Exploring the Performance Potential of Itanium Processors with ILP-based Scheduling

Sebastian Winkel

Compiler Research Group, Informatik
Saarland University, Saarbrücken, Germany
E-mail: sewi@cs.uni-sb.de

Abstract

HP and Intel's Itanium Processor Family (IPF) is considered as one of the most challenging processor architectures to generate code for. During global instruction scheduling, the compiler must balance the use of strongly interdependent techniques like code motion, speculation and predication. A too conservative application of these features can lead to empty execution slots, contrary to the EPIC philosophy. But overuse can cause resource shortage which spoils the benefit.

We tackle this problem using integer linear programming (ILP), a proven standard optimization method. Our ILP model comprises global, partial-ready code motion with automated generation of compensation code as well as vital IPF features like control / data speculation and predication. The ILP approach can – with some restrictions – resolve the interdependences between these decisions and deliver the global optimum. This promises a speedup for compute-intensive applications as well as some theoretically funded insights into the potential of the architecture.

Experiments with several hot functions from the SPEC benchmarks show substantial improvements: Our postpass optimizer reduces the schedule lengths produced by Intel's compiler by about 20-40%. The resulting speedup of these routines is 16% on average.

1. Introduction

The Itanium Processor Family is an Explicitly Parallel Instruction Computing (EPIC) architecture where the compiler is solely responsible for extracting and managing instruction-level parallelism. The compiler explicitly groups parallelly executable instructions and relieves the hardware of detecting dependences inside these groups [14].

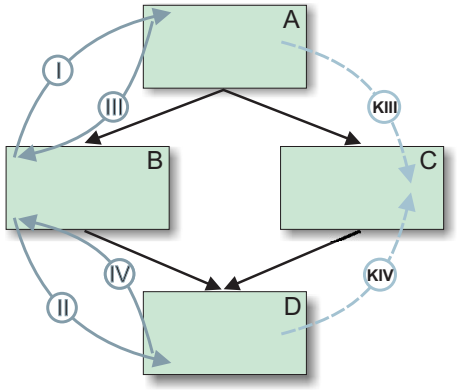

Figure 1. Code motion: upward (I+IV), downward (III+II), speculative (I+II), non-speculative (III+IV)

Instruction groups can be arbitrarily large. However, the compiler should choose groups which do not need more execution units than the target processor has to avoid split cycles. So code generators, including ours, usually form groups which can be issued in one cycle. The execution of instruction groups always starts in-order, but individual instructions can terminate out-of-order. The execution pipeline stalls if an operand of an instruction is not yet available; the processor tracks inter-group dependences to detect this.

Register anti-dependences and memory dependences are allowed inside instruction groups (*intra-group dependences*). In these cases the order of instructions inside the groups still matters. The code generator must allow for this when it packs the instructions into fixed-sized 128-bit bundles of three (together with the template restrictions). This work targets the Itanium 2 processor which can execute two of these bundles with six instructions altogether per cycle [15].

EPIC allows a simple, wide execution hardware with plenty of parallel execution units – but it also places the burden of feeding them on the compiler. During scheduling it has to apply global code motion, speculation and predication to shorten the schedule length, but these techniques can also increase the demand for execution slots in several ways, as Fig. 1 shows:

- A speculative upward movement of an instruction from block B to A (I) has the effect that this instruction occupies an execution slot unnecessarily on the path A-C-D.
- An upward movement across a join from D to B (IV) enforces the placement of a *compensation copy* of the instruction in block C (KIV), increasing the instruction count.
- Moreover, control speculative loads require additional check instructions.

There is a two-fold effect how resource pressure increases as these transformations are applied: First, the demand for execution slots grows as described. Second, at the same time the supply of execution slots diminishes because the schedule length shrinks. Apart from these difficult trade-offs, instruction scheduling is – even locally – an NP-complete problem where heuristics deliver only approximations.

We use integer linear programming to obtain *globally optimal* and *provably correct* solutions to this problem. Note that the notion of "optimality" is used in an algorithmic sense here: The resulting schedule is optimal with respect to our mathematical definition of the global scheduling problem, i. e. it has a minimal global schedule length.

In the real world, however, optimality is relative and more complex: for example, it depends on the input set which we must approximate by profiling information. Also there are a lot of difficult to predict, influential dynamic effects like cache misses interacting with the schedule. Clearly, no mathematical model can fully and precisely describe and minimize these runtime effects. We can achieve strict optimality only within a well-defined problem scope. However, on a statically scheduled architecture, there should be a strong correlation between schedule length and performance – and this is also what our experiments confirm.

Optimal solutions for NP-complete problems need their time – in our case up to a few minutes for individual routines – hence this method is not suited for production compilers. Instead it is intended for use in professional optimization and research tools (see Sec. 7).

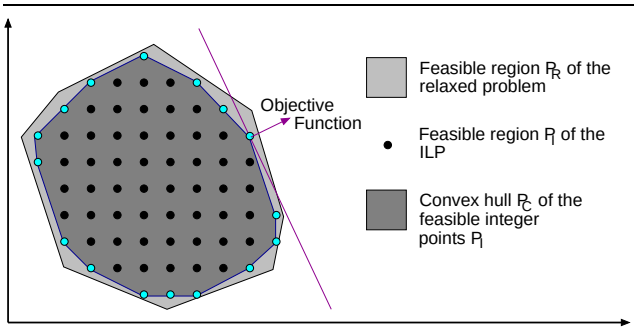

Figure 2. Feasible regions

2. Integer Linear Programming

Since the invention of the simplex algorithm by George B. Dantzig over fifty years ago [9], *linear programming* (LP) 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). Both LP and ILP minimize (resp. maximize) an objective function subject to linear constraints. The distinguishing feature of ILP is that the variables belong to a discrete set, namely a subset of integers, which enables the modeling of *combinatorial* or *discrete* optimization problems.

The potential of ILP was almost immediately recognized after its discovery in the fifties [5], 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 as well as ILP formulations [5]. This is also confirmed by our own experiences.

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

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

The integer points $P_I = P_R \cap \mathbb{Z}^n$ inside the polyhedron P_R form the *feasible solutions* or the *search space*. Searched is the point that is furthest in the direction of the objective function. Computing such an optimal solution is NP-complete, but the *relaxed problem* without the integrality restriction of equation (1) can be solved in *polynomial time* [19, 20]. Then one of the vertices of the polyhedron is always an optimal solution [19, 20].

Note that, following from the last sentence, 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. 2). 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 many heuristics like [4], there exist only few ILP-based exact approaches to instruction scheduling: Wilson et al. [21] and Chang et al. [6] 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. [13] 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. An earlier work by the author with Daniel Kästner [17] combines scheduling and bundling for the Itanium architecture in a two-phase approach.

All works mentioned so far 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 [16]. However, it allows no disjoint control flow paths in the ILP and code motion only between control equivalent basic blocks. PROPAN adopts, like this work, the OASIC formulation of the precedence constraints [11] whose high efficiency has been proven through polyhedral analysis in [7, 16].

A local scheduler based on optimal approaches without ILP is presented by Haga and Barua [12]. 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 find a set of linear inequalities for a 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.

It must be ensured that no incorrect schedule is feasible (*correctness*) and that at least one optimal schedule is feasible in order to be found by the ILP solver (*completeness*). Also it should be considered that, for search-based methods like ILP, solution efficiency is a dominating issue. Hence the search space should be kept as *compact* as possible and the linear inequalities should describe a *tight* polyhedron.

Before we present our ILP model which aims to fulfill these requirements, we examine under which conditions code motion is semantics-preserving and where compensation copies have to be scheduled. At first we assume that the scheduling region is acyclic; we will expand on loops later in section 5.2. Let $G_B = (\mathcal{B}, E_C, \mathcal{B}_{entry}, \mathcal{B}_{exit})$ 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 $\mathcal{B}_{entry} \subseteq \mathcal{B}$ and $\mathcal{B}_{exit} \subseteq \mathcal{B}$, 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; false and memory dependences have the latency zero. On the Itanium architecture, almost all execution units 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 δ to δ' 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 δ also occur there in δ', and when all dependences between these instructions are preserved. Additionally, *non-speculative* instructions may only appear on a path in δ' if they appear there in δ, 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*[1] for instruction n. This range of destination blocks is further limited for non-speculative (and unpredicated) in-

1 In the following, we often call destination block candidates simply "destination blocks".

structions which must not be executed unnecessarily like loads and stores. 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, 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 $s(n)$

Moreover, non-speculative instructions can be executed speculatively if they are guarded by a predicate which eliminates 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 is postdominated by 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; in addition, this compare then must not be speculated itself).

We perform a similar extension for *downward motion* of an instruction [22]. 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; in doing so we include predication in our model as a *side-effect of code motion*.

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

$$x_n^{At} = 1 \quad \Leftrightarrow \quad \text{A copy of instruction } n \text{ is scheduled at cycle } t \text{ in } A$$

These binary variables are generated for all instructions n, all destination blocks $A \in \Theta(n)$ and all cycles 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:

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

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 $\Theta(n)$

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

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[2]. 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 Ω is a new, empty pseudo block which is added as a successor of all exit blocks:

$$a_n^{\uparrow \Omega} = 1 \qquad \forall n \in V \tag{3}$$

4.1. Precedence Constraints

Furthermore, if an instruction n is dependent on m, then 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 $A \in \Theta_{spec}(m) \cap \Theta_{spec}(n)$:

$$a_n^{\uparrow A} \leq a_m^{\uparrow A} \tag{4}$$

To ensure that dependences *inside* a basic block are met, we adopt the proven and efficient (tight) *local precedence constraints* from [11, 17, 23]:

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

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

2 In practice we use an optimized version which omits the equation in this case. This simpler version is presented here for the sake of comprehensibility.

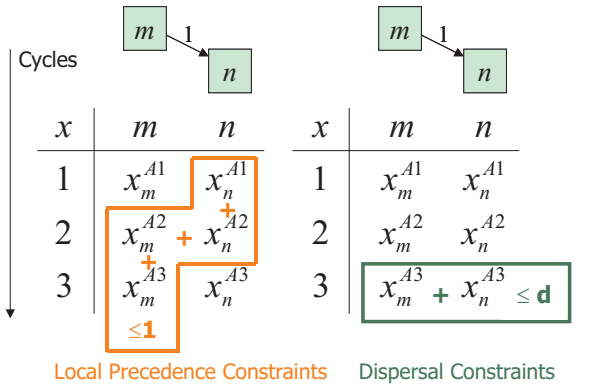

Figure 3. Simple example for the local precedence and dispersal constraints

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

Fig. 3 helps to understand the intuition behind these constraints with a simple example consisting of two dependent instructions m and n and three cycles. 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 cycle 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 [16] it has been shown that any infeasible instruction ordering is excluded but no feasible solution discarded.

4.2. Resource Constraints

Further constraints must ensure that the number of instructions scheduled per cycle 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 Θ^{-1} of Θ, we can formulate that not more than d instructions may be issued in one cycle:

$$\sum_{n \in \Theta^{-1}(A)} x_n^{At} \leq d \qquad \forall A \in \mathcal{B},\ \forall t \in G(A) \qquad (6)$$

Similarly, we generate resource constraints which limit the number of instructions scheduled for a specific execution unit type [22, 23]. For complexity reasons, we do not integrate resource binding into the ILP, i. e. the ILP solver does not decide whether, for instance, an ALU instruction is executed on a memory or an integer unit. In contrast to our earlier work [17], 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 [15]).

In the presence of intra-group dependences, however, it can happen that later an instruction group does not fit in any possible template sequence (see Sec. 1, [17, 8]). It is possible to integrate template selection into the model, but to avoid the additional complexity we have chosen a different solution. Our implementation searches in advance for potential instruction groups which would later fail during bundling. Given such an "illegal" set of instructions S, it adds the *bundling constraint* $\sum_{n \in S} x_n^{At} \leq |S| - 1$ for each cycle t where these instructions can be scheduled to prevent the formation of this group. In our experiments only two out of several hundred groups failed during bundling (they had four memory instructions, all depending on a `chk.s`). This has been resolved by adding a specific class of bundling constraints [23]. We will add further classes when more of these cases occur.

The polytope for global scheduling is now complete. 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. $\mathbb{G}_A$, the number of cycles reserved for a basic block A, should be chosen as small as possible since this value affects the size and thereby the solution times of the produced ILPs. However, 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 $\mathbb{G}_A$. 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.

We conclude the presentation of the basic model with two theorems about its correctness and completeness. The proofs are given in our earlier paper [22] and in [23], respectively.

Theorem 1 *Every integer point satisfying the constraints corresponds to a correct global schedule.*

Theorem 2 *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.*

4.3. Objective Function

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.

To integrate this into the model, we introduce new binary variables B_t^A. A variable B_t^A is equal to one if

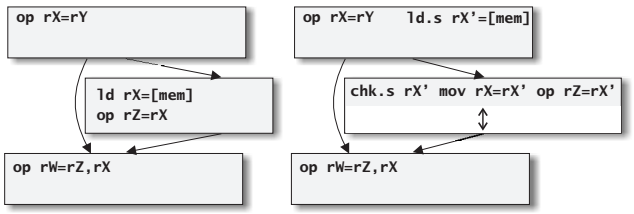

Figure 4. Control speculation with register renaming before (left) and after application (right)

and only if basic block A has length t in the schedule. This variable can be seen as the variable x_n^{At} of an imaginary "last" instruction n inside block A which is dependent on *all* other instructions there (with latency zero). Accordingly, we can use instances of the local precedence constraints (5) to link the B_t^A variables to the model. With the execution frequency of block A given as f_A, the objective function now can be written as:

$$\min \sum_{A \in \mathcal{B}} f_A \cdot \left(\sum_{t \in G(A)} t \cdot B_t^A \right) \tag{7}$$

If block A has length $\tilde{t}$, then the term in the brackets evaluates to $\tilde{t}$ since only $B_{\tilde{t}}^A$ is equal to one in the sum – then $\tilde{t} \cdot B_{\tilde{t}}^A = \tilde{t}$ is the only non-zero addend.

5. Extensions

The presented model is our basis for several essential extensions which incorporate different kinds of speculation and extend the range of code motion. Note that we can only outline the implementation of these extensions in this paper. For more details, please refer to [23, 22].

5.1. Speculation

Non-speculative instructions are limited in their scope of code motion as they must not be executed unnecessarily. There are several reasons why the unnecessary execution of an instruction could harm correctness: It could trigger a false exception, which concerns mostly memory instructions. Furthermore, it could overwrite a live value; this applies to stores and to instructions in a UD chain, i. e. several definitions that reach a common use.

The Itanium architecture has control speculative loads to overcome the first restriction [14]. Fig. 4 shows an example of their application[3]: On the right-hand side, the load is moved upwards as a control speculative version `ld.s` which sets a special bit in the target register if an exception occurs. The check instruction `chk.s` detects a deferred exception by testing this bit. If set, it branches to recovery code that re-executes the load (and possible additional speculative[4] uses which have been scheduled before the `chk.s`; not shown in the figure). The recovery code then eventually triggers the exception.

Furthermore, in the example from Fig. 4, the result register `rX` must not be written speculatively because the load is in a UD chain with "`op rX=rY`". We can overcome this second restriction by letting the load write to a new temporary register. *Exclusive* uses like the "`op rZ=rX`" can directly read the temporary register and can possibly also be speculated with the load, whereas for the "`op rW=rZ,rX`", we insert a `mov` instruction which moves the value back to the original register. All *non-exclusive* uses are dependent on this new `mov` instruction, which must – like the `chk.s` – be treated as a non-speculative instruction.

The scheme is flexible in the sense that it is also possible to speculate a definition inside a UD chain that is not a load and vice versa – the `chk.s` and the `mov` are then dropped, respectively. It also allows to cascade several dependent speculated instructions. If the load is predicated, both the `chk.s` and the `mov` inherit the predicate, while it can be left out for the the `ld.s`.

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

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 then replaced by $(1 - usespec)$ 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

3 We use assembly pseudo-ops in the examples; the result registers are on the left-hand side of the "=", the operand registers on the right-hand side. Those instructions which are executed in parallel are written in the same line.

4 Analogously to the term "non-speculative", we call instructions "speculative" if they can be executed speculatively.

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. *Data speculation* [14] is implemented very similarly.

In Fig. 4, the resulting schedule length reduction is depicted by the double-headed arrow inside the block. It has been achieved at the cost of two additional instructions plus recovery code.

It should be noted that the branch to recovery code is only taken if the load triggers an exception. In practice, this is a very rare event that happens in less than 0.001% of all cases [3]. Nevertheless, the use of control speculation should be guided by a cost model which estimates the failure probabilities of individual loads, and which also allows for the rare but significant penalties that a `ld.s` may incur if it misses the L1 cache or the TLBs [15]. This information – if available from static analysis, heuristics or profiling – can be integrated into the objective function of the model to increase its precision.

This information was not available during our experiments, but we have excluded code motion of speculative loads into blocks whose execution frequency is by a factor k times higher than that of the source block, i. e. we forbid control speculation which is likely to be useless (we used $k = 5$ in the experiments).

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 (where software pipelining is not used).

Two variants of **code motion into loops** are used: *Non-speculative upward* motion guards instructions with a predicate that is true only on loop exit. This introduces a dependence on the compare instruction computing the predicate. If we do not predicate the instruction, it is executed speculatively during each loop iteration (at most) although only the result of the last execution before the loop exit is actually being used. This sounds wasteful but it can still be profitable if instead a nop would be executed. This kind of code motion can only be allowed for speculative instructions (except loads) which

- do not write a value which is live at the loop header
- are *multiply executable* without a changing semantics, which is not the case for instructions where results and operands overlap, like `add r1=1,r1`, or with post-increment.

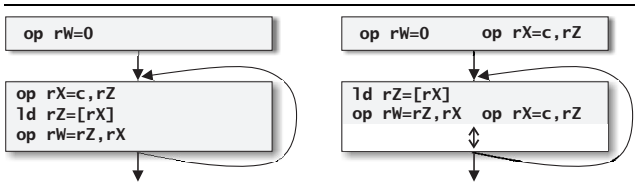

Figure 5. Example of cyclic code motion

Code motion out of loops if trivial if the code is loop invariant. Otherwise an instruction can still be hoisted upwards out of the loop if it is not only moved 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*. Cyclic motion can be profitable if the cyclically moved code is overlapped inside the loop body with instructions from the previous iteration, reducing the critical path here. Fig. 5 shows a small example of such a case (the "`op rX=c,rZ`" is cyclically moved).

Cyclic code motion can be regarded as a simple variant of software pipelining without fill overhead. In practice, there are still many loops where software pipelining can or should not be applied – for example if they contain other loops, function calls or complex control flow, or if they have low trip counts – cyclic code motion can then help alleviate the inefficiencies that static scheduling suffers from in these cases. However, it should be used carefully since it comes at the price of code expansion.

Currently we have integrated cyclic code motion into the ILP only upwards for speculative instructions (including those added in Sec. 5.1), and only out of the innermost loop the instruction is contained in. If this loop has the header block 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 then left open to the ILP solver.

We have decided to use the same x_n^{At} variables to model both normal and cyclic placement of an instruction. This prevents a potential duplication of these variables, but it also complicates the design of the global precedence and assignment constraints (see [22, 23] for details).

5.3. Partial-Ready Code Motion

In principle, partial-ready code motion [4] is a special form of control speculation: it speculates that a particular control flow path is taken and ignores the

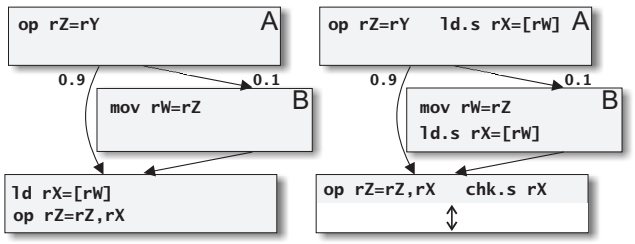

Figure 6. Partial-ready code motion before (left) and after (right) application

data dependences from other paths. Fig. 6 gives an example where this is profitable.

On the left-hand side we cannot move the load into block A because of the dependence on the `mov`. The load is ready for scheduling there only under the assumption that the left, likely path is taken ("partial-ready"). On the right-hand side we ignore the dependence by scheduling the load in block A, and we insert a compensation copy on the other path which re-executes it after the `mov`, overwriting `rX` with the correct value. An important detail is that a speculative load must be used in A because its load address is undefined if the path A-B is taken.

Generally, the idea is to schedule instructions earlier on a path by speculatively ignoring dependences from other paths, and to place compensation copies on the other paths which respect these dependences.

Some profound changes are necessary in the model to support this transformation since it violates two of its inherent assumptions: namely, that no instruction is placed twice on any path (enforced by constraints (2), cf. also Theorem 2; in the example violated by the `ld.s`), and that data dependences are never violated globally (constraints (4)). It is necessary to relax these assumptions.

This can be done for the first assumption by replacing the "=" by "≤" for specific instances of equation (2), in the example for the edge A-B. This increased flexibility also increases the search space and thereby the solution times. To cope with the growing complexity we currently impose several restrictions: Partial-ready code motion is only allowed for speculative instructions (including those added in Sec. 5.1) and their speculative uses; a possible combination with predication is not yet implemented. For instructions which support cyclic code motion, it is only applicable together with this transformation (which turns out to be a profitable combination in practice).

5.4. Branches and If-Conversion

Branches are modeled as special instructions that always appear in the last cycle of a basic block. If the ILP solver empties blocks, branches to these blocks may disappear and free execution slots. This did not happen rarely during our experiments, where almost 10% of all blocks were collapsed. Therefore we have developed a formulation (with little additional complexity) that models exactly the resulting changes in the branch structure (including choices for fall-through edges). It would also easily be possible to incorporate branch misprediction penalties.

Calls are treated almost like normal instructions, but are not subject to global code motion. Details are given in [23].

5.5. Subsequent Optimization Phases

A consequence of the used objective function is that the ILP solver has a blind spot for everything that is not related to the schedule length. This can result in schedules which are suboptimal with respect to other aspects like register usage. To compensate for this we can perform subsequent optimizations *while preserving the minimal schedule length.*

This can be done by solving a second ILP subsequently which is the same as the first one except that it has a different objective function and that the length of each block is fixed to its solution value of the first phase. We briefly sketch some possible objectives for the second phase; only the first one is currently used in the experiments.

- **Minimization of the instruction count**: Nothing detains the ILP solver from using more speculation and more compensation copies than necessary, as long as the resulting schedule is valid and optimal. Hence we use an objective function during the second phase that minimizes the number of scheduled instructions (i. e. the sum of all x_n^{At} variables); this takes only a few seconds for all input programs.
- **Reducing register pressure**: Long-range code motion increases the register pressure, and the first phase could use more of it than necessary. A subsequent phase could alleviate this by shrinking live ranges.
- **Stall minimization** expands the distances between loads and their nearest use. This reordering could utilize slack in the schedule to minimize the stall cycles due to cache misses.

6. Experimental Evaluation

We have implemented all described modelings and tested them on routines from the SPECint 2000 benchmark using a postpass approach. This method is too new and too expensive to optimize the whole SPEC benchmark with it; instead we had to select several routines according to the following criteria:

- The assembled routine should not have more than a few hundred instructions to limit the complexity.
- It should be hot so that the impact of the optimization can be measured. We have chosen only routines whose weight (i. e. the time spent in these routines) is at least 5%.
- It should not contain hot software-pipelined loops because software pipelining is currently not supported by the model.
- It should not contain too many long-latency (floating-point) instructions because the model is imprecise here: the local precedence constraints (5) can deal with latencies greater than one, but the global propagation of these latencies is not allowed for.

Table 1 shows the used input routines with their weights as measured with the Caliper tool [2]. "Ins. in" gives their instruction count. All functions are optimized as a whole except *prune_match* where we have omitted a large, cold part of the routine (the last six entries for *prune_match* in Table 1 refer to the optimized part). Table 2 shows the numbers of basic blocks and loops for each routine.

6.1. Experimental Setup

The selected routines were compiled to assembly with Intel's Linux compiler 7 for the Itanium 2. We used full optimization (`-O3`) and profiling information (`-prof_use`). The assembly files are directly input to our optimizer. The latter reconstructs control flow, data dependences and also reads the execution frequency estimates for the objective function which are annotated by Intel's compiler in the assembly code.

It then undoes all uses of control and data speculation (with manual interaction for some instances) and performs register renaming to remove all false dependences which would otherwise restrict code motion. The tool does not undo predication where it is used by Intel's compiler. In the input routines this feature is used rarely and conservatively so that we see no benefit in reversing these decisions.

The advantage of the postpass approach is that it allows the direct comparison with Intel's state-of-the-art compiler [10]. But during the comparison it should always be kept in mind that both code generators play in different leagues regarding the computation times.

A drawback of the postpass approach is that no information about memory disambiguation is available. Hence all memory dependences must be reconstructed conservatively. Regarding data speculation our policy is as follows: we include a possibility for data speculation in the ILP only if the two memory accesses are independent under the ANSI C aliasing rules. In these cases it is assumed that aliasing is unlikely so that the cost of recovering does not need to be taken into account. In only two routines more data speculation is used in the optimal schedule than before (*get_heap_head* and *add_to_heap*).

The tool then performs several fully 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. These and further optimizations are applied to the model to achieve acceptable solution times. They are the result of a long process of analyzing and experimenting [23].

The ILPs are then solved with CPLEX 8.0 [1] on a 900-MHz-UltraSparc III+. CPLEX uses the same settings for all input programs, there is no routine-specific tuning. Also there is no optimality tolerance interval granted to the ILP solver – only a 100% optimal result is accepted. Table 2 shows the ILP sizes, the numbers of branch-and-bound nodes and the solution times. To keep the ILP sizes small, the value $\mathbb{G}_A$ – the number of cycles reserved for a basic block – is chosen pragmatically: it is set to the length of A in the input schedule plus a constant reserve (usually $k = 1$, for six blocks it has been extended to 2).

After CPLEX has finished, the optimal schedule is constructed from the delivered solution. It is then passed to a bundler which generates the final assembly output. The bundler does not use the ILP solver; it is based on precomputed results and dynamic programming [17] and incorporates all disclosed information about bundling for the Itanium 2 [15]. Finally, recovery code is added manually.

6.2. Results and Analysis

Table 1 shows the achieved reductions of the global schedule lengths in column "Static Red.". Sometimes the reductions are considerable like for *longest_match* and *get_heap_head*; in other cases the critical path is a limiting factor like in *deflate* and *add_to_heap*, but the percentages are still respectable here with 19% and

Routine	Program	Input Set	Weight	Speedup Program	Speedup Routine	Static Red.	Ins. in	Ins. out	Delta Ins.	Delta Bundl.	Weigh. IPC in	Weigh. IPC out
longest_match	gzip	program	68%	28.97%	43%	44%	191	230	20%	7%	2.4	5.4
deflate	gzip	random	14%	1.72%	12%	19%	226	233	3%	-3%	2.6	3.6
send_bits	gzip	graphics	15%	3.05%	20%	30%	86	95	10%	3%	2.6	4.7
firstone	crafty	ref	10%	0.88%	9%	37%	37	42	14%	0%	2.6	4.8
get_heap_head	vpr	route/ref	30%	4.25%	14%	43%	71	94	32%	9%	2.3	4.6
add_to_heap	vpr	route/ref	13%	1.17%	9%	17%	108	119	10%	4%	3	4.1
qSort3	bzip2	ref	12%	1.93%	16%	26%	241	279	16%	4%	2.9	4.5
xfree	parser	ref	10%	0.76%	7%	22%	46	50	9%	-5%	2.3	3.6
prune_match	parser	ref	6%	0.73%	12%	41%	69	84	22%	-3%	2.5	5.4
	Average				**16%**	**31%**			**15%**	**2%**	**2.6**	**4.5**

Table 1. Results of the optimization

Routine	#BB	#Loops	Spec. in	Spec. poss.	Spec. out	#Constraints	#Variables	#Nodes	Sol. Time / Seconds
longest_match	26	2	15	47	24	5619	2865	500	141
deflate	37	3	4	28	7	4570	2686	2	3
send_bits	12	0	0	10	1	2583	1417	8	4
firstone	8	0	0	7	5	458	277	0	0
get_heap_head	9	2	3	23	11	4126	1673	1	13
add_to_heap	12	1	2	16	5	3248	1665	0	2
qSort3	22	4	7	44	18	10723	4984	914	179
xfree	9	1	2	7	4	759	403	6	0
prune_match	10	1	4	19	11	1294	766	2	1

Table 2. Further characteristics of the input routines and the solution process

17%, respectively. Later we will examine how these improvements could have been achieved.

Tab. 2 displays the number of control and data speculative loads used in the input schedule ("Spec. in"). The next two columns show how often speculation according to Sec. 5.1 is included as a possibility in the ILP and how many of these possibilities are actually utilized in the output schedule (44% on the average), respectively. This includes not only speculative loads but also speculated instructions from UD chains. Counting only uses of control and data speculation yields a 60% increase compared to the input schedules.

The schedule length reductions are accompanied by a drastically increased static instructions-per-clock rate: the average IPC (without nops) weighted by the execution frequency of the blocks goes up from 2.6 to 4.5 (the unweighted IPC from 2.5 to 3.8). This shows that the ILP scheduler is successful in extracting more parallelism and approaches the maximum IPC of six for the Itanium 2.

However, the uninhibited use of free issue ports also results in an increased instruction count by 15% on average ("Delta Ins.", without nops and recovery code). This is potentially harmful to the instruction cache efficiency. But interestingly, the relevant number of bundles grows only by 2% ("Delta Bundl.").

This comes from the fact that the new schedules use fewer, but larger instruction groups which fit much better into the bundling scheme of this architecture. Small groups are more often forced to be filled up with nops. Hence most new instructions move into execution slots which were previously occupied by nops. With the small code size increase the negative impact on the instruction cache should be very limited.

All benchmark programs were run on a 1.4 GHz Itanium 2. Table 1 shows the speedups of these programs when the routine is replaced by its optimized variant. These numbers are sometimes less than 1% because only a single small routine has been changed, therefore several runs were performed to determine them

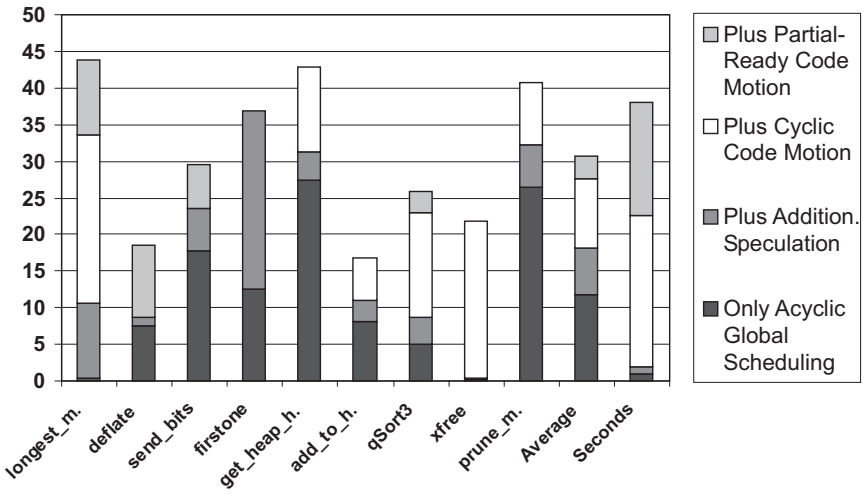

Figure 7. Schedule length reductions as different extensions are switched on incrementally

precisely. In some cases we have used those input sets where the weight of the routine is maximal.

From the program speedup and the weight we can derive the speedup of the individual routines ("Speedup Routine"). It can be seen that the runtime impact of the static improvements varies widely among the routines, which can be attributed to different stall characteristics. The impact ranges from about one third for routines with a relatively high average memory latency like *xfree* to more than two thirds for the compute-intensive, cache-friendly routines from *gzip*, with the average at half. The latter is plausible because we currently only optimize the unstalled execution time which is about half of the total execution time for SPECint 2000 [18].

Finally, Figure 7 shows how the schedules shrink as the extensions additional speculation (i. e. more than in the input schedule), cyclic code motion and partial-ready code motion are switched on incrementally (in this order). All these features improve only a subset of the routines, but on the average, each is essential. The last bar shows the accompanying increase in the average solution time (the y-axis displays both percents and seconds). While scheduling with speculation generally can be solved within a few seconds, the two other extensions cause search space expansions especially for two of the larger routines (see Table 2). It is likely that the efficiency of the ILP model can still be improved here.

7. Applications

The experimental results have shown that the ILP method is attractive as an *optimization tool* for compute-intensive application kernels like compression and encryption routines. Not less interesting is its application as a *research tool* to obtain insights into the potential of EPIC architectures: for instance, we can evaluate the impact of microarchitectural changes on performance without compiler influence – in contrast to scheduling heuristics, it is simple to model architectural restrictions and asymmetries with this method and to obtain schedules that account for them optimally. Another unique property of this approach is that the researcher can formulate different optimization goals in the objective function without caring about how to achieve them. This can greatly facilitate experimenting with different optimization objectives.

A further application arises from the fact that we have proven the correctness of the basic model [17, 23]. It follows that a schedule is proven to be 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 be used *to validate the schedules produced by heuristics*. It is an inherent advantage of this approach which does not build on an algorithm, but on a precise mathematical model.

8. Conclusion and Outlook

We have modeled and solved global scheduling for the Itanium 2 using integer programming. Our formulation comprises generation of compensation code, support for partial-ready and cyclic motion, control and data speculation and predication. To our knowledge, we are the first to provide an exact solution to this problem.

Our experiments have shown that this method can reduce the schedule lengths produced by Intel's compiler by about 20-40%. The tested routines may be not representative enough to draw final conclusions from these numbers. Nevertheless, the extent of the improvements in both schedule length and IPC indicates that there is still some considerable performance headroom waiting to be unleashed in some tasks that are very fundamental to EPIC: static scheduling and use of speculation.

Currently we are studying how the efficiency of the model can be improved further and how it can be modified to support software pipelining.

9. Acknowledgements

This research has been funded by the graduate studies program "Quality Guarantees for Computer Systems" supported by the Deutsche Forschungsgemeinschaft. Thanks go to Ingmar Stein who has implemented the bundler used in the experiments. I am also grateful to the Max-Planck-Institut für Informatik, Saarbrücken, for giving access to their CPLEX installation.

References

[1] ILOG CPLEX 8.0, 2002. www.cplex.com.

[2] HP Caliper, 2003. www.hp.com/go/caliper.

[3] D. Alpert. Itanium Processor Status Report. *Microprocessor Report*, July 2003.

[4] J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. *Journal of Instruction-Level Parallelism*, 1(6):1–6, 2000.

[5] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. 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.

[6] 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, Nov. 1997.

[7] 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*, 2(4):456–471, Dec. 1994.

[8] D.-Y. Chen, L. Liu, C. Fu, S. Yang, C. Wu, and R. Ju. Efficient Resource Management during Instruction Scheduling for the EPIC Architecture. In *Proceedings of the PACT 2003*, New Orleans, Sept. 2003.

[9] G. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In T. C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley, New York, 1951.

[10] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An Overview of the Intel® IA-64 Compiler. *Intel Technology Journal*, (Q4), 1999.

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

[12] S. Haga and R. Barua. EPIC Instruction Scheduling Based on Optimal Approaches. *Proceedings of the EPIC-1 Workshop*, Dec. 2001.

[13] 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.

[14] Intel. *Intel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture*, Oct. 2002.

[15] Intel. *Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization*, Apr. 2003.

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

[17] D. Kästner and S. Winkel. ILP-based Instruction Scheduling for IA-64. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Snowbird, June 2001.

[18] J. McCormick and A. Knies. A Brief Analysis of the SPEC CPU2000 Benchmarks on the Intel® Itanium® 2 Processor. *HotChips 14*, 2002.

[19] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[20] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, Berlin; Heidelberg; New York, 2003.

[21] T. Wilson, G. Grewal, and D. 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.

[22] S. Winkel. Optimal Global Scheduling for Itanium Processor Family. In *Proceedings of the EPIC-2 Workshop*, Istanbul, Nov. 2002.

[23] S. Winkel. *Optimal Global Instruction Scheduling for the Itanium Processor Architecture*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004. To appear.