions, but does not change the control flow structure (although it may empty some blocks). Hence the set of *program paths* – paths which go from an entry block to an exit block through the scheduling region – remains unchanged.

This allows us to take a path-based view of correctness and say that a transformation from schedule  $\delta$  to  $\delta'$  is correct if the same computations (and probably exceptions) are performed in both schedules along every program path.

To be more precise, this is the case when all instructions that occur along a path in  $\delta$  also occur there in  $\delta'$ , and when all dependences between these instructions are preserved. Additionally, *non-speculative* instructions may only appear on a path in  $\delta'$  if they appear there in  $\delta$ , too.

For each instruction  $n \in V$ , we call the block where it originates from before scheduling *source block*, denoted by  $s(n)$ . Code motion moves the instruction from this source block to a *destination block*. Possible destination blocks are all predecessors and successors of the source block in  $G_B$ . We denote  $\Theta_{spec}(n)$  as the set of those *speculative destination block candidates*<sup>2</sup> for instruction  $n$ . The range of destination blocks is further limited for non-speculative (and unpredicated) instructions like the following:

- unsafe loads [BRS92],
- stores,
- concurrent definitions, where a value can be defined by more than one instruction, depending on control flow. This is discussed in Section 5.1.
- branches, which are considered as special instructions and not included in the set  $V$ .

For those instructions a speculative placement can be ruled out if the source block dominates and postdominates the destination block for downward and upward code motion,

<sup>1</sup>i. e. dependences resulting from accesses to memory locations, for example between a store and a load.

<sup>2</sup>In the following, we often call destination block candidates simply “destination blocks”.

respectively. Accordingly, we define a set  $\Theta(n)$  of (actual) *destination block candidates* which is the same as  $\Theta_{spec}(n)$  except that the following blocks are excluded for non-speculative instructions:

- all predecessors of  $s(n)$  which are not postdominated by  $s(n)$  and
- all successors of  $s(n)$  which are not dominated by  $s(n)$

Each instruction can be scheduled into parallelly executable *instruction groups* in its destination blocks. Within each destination block  $A$ , there is a range  $G(A) = \{1, \dots, \mathbb{G}_A\}$  of possible successive groups (or *time steps / cycles*) given. Our ILP model uses the following main decision variables to model this:

$$x_n^{At} = 1 \iff \text{A copy of instruction } n \text{ is scheduled at time step } t \text{ in } A$$

These binary variables are generated for all instructions  $n$ , all destination blocks  $A \in \Theta(n)$  and all time steps therein.

In a correct schedule, every path through the source block of an instruction must contain a copy of the instruction. To express this later in an equation, we employ binary variables for all  $n \in V$  and all  $A \in \Theta_{spec}(n)$  with the following semantics:

$$a_n^{\uparrow A} = 1 \iff \text{A copy of instruction } n \text{ is scheduled on all program paths through } s(n) \text{ before } A$$

We need to couple the  $x$  and the  $a$  variables with constraints to model the described semantics. This is done inductively using the following observation:

Let  $B \in \Theta_{spec}(n)$  be a destination block and  $A \in \Theta(n)$  be a direct predecessor of  $B$ . If  $n$  is scheduled on all paths through  $s(n)$  before  $B$ , then it is *either* scheduled at  $A$  or on all paths through  $s(n)$  before  $A$ . This is expressed by the following equations which are added to the model for all instructions  $n$ , all blocks  $B \in \Theta_{spec}(n)$  and all of  $B$ 's direct predecessors  $A$  in  $\Theta(n)$ :

$$a_n^{\uparrow B} = a_n^{\uparrow A} + \sum_{t \in G(A)} x_n^{At} \quad (2)$$

In the case that a predecessor  $A$  is only a speculative destination block and not element of  $\Theta(n)$ , we generate the equation without the sum. If  $B$  has no predecessors at all, we set  $a_n^{\uparrow B} = 0$ .

It should be clear that equation (2) realizes the desired semantics. These two classes of variables are now sufficient to model global scheduling:

First, we have to ensure that every program path through the source block of an instruction contains a copy of it. This is done by the following *assignment constraints*, where  $\Omega$  is a new, empty pseudo block which is added as a successor of all exit blocks:

$$a_n^{\uparrow \Omega} = 1 \quad \forall n \in V \quad (3)$$

Further, if an instruction  $n$  is dependent on  $m$ , it must appear after  $m$  on every path. *Globally*, this can be achieved by adding the following *precedence constraints* for all  $(m, n) \in E_D$  and for all  $A \in \Theta_{spec}(m) \cap \Theta_{spec}(n)$ :

$$a_n^{\uparrow A} \leq a_m^{\uparrow A} \quad (4)$$

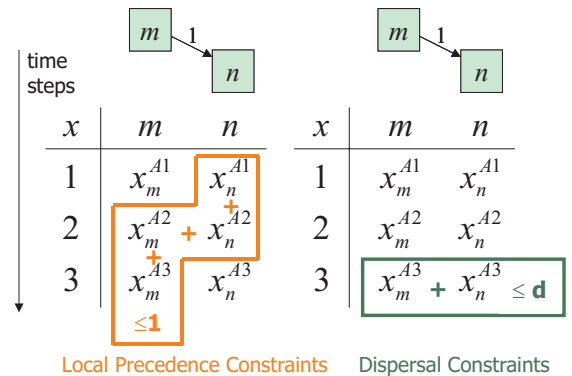
The correctness proof below will demonstrate the functioning of these inequalities. To ensure that dependences *inside* a basic block are met, we adapt the proven and efficient (tight) *local precedence constraints* from [GE93, KW01, Win01]:

$$\sum_{\substack{t_n \leq t \\ t_n \in G(A)}} x_n^{At_n} + \sum_{\substack{t_m \geq t - lat_e + 1 \\ t_m \in G(A)}} x_m^{At_m} \leq 1 \quad (5)$$

$$\forall e = (m, n) \in E_D,$$

$$\forall t \in \{t' + lat_e - 1 \mid t' \in G(A)\} \cap G(A)$$

Fig. 4 (left) helps to understand the intuition behind these constraints with a simple example consisting of two dependent instructions  $m$  and  $n$  and three time steps. The bordered area represents the variables on the left-hand side of inequality (5) (for one instance with  $t = 2$ ) – if an  $x_m$  and an  $x_n$  variable in this sum were one, this would imply that  $m$  is scheduled at time step two or three and  $n$  at one or two, which would violate the dependence. This violation is excluded by setting the sum less than or equal to one. In [Käs00] it has been shown that any infeasible instruction ordering is excluded but no feasible solution discarded.



**Figure 4: Simple example for the local precedence and dispersal constraints.**

Further constraints must ensure that the number of instructions scheduled at one time step does not exceed the target processor's execution resources. On Itanium processors, this number is generally limited by the dispersal window size  $d$  (six for Itanium 2). With help of the inverse  $\Theta^{-1}$  of  $\Theta$ , we can formulate that not more than  $d$  instructions may be issued in one cycle:

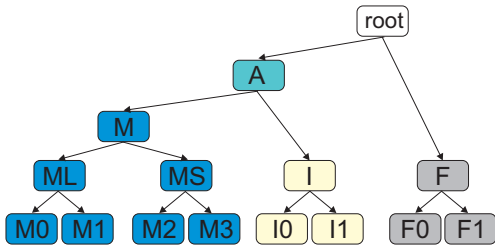
$$\sum_{n \in \Theta^{-1}(A)} x_n^{At} \leq d \quad (6)$$

$$\forall A \in \mathcal{B}, \forall t \in G(A)$$

Additionally, the number of instructions for a specific execution unit type is limited. The Itanium 2 has ten functional units: Two load units (M0+M1), two store units (M2+M3), two integer units (I0+I1), two floating-point units (F0+F1) and three branch units. Different instruction types can be executed on different subsets of execution units, for example A(LU)-type instructions on all six memory and integer units.

For complexity reasons, we do not integrate resource binding into the ILP, i. e. the ILP solver does not decide whether, for instance, an A-type instruction is executed on a memory or integer unit. In contrast to our earlier work [KW01], this decision is now left open to a later bundling phase. We can afford to do so since the Itanium 2 has significantly less restrictions than the first generation (e. g. full ALU bypassing, much more flexible bundling [NH02, Int02]).

Instead, constraints ensure that a later binding of instructions to execution units is *possible*. For this we define a set  $\{\mathcal{M}0, \mathcal{M}1, \mathcal{M}L, \mathcal{M}2, \mathcal{M}3, \mathcal{M}S, \mathcal{M}, \mathcal{I}0, \mathcal{I}1, \mathcal{I}, \mathcal{A}, \mathcal{F}0, \mathcal{F}1, \mathcal{F}\}$  of *abstract execution units*, which themselves are sets of execution units; e. g.  $\mathcal{M}L = \{\mathcal{M}0, \mathcal{M}1\}$  and  $\mathcal{I}0 = \{\mathcal{I}0\}$ . These sets are hierarchically nested according to Fig. 5, i. e. each set forms the union of its successors. We denote  $p(\mathcal{U})$  as the predecessor of  $\mathcal{U}$  in the hierarchy.



**Figure 5: Hierarchy of abstract execution units.**

For each abstract execution unit  $\mathcal{U}$ , let the set  $\sqcup(\mathcal{U}) \subseteq V$  contain those instructions which can be executed on *at least one* execution unit in  $\mathcal{U}$ , and  $\sqcap(\mathcal{U})$  those instructions which can be executed on *all* execution units in  $\mathcal{U}$ . Then the following inequalities form, as shown in [Win01], valid *resource constraints* when created for all abstract units  $\mathcal{U}$ , all blocks  $A$  and all time steps  $t$  therein:

$$\sum_{\substack{n \in \sqcup(\mathcal{U}) \setminus \sqcap(p(\mathcal{U})) \\ \cap \Theta^{-1}(A)}} x_n^{At} \leq |\mathcal{U}| \quad (7)$$

A small example shows that the idea behind these constraints is straightforward. Consider three loads  $l_1, l_2, l_3$ , four A-type instructions  $a_1, a_2, a_3, a_4$  and two instructions  $i_1$  and  $i_2$  which can only be executed on I0. Then the following constraints are generated for time step 1:

$$\begin{aligned} x_{l_1}^{A1} + x_{l_2}^{A1} + x_{l_3}^{A1} + x_{a_1}^{A1} + \\ x_{a_2}^{A1} + x_{a_3}^{A1} + x_{a_4}^{A1} + x_{i_1}^{A1} + x_{i_2}^{A1} &\leq 6 \quad (\mathcal{A}) \\ x_{l_1}^{A1} + x_{l_2}^{A1} + x_{l_3}^{A1} &\leq 2 \quad (\mathcal{ML}) \\ x_{i_1}^{A1} + x_{i_2}^{A1} &\leq 1 \quad (\mathcal{I}0) \end{aligned}$$

The polytope for global scheduling is now complete. Regarding the demand for simplicity from Rem. (1), it should be noted that we need only two classes of variables and six sorts of constraints for the basic model. Under the assumption that  $|V| \leq |E_D|$  and with  $\mathbb{G} = \sum_{A \in \mathcal{B}} \mathbb{G}_A$ , we need  $\mathcal{O}(\mathbb{G} \cdot |V|)$  variables and  $\mathcal{O}(\mathbb{G} \cdot |E_D|)$  constraints.

We conclude with a correctness proof:

**Corollary 1** *For any feasible schedule holds: Instruction  $n$  is scheduled at block  $A$  if and only if the variables  $a_n^\uparrow$  have value one for all successors of  $A$  in  $\Theta_{spec}(n)$  and value zero for  $A$  and its predecessors in  $\Theta_{spec}(n)$ .*  $\square$

PROOF Follows inductively from equation (2).  $\blacksquare$

**Theorem 1 (Correctness)** *Every integer point that satisfies constraints (2)-(7) corresponds to a correct global schedule.*  $\square$

PROOF Let the corresponding schedule and a program path  $P$  be given. It is sufficient to show for any instruction  $n$  with source block on  $P$  that it (1) appears along this path in the schedule, and that (2) dependences on other instructions are not violated.

The first claim follows directly from constraint (3) since  $P$  passes through  $\Omega$ . So let  $n$  be scheduled in block  $A$  on  $P$ .

For the second claim, we show that for each dependence  $(m, n) \in E_D$  (with  $s(m) \in P$ ), a copy of  $m$  is scheduled *before or in*  $A$  on  $P$  and no copy *after*  $A$  on  $P$ .

We first note that all blocks on  $P$  are in  $\Theta_{spec}(m)$  and  $\Theta_{spec}(n)$ . From the corollary we know that the variables  $a_n^\uparrow$  have value one for all successors of  $A$  on  $P$  and value zero for  $A$  and its predecessors on  $P$ . With (4) it follows that the variables  $a_m^\uparrow$  are one for all successors of  $A$  on  $P$ , too.

Consequently, no copy of  $m$  is scheduled in a successor of  $A$  on  $P$  (again with the corollary) and, since  $a_m^\uparrow$  is one for the *direct* successor of  $A$  on  $P$ ,  $m$  is scheduled *before or in*  $A$  on  $P$ .

Concerning dependences inside a basic block and the resource limitations (constraints (5) and (7), respectively), references to the correctness proofs have been given in the text above. ■

The correctness of the generated code follows directly from this correctness proof. In general, any schedule is correct if it is a feasible solution of the ILP (which can be checked in time that is linear in the size of the ILP).

This property can also be used to validate schedules produced by heuristics. It is an inherent advantage of this approach which builds not on an algorithm, but on a precise mathematical model.

## 4.2 Completeness

At least one optimal and correct schedule must be feasible in order to be found by the ILP solver. We can show that with the given ILP model – independent of how optimality is defined – any correct schedule is feasible:

**Theorem 2 (Completeness)** *Let a correct schedule be given where no instruction is placed twice on any path. Then the corresponding integer point is a feasible solution of the ILP model.* □

We must leave out the proof here for lack of space. It is important for the proof that we exclude the possibility to schedule an instruction twice on a path in the theorem. This restriction can be lifted for *P-ready code motion* later (see Section 5.3).

There are several other aspects that could restrict completeness, but are out of the scope of the above theorem.

One concerns the number of time steps  $\mathbb{G}_A$  given for each basic block  $A$ . The ILP solver could choose to grow less frequently executed blocks by moving code into them – this possibility should not be limited by a too small  $\mathbb{G}_A$ . This is a sensitive issue since this value heavily affects the size and thereby the solution times of the produced ILPs. A save choice is to collect all instructions which could possibly be moved into the block,  $\Theta^{-1}(A)$ , and compute via list scheduling an upper bound on the length of an optimal local schedule of all these instructions.

Another point concerns the destination blocks of non-speculative instructions: the latter can be executed speculatively if they are guarded by predicates which eliminate the speculativeness; this allows an extension of the range of destination blocks.

For *upward motion* of an instruction, such a predicate register can be found as follows: For all control flow edges  $(A, B) \in E_C$  where  $B$  dominates the source block of the instruction and  $A$  not, the qualifying predicate of the branch associated with the edge is a candidate. Guarded by this predicate register, the instruction can be safely

moved to  $A$  and all of its predecessors (but there is a new data dependence on the compare which generates the predicate value).

We perform a similar extension for *downward motion* of an instruction: let the set  $\mathcal{E}$  contain the source block and all control equivalent basic blocks, and let  $A \in \mathcal{E}$  be the “top” block of them, i. e. the one that has no predecessors in  $A$ . We examine the control flow edges leading to  $A$  – if it is only one edge with a predicate associated with it, then the instruction is executed if and only if this predicate is true. When used as a qualifying predicate of the instruction, the latter can be moved downwards arbitrarily far.

This way we determine for each new destination block a predicate register which must be used as qualifying predicate if the instruction is scheduled there; so we include predication in our model as a *side-effect of code motion*.

We finally define the notion of optimality exactly: Optimization goal is to minimize the *global schedule length*, which we define as the sum of the schedule lengths of all basic blocks, each weighted by the execution frequency of the block (which we assume to be given).

To integrate this into the model, we introduce a new integer (but not binary) variable  $T_A$  which is greater than or equal to the length of basic block  $A$  in the schedule:

$$\sum_{t \in G(A)} t \cdot x_n^{At} \leq T_A \quad \forall n \in V, \forall A \in \Theta(n)$$

With the execution frequency of block  $A$  given as  $f_A$ , the objective function can be written as:

$$\min \sum_{A \in B} f_A \cdot T_A \quad (8)$$

We do not integrate register allocation into the model since IPF offers a large amount of 128 architecturally visible registers. Hence the interaction between register allocation and scheduling is relatively low so that it can be done in separate phases efficiently.

## 5 Extensions

### 5.1 Speculation

Non-speculative instructions are limited in their scope of code motion as they may not be executed unnecessarily. In general, there are two reasons why the unnecessary execution of an instruction could harm correctness: First, it could trigger a false exception, which concerns mostly memory instructions. Second, it could overwrite a live value; this applies to stores and to *concurrent definitions* [Käs00]:

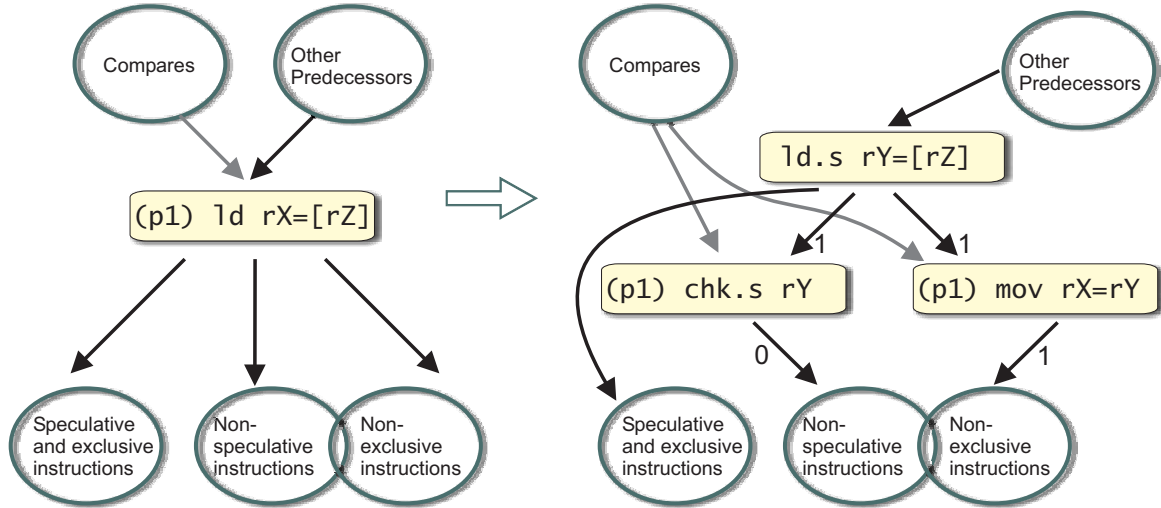


Figure 6: Control speculation with register renaming before (left) and after application (right).

**Definition 1 (Concurrent Definitions)** Two instructions are called *concurrent definitions* if they write the same register and if there is a use of that register reachable by both definitions (i. e., both are in the use-def chain of the use).

A use of the register is called *exclusive* if it is reachable only by one definition.  $\square$

Fig. 6 presents a scheme that allows to execute a load which is also a concurrent definition speculatively – at the price of two additional instructions plus recovery code. The left-hand side shows the original load where the arrows represent sets of true data dependence edges. We distinguish three groups of uses of the register `rX` written by the load: speculative<sup>3</sup> uses which are also exclusive with respect to `rX`, non-speculative instructions and non-exclusive uses, where the latter two groups may overlap, as depicted in the figure.

The right-hand side shows how the load is replaced by a control-speculative version `ld.s` which writes to a new temporary register. All speculative and exclusive uses can directly read the temporary register and thereby be speculated with the load.

The check instruction `chk.s` detects an exception deferred by the speculative load `ld.s` and must be scheduled before all non-speculative uses. The new `mov` instruction moves the loaded value from the temporary to the original register. It must analogically be executed before all non-exclusive uses.

These two new instructions are non-speculative and hence must be guarded by the predicate of the former non-speculative load.

The scheme is comprehensive in the sense that it is also possible to speculate a concurrent definition that is not

<sup>3</sup>Analogous to the term “non-speculative”, we call instructions “speculative” if they can be executed speculatively.

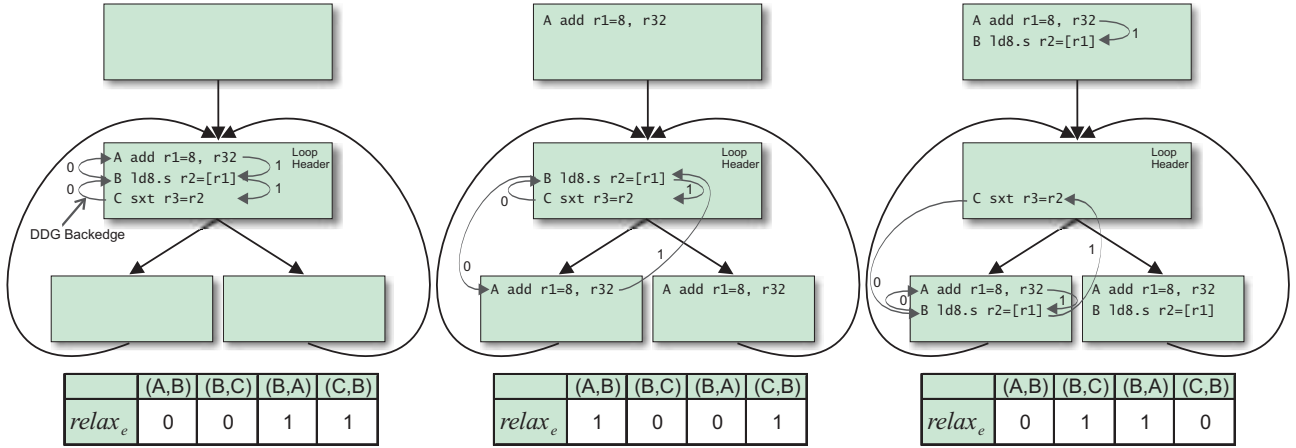
a load and vice versa – the `chk` and the `mov` are then dropped, respectively. It also allows to cascade several dependent speculative instructions – an example of this is discussed later in Sec. 6.

We integrate the possibility to use this kind of speculation into the search space so that the ILP solver can optimally decide whether to employ it. It decides between two mutually exclusive instruction groups: The first consists of the normal load and the second of the speculative version and the `chk` and/or the `mov`. Either the first or the second group must appear in the final schedule, and their dependences must be obeyed.

To realize this, we include the instructions of *both groups* in the ILP and define a binary variable *usespec* as a “speculation switch”. The right-hand side of the assignment constraints (3) is replaced by  $(1 - \text{usespec})$  and *usespec* for instructions from the first and second group, respectively.

To switch the global precedence constraints on and off, we add the first and the second term to the right-hand side of (4) for all dependences involving instructions from the first and second group, respectively. The terms *relax* these inequalities if the dependences are “switched off”. All other constraints of the ILP are not affected by these mutually exclusive instruction groups.

It should be noted that the branch to recovery code is not taken if the speculation fails, but only if the load triggers an exception. These cases are very rare and expensive anyway, so the cost of recovering does not need to be taken into account. This is different for data speculation, which is implemented very similarly, but we cannot go into details here for lack of space.



**Figure 7: Example for cyclic code motion. The DDG backedges with latency zero are WAR dependences. The relax terms in the table show how these and the other dependences are switched on and off as the add and the load are cyclically moved ( $relax_e = 0$  and  $relax_e = 1$ , respectively).**

## 5.2 Cyclic Code Motion

Unlike many other global scheduling algorithms, we do not limit the scheduling scope to acyclic regions. Loops are such an essential element of every program that this restriction would limit the search space and thereby our objective of truly optimal schedules too strongly.

Consequently, we allow code motion *into* loops and *out of* loops. The former can only be allowed if the instruction

- does not write a value which is live at the loop header
- is *multiple executable* without a changing semantics, which is not the case for instructions where results and operands overlap, like `add r1=1, r1`.

In most cases, we cannot move an instruction completely out of a loop<sup>4</sup>. For instance, upward code motion of an instruction requires that copies of it are moved not only upwards into blocks above the loop header, but also *along every backedge* to the bottom blocks of the loop and their predecessors. We call this kind of code motion *cyclic*.

Fig. 7 shows a small example where the `add` (middle) and subsequently the `load` (right) are moved cyclically.

One benefit of cyclic motion is that parts of the first iteration of the loop can be overlapped with code preceding the loop and thereby be executed earlier. If the loop-carried dependences permit it, the cyclically moved code can also be overlapped inside the loop body, which may lead to a drastically reduced critical path here.

Cyclic code motion can then be regarded as a simple variant of software pipelining without fill overhead. In practice, there are still many loops where software pipelining

<sup>4</sup>This would only be possible for loop invariant instructions, but we expect that this optimization has already been done.

can or should not be applied – for example if they contain other loops or function calls, or if they have low trip counts – cyclic code motion can then help alleviate the inefficiencies static scheduling suffers from in these cases.

We currently have integrated only cyclic upward code motion for speculative instructions into the ILP. Little changes are necessary and no new variables and constraints need to be introduced. For an instruction  $m$  originating from a loop with header  $H$ , the variable  $a_m^{\uparrow H}$  is assigned a special meaning:  $m$  is cyclically moved if and only if  $a_m^{\uparrow H} = 1$ . This choice is left open to the ILP solver.

As cyclic code motion changes the order of instructions inside the loop body, the data dependences change, too. This is modeled by adding a  $relax_e$  term to the right-hand side of the inequalities (4) and (5), which is  $relax_e := 1 + a_m^{\uparrow H} - a_n^{\uparrow H}$  for DDG backedges  $(m, n)$  and  $relax_e := a_m^{\uparrow H} - a_n^{\uparrow H}$  for normal DDG edges. How the dependences are switched on and off is illustrated exemplarily in Fig. 7.

## 5.3 Further Work

We briefly sketch several extensions we are currently developing and testing:

**Partially-ready code motion** can be supported, as described in [BM00]. Global data dependences can vary with control flow, and P-ready code motion allows those to be ignored which are not relevant on a program path. This can lead to a further schedule length reduction.

**Long-latency instructions** like floating-point commands are currently not properly supported as the global effect of these latencies between basic blocks is not allowed for. An extension is being developed.

**Software pipelining** [AJLA95]: A model to minimize the length of the software pipeline and the kernel has been presented in [Win01].

**Stall minimization** is a code reordering technique which is applied in a separate phase after scheduling and expands the distances between loads and their nearest use, while preserving the optimality of the schedule. The objective function of the ILP can be modified for this purpose. The reordering minimizes the stall cycles due to cache misses, which often account for a large proportion of the whole execution time on statically scheduled architectures.

## 6 Experimental Results

We have implemented all described modelings and conducted several experiments. With the help of Intel’s VTune Analyzer, we first located some hot routines in the SPEC CINT2000 benchmark. Our selection is currently limited to four routines since floating-point instructions, function calls and software pipelining are not yet supported by the implementation.

The selected routines were compiled to assembly with Intel’s compiler 6.0.1 for Itanium. We used full optimization (-Ox) and the switch -G2 to obtain code for Itanium 2.

The assembly files are directly input to our optimizer. The latter reconstructs control flow, data dependences and also reads the execution frequency estimates for function (8) which are delivered by Intel’s compiler although no profile feedback could be used. It also undoes all usages of control speculation and performs register renaming to remove all false dependences which would otherwise restrict code motion. We also perform several automated optimizations to make the search space *compact*, e. g. we exclude possibilities for code motion which cannot be utilized in any correct and optimal schedule.

The advantage of the postpass approach is that we can compare the results directly with those produced by Intel’s state-of-the-art compiler [DKK<sup>+</sup>99]. Since we lack an Itanium 2 machine, however, this comparison currently can only occur statically.

A drawback of the postpass approach is that no information about memory disambiguation is available. Hence memory dependencies must be reconstructed conservatively. However, in *longest\_match* and *add\_penal* one and six instances of data speculation, respectively, are applied in the optimal schedule to overcome such dependences. From the source code it is evident that no address aliasing is possible in these cases, so the cost of failed data speculation does not need to be taken into account. Nevertheless, more comprehensive memory disambiguation information is indispensable for further experiments [GLS01].

Table 1 shows for each routine the number of basic blocks, loops and the number of instructions and speculation usages at different stages: as produced by Intel’s compiler (“Int.”), as possibilities included in the ILP (“ILP”), and finally as occurring in the optimal schedule (“Opt.”).

The number of instructions included in the ILP, #Ins.ILP, is generally higher than the original number, #Ins.Int, because additional instructions like speculative loads with their checks are generated. Since not all of those speculation possibilities are used in the optimal schedules (#Spec.Opt. < #Spec.ILP), #Ins.Opt is usually lower than #Ins.ILP.

An exception is *get\_bb\_from\_scratch*, where multiple compensation copies of instructions (see Sec. 1) lead to a higher count in the final schedule.

Table 2 shows the size of the ILPs after *presolve*, a preprocessing by the ILP solver which removes redundant constraints and variables. The solution times range between 3 and 8 seconds with the ILP solver CPLEX 8.0 [cpl02] on a 900-MHz-UltraSparc III+.

Fig. 8 shows the results – the average improvement is about 30%. Further improvements – although diminishing – occur when we define a target processor with more execution units. This indicates that the schedules for Itanium 2 are still resource-bound.



**Figure 8: Global schedule length improvements over Intel’s compiler on the same Itanium 2 target processor (left bars) and on a processor with the double number of execution units (right bars).**

Fig. 9 shows exemplarily for the entry block of *get\_bb\_from\_scratch* how such improvements are possible. The reduction of the critical path is achieved by speculating two concurrent definitions

- (p7) add r10=16, r17
- (p8) shladd r10=r23, 3, r16

and three loads

- (p7) ld8 r14=[r10]
- (p8) ld8 r14=[r10] (both concurrent)
- (p8) ld8 r16=[r10].

Routine Name (Benchm.)	#Ins.Int.	#Ins.ILP	#Ins.Opt.	#Spec.Int.	#Spec.ILP	#Spec.Opt.	#BB/Loops
get_bb_from_scratch (vpr)	122	137	142	0	8	7	3/1
add_penal (twolf)	109	163	123	0	26	11	9/1
find_new_pos (twolf)	107	114	108	0	8	1	11/0
longest_match (gzip)	154	202	171	12	37	18	20/2

**Table 1: Input routines.**

Routine Name (Benchm.)	#Constraints	#Variables	Solution Time (in seconds)
get_bb_from_scratch (vpr)	1981	1465	3
add_penal (twolf)	2666	1603	6
find_new_pos (twolf)	1457	873	8
longest_match (gzip)	2878	1794	8

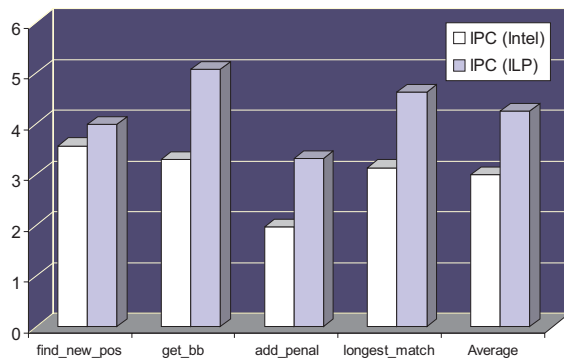
**Table 2: Numbers of constraints, variables and the solution times of the ILPs.**

Cycle	Input Schedule	Output Schedule
1	add r11=@gprel(net#),gp add r9=@gprel(duplicate_pins#),gp add r10=@gprel(unique_pin_list#),gp sxt4 r23=r32	add r11=@gprel(net#),gp add r9=@gprel(duplicate_pins#),gp add r10=@gprel(unique_pin_list#),gp sxt4 r23=r32
2	ld8 r8a=[r9] ld8 r3a=[r11] shl r22=r23,5	ld8 r8a=[r9] ld8 r3a=[r11] shl r22=r23,5
3	shladd r19=r23,2,r8a add r17=r3a,r22	shladd r19=r23,2,r8a add r17=r3a,r22 <b>ld8.s r16=[r10]</b>
4	ld4 r18=[r19]	ld4 r18=[r19] <b>add r10b=16,r17</b> <b>shladd r10a=r23,3,r16</b>
5	cmp4.ne.unc p8,p7=r18,r0	cmp4.ne.unc p8,p7=r18,r0 <b>ld8.s r14b=[r10b]</b> <b>ld8.s r14a=[r10a]</b>
6	(p7) add r10=16,r17 (p8) ld8 r16=[r10]	(p8) <b>chk.s r16</b> (p7) <b>mov r14=r14b</b> (p8) <b>mov r14=r14a</b> (p7) <b>mov r10=r10b</b> (p8) <b>mov r10=r10a</b>
7	(p7) ld8 r14=[r10] (p8) shladd r10=r23,3,r16	(p7) <b>chk.s r14b</b> (p8) <b>chk.s r14a</b> ld4 r26=[r14] add r32=4,r14
8	(p8) ld8 r14=[r10]	
9	ld4 r26=[r14] add r32=4,r14	

**Figure 9: Example for schedule length reduction through control speculation. Shown is a slice from *get\_bb\_from\_scratch* before and after optimization. New instructions are written in bold; new, renamed registers have an appended 'a' or 'b'.**

Cyclic code motion accounts for another large part of the improvements: when switched off, the numbers from Fig. 8 drop to 17% on the average. In *longest\_match*, for example, cyclic code motion is applied to six instructions from the outer loop and to another six from the inner loop, which significantly reduces the critical path lengths in both loops.

Finally, Fig. 10 shows how the instruction-per-clock rate moves closer to the maximum of six as the schedule length decreases and more slots are used for speculation.



**Figure 10: The optimal schedules (right bars) come closer to the theoretical maximum of six instructions per cycle.**

## 7 Conclusion and Outlook

To our knowledge, we are the first to present a comprehensive formal description of global scheduling with integrated generation of compensation code, cyclic code motion and speculation.

Much effort has been spent to make the ILP model not only correct and mostly complete, but also simple and efficient. As a result, we are able to compute optimal schedules for regions with 100 to 200 instructions within a few seconds.

Our early experimental results are very promising: the ILP solver could, applied as a postpass optimizer, reduce the schedule lengths of several functions about 20-30% compared with Intel’s sophisticated compiler. Cyclic code motion accounts for a major part of the reduction.

With the small amount of static experimental results so far, it is too early to draw final conclusions from these numbers. However, the extent of the improvements indicates that there is still significant performance headroom in some tasks which are very fundamental to EPIC: static scheduling and usage of speculation. The optimal schedules show the way.

## 8 Acknowledgements

This research has been funded by the graduate studies program “Quality Guarantees for Computer Systems” supported by the Deutsche Forschungsgemeinschaft. My further thank goes to the Max-Planck-Institut für Informatik, Saarbrücken, for giving access to their CPLEX installation.

## References

- [AJLA95] V.H. Allan, R.B. Jones, R.M. Lee, and S.J. Allan. Software Pipelining. *Computing Surveys*, 27(3):367–432, September 1995.
- [BFG<sup>+</sup>00] Robert E. Bixby, Mary Fenelon, Zonghao Gu, Ed Rothberg, and Roland Wunderling. ILOG CPLEX Division. MIP: Theory and Practice - Closing the Gap. In M. J. D. Powell and S. Scholtes, editors, *System Modelling and Optimization: Methods, Theory and Applications*, pages 19–49. Kluwer, The Netherlands, 2000.
- [BM00] J. Bharadwaj and C. McKinsey. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. *Journal of Instruction-Level Parallelism*, 1(6):1–6, 2000.
- [BR91] David Bernstein and Michael Rodeh. Global Instruction Scheduling for Superscalar Machines. *Proceedings of the ACM SIGPLAN ’91 on Programming Language Design and Implementation (PLDI)*, June 1991.
- [BRS92] D. Bernstein, M. Rodeh, and M. Sagiv. Proving Safety of Speculative Load Instructions at Compile-Time. *Proceedings of the 4th European Symposium on Programming*, 1992.
- [CCK97] 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 and Mathematics with Applications*, 34(9):1–14, November 1997.
- [cp102] ILOG CPLEX 8.0, 2002. [www.cplex.com](http://www.cplex.com).
- [Dan51] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In Tj. C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley, New York, 1951.
- [DKK<sup>+</sup>99] Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John

- Ng, and David Sehr. An Overview of the Intel® IA-64 Compiler. *Intel Technology Journal*, (Q4), 1999.
- [GE93] C.H. Gebotys and M.I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1266–1278, 1993.
- [GLS01] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, pages 47–58, June 2001.
- [HB01] Steve Haga and Rajeev Barua. EPIC Instruction Scheduling Based on Optimal Approaches. *Proceedings of the EPIC-1 Workshop*, December 2001.
- [HLW00] 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*, pages 121–133, June 2000.
- [Int02] Intel. *Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*, June 2002.
- [Käs00] Daniel Kästner. *Retargetable Code Optimization by Integer Linear Programming*. PhD thesis, Saarland University, 2000.
- [KW01] Daniel Kästner and Sebastian Winkel. ILP-based Instruction Scheduling for IA-64. *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, June 2001.
- [NH02] Samuel D. Naffziger and Gary Hammond. The Implementation of the Next Generation 64b Itanium™ Microprocessor. *Proceedings of the IEEE International Solid-State Circuits Conference*, 2002.
- [NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.
- [WGB94] T.C. Wilson, G.W. Grewal, and D.K. Banerji. An ILP Solution for Simultaneous Scheduling, Allocation, and Binding in Multiple Block Synthesis. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 581–586. IEEE Computer Society Press, 1994.
- [Win01] Sebastian Winkel. ILP-basierte Instruktion-sanordnung für IA-64. Master's thesis, Saarland University, 2001. In German.