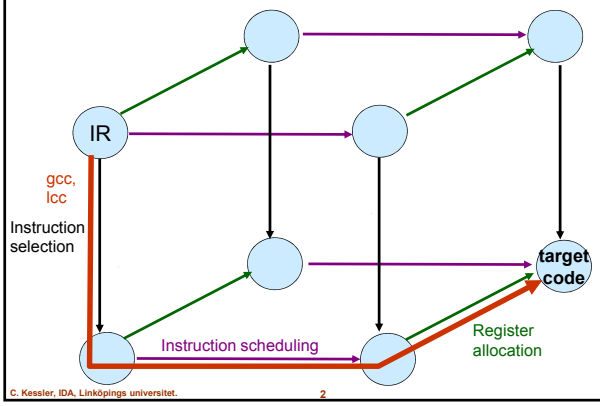
DF00100 Advanced Compiler Construction
TDDC86 Compiler Optimizations and Code Generation

Integrated Code Generation

- Phase ordering problems
- Clustered VLIW Processors: More phase ordering problems
- Integrated Code Generation
- Our research project: OPTIMIST

Christoph Kessler, IDA, Linköpings universitet, 2009.

Phase ordering problems

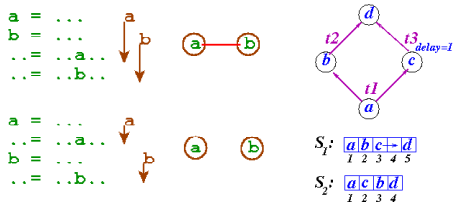


Phase ordering problems (1)

Instruction scheduling vs. register allocation

(a) Scheduling first:
determines Live-Ranges
→ Register need,
possibly spill-code to be
inserted afterwards

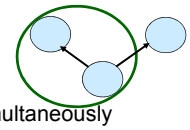
(b) Register allocation first:
Reuse of same register by different values introduces "artificial" data dependences
→ constrains scheduler



Phase ordering problems (2)

Conflicts Instruction selection ←→ Scheduling / Reg. alloc.

- **Selection first:** Cost attribute of a pattern covering rule is only a coarse estimate of the real cost (effect on e.g. overall time)

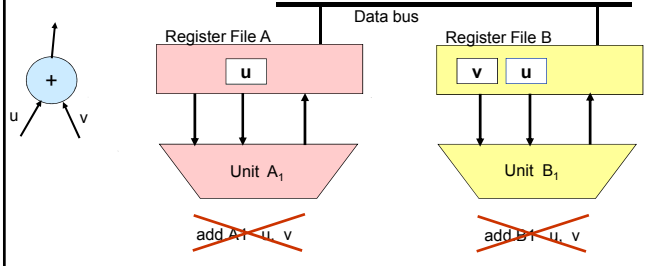


  Real cost based on
  - currently free functional units
  - other instructions ready to execute simultaneously
  - pending latencies of already issued but unfinished instructions

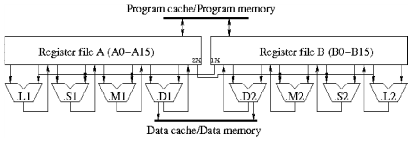
  → Integration with instruction scheduling desirable
- **Mutations** with different resource requirements
  - a = 2 * b oder a = b << 1 oder a = b + b ?
- Different instructions with different register need

Clustered VLIW processor

- E.g., TI C62x, C64x DSP processors
- Register classes
- Parallel execution constrained by operand residence

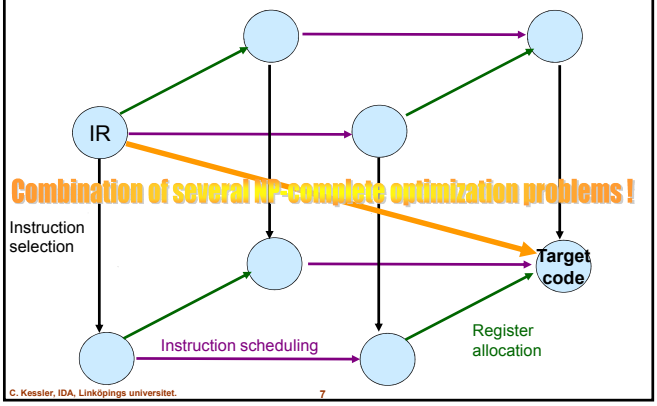


More phase ordering problems

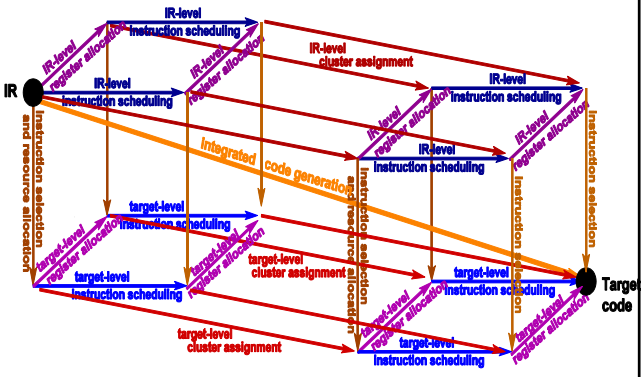


- In parallel e.g.: load on A || load on B || move A→B
- Mapping instructions → cluster
  - should preferably know already beforehand about free move-slots in schedule...
- Instruction scheduling
  - must know mapping to generate moves where needed
- Heuristic [Leupers'00]
  - iterative optimization by simulated annealing

Integrated Code Generation for non-clustered architectures



Integrated Code Generation for Clustered VLIW Architectures

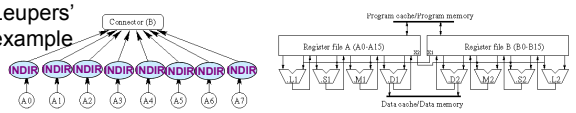


Integrated Code Generation Some Approaches

- **Weak integration / Phase coupling**
  - E.g., register pressure aware scheduler [Goodman/Hsu '88], HRMS…
- **Heuristic Phase Interleaving**
  - E.g., alternating partitioning / scheduling for clust. VLIW [Leupers'00]
- **Integration of 2 phases**
  - Instruction selection and register allocation
    - DP (Dynamic pr.) for space optimization, e.g. [Aho/Johnsson '77]
    - DP for space- or time optimization, e.g. [Fraser et al.'92]
  - Instruction scheduling and register allocation
    - ILP (Integer Linear Progr.) e.g. [Kästner'00]
- **Integration of 3 and more phases**
  - ILP for simple, non-pipelined RISC processor [Wilson et al.'94]
  - DP, for clustered VLIW: OPTIMIST [K., Bednarski'06]
  - ILP, for clustered VLIW: Integrated software pipelining [Eriksson, K.'09]

Example

- Clustered 8-issue VLIW processor TI C6201
- Leupers' example



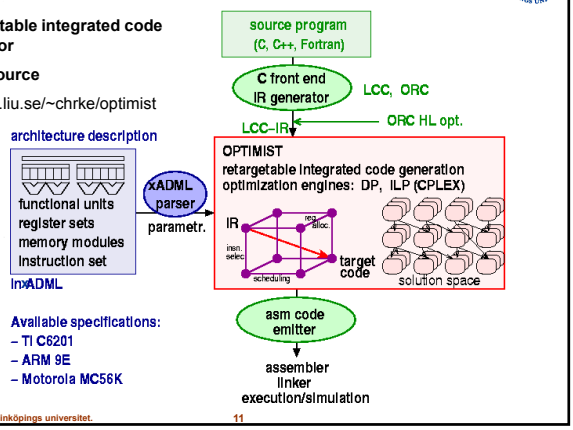
TI-C comp.	Schedule by Leupers' heuristic	Optimal schedule by OPTIMIST
LD *A4,B4	LD *A0,A8 || MV A1,B8	LD *A0,A8 || MV A1,B8
LD *A1,A8	LD *B8,B1 || LD *A2,A9 || MV A3,B10	LD *B8,B1 || LD *A2,A9 || MV A3,B10
LD *A3,A9	LD *B10,B3 || LD *A4,A10 || MV A5,B12	LD *B10,B3 || LD *A4,A10 || MV A5,B12
LD *A0,B0	LD *B12,B5 || LD *A6,A11 || MV A7,B14	LD *B12,B5 || LD *A6,A11 || MV A7,B14
LD *A2,B2	LD *B14,B7	LD *B14,B7 || MV A8,B0
LD *A5,B5	MV A8,B0	MV A9,B2
LD *A7,A4	MV A9,B2	MV A10,B4
LD *A6,B6	MV A10,B4	MV A11,B6
NOP	MV A11,B6	NOP
MV A8,B1		
MV A9,B3		
MV A4,B7		
12 cycles	9 cycles	9 cycles (15min)

Our project: OPTIMIST

Retargetable integrated code generator

Open Source

www.ida.liu.se/~chrke/optimist



Processor specification language xADML

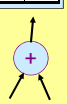
[A. Bednarski, PhD thesis, Linköping Univ., 2006]

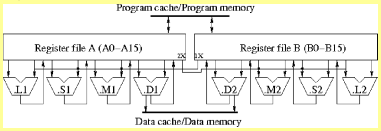
```
<architecture omega="8">
 <registers> ... </registers>
 <residenceclasses> ... </residenceclasse
 <funits> ... </funits>
 <patterns> ... </patterns>
 <instruction_set>
  <instruction id="ADDP4" op="4407">
   <target id="ADD .L1" op0="A" op1="A" op2="A" use_fu="L1"/>
   <target id="ADD .L2" op0="B" op1="B" op2="B" use_fu="L2"/>
   ...
  </instruction>
  ...
  <transfer>
   <target id="MOVE" op0="A" op1="B">
    <use_fu="X2"/>
    <use_fu="L1"/>
   </target>
   ...
  </transfer>
 </instruction_set>
</architecture>
```

Specify reservation tables by

```
<cycle_matrix>
...
</cycle_matrix>
```

OPx	L1	L2	X2
t+2	X		
t+1	X	X	
t	X		X





DF00100 Advanced Compiler Construction
TDDC86 Compiler Optimizations and Code Generation

APPENDIX

Optimal Integrated Code Generation with OPTIMIST

Christoph Kessler, IDA,
Linköpings universitet, 2009.

Optimal Integrated Code Generation ...

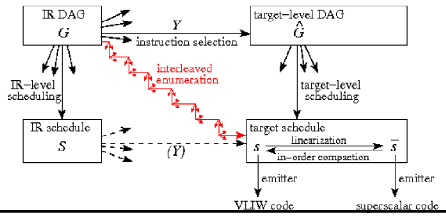
- Why "Optimal" ?
  - Quantitative assessment of heuristics
    - absolute instead of relative comparison
  - Feedback to architecture design of an application specific processor
  - Often feasible for small, sometimes not so small instances
    - thanks to modern solver technology and problem transformations
  - Scientific challenge

Optimal Method 0: Exhaustive Enumeration

For all possible instruction selections
  For all possible schedules with each selection
    For all possible moves of live values ...
      Evaluate resulting code, keep optimum

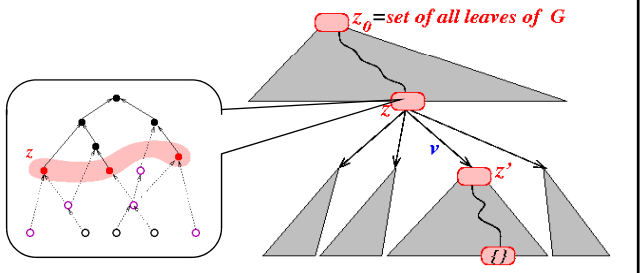
Any ordering of these loops is possible...

- **Variant**: Interleaved exhaustive enumeration

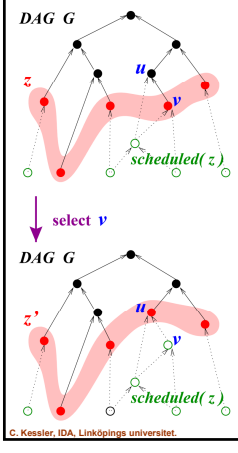


Interleaved Exhaustive Enumeration

- Explores a decision tree (enumeration tree, backtracking tree) with partial solutions (= code for sub-DAGs)
- Example: (only) instruction scheduling = topological sorting



Recall: List Scheduling = Local Scheduling by Topological Sorting



top_sort(Set z, int[] $INDEG$, int t)
if $z \neq \emptyset$ // $(t \leq n)$
  select arbitrary node $v \in z$;
  // *implicitly remove all edges* (v,u) $\forall u$:

$$INDEG'(u) = \begin{cases} INDEG(u) - 1 & \text{where } \exists (v,u) \\ INDEG(u) & \text{elsewhere} \end{cases}$$

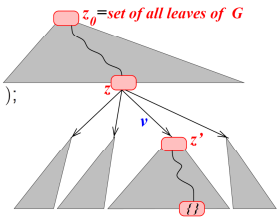
  // *update zero-indegree set:*
  $z' \leftarrow z - \{v\} \cup \{\text{new leaves}\}$
    $= \{u : INDEG(u) = 0\}$
  $S[t] \leftarrow v$;
  top_sort(z', $INDEG'$, $t+1$);
else output $S[1:n]$ fi

Call *top_sort*(z_0, $INDEG_0$, 1)
produces a schedule in $S[1:n]$

Graph-based method: Naive Enumeration, Selection Tree (1)

all_top_sort(Set z, int[] $INDEG$, int t)
if $z \neq \emptyset$ // $(t \leq n)$
  for all $v \in z$ do
    (z', $INDEG'$) $\leftarrow$ *selection*(v,z, $INDEG$);
    $S[t] \leftarrow v$;
    all_top_sort(z', $INDEG'$, $t+1$);
  od
else output $S[1:n]$;

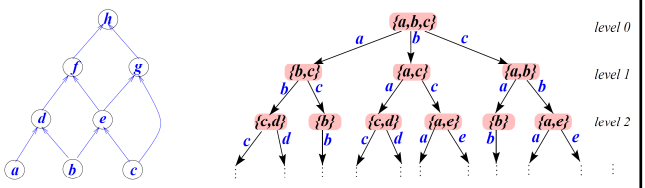
Call *all_top_sort*(z_0, $INDEG_0$, 1)
enumerates all topological sortings of DAG G



selection tree T
nodes = inst. of zero-indeg sets
edge (z,z'), labeled by v,
iff $(z',\ldots) \leftarrow$ *selection*$(v,z,\ldots)$

Time: $O(n \cdot$ # enumerated schedules$) = O(n \cdot n!)$

Naive Enumeration, Selection Tree (2)

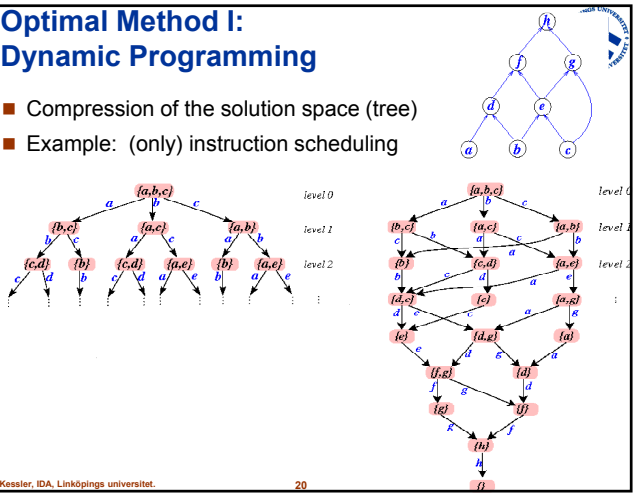


Observations:

- in general: multiple instances of same zero-indeg set z
- all paths to instances of the same zero-indeg set have equal length, as the same set $scheduled(z)$ of DAG nodes has been scheduled
- labels on a path π from z_0 to node z correspond to a schedule S of $scheduled(z)$ and vice versa

Optimal Method I: Dynamic Programming

- Compression of the solution space (tree)
- Example: (only) instruction scheduling



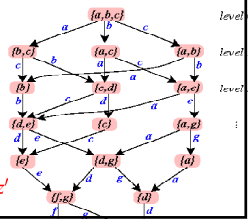
From Selection Tree to Selection DAG (2)

Idea: Summarize different instances of ***same*** zero-indeg set z in ***one*** selection node

$\rightarrow$ selection tree T becomes selection DAG D

For all schedules S for the same zero-indeg set z the corresponding paths π_S in D end up in the same node.

Equal path lengths
$\rightarrow$ selection DAG D is leveled
in $n+1$ levels $L_0, L_1, ..., L_n$
with $L_0 = \{z_0\}$ and $L_n = \{\emptyset\}$



\+ search in only one level for existence of z'

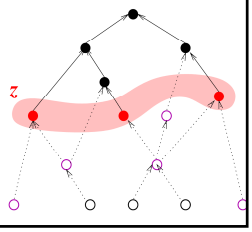
From Selection Tree to Selection DAG (3)

Lemma: It is sufficient to store for each selection node z ***one*** schedule S_z that is optimal for z.

Proof (idea): For each selection DAG path π ending in the same zero-indeg set z holds after execution of the (prefix) schedule S_π corresponding to π:
The same values reside in registers, namely
$alive(z) = \{u \in scheduled(z) : \exists (u,v) \in E, v \notin scheduled(z)\}$

Hence we may choose any path $\pi_{S'}$ to z that minimizes the register need $m(S')$.

By induction follows:
The schedule S_0 stored in selection node $\emptyset$ is optimal for G.



Level-wise construction of the selection DAG

$L_i \leftarrow$ *empty List* $\forall i > 0$
$L_0 \leftarrow$ **new List**(z_0)
for L_i **from** L_0 **to** L_{n-1} **do**
  for all $z \in L_i$ **do**
    for all $v \in z$ **do**
      $(z', ...) \leftarrow$ ***selection*** $(v, z, ...)$; $S \leftarrow S_z \bowtie \{v\}$;
      $z'' \leftarrow L_{i+1}$.***lookup***(z');
      if $(!z'')$ L_{i+1}.***insert***(z'); $S_{z'} \leftarrow S$;
      else if $m(S) < m(S_{z''})$ $S_{z''} \leftarrow S$ **fi fi**
od od od
Output: S_0 in L_n.***first***$()$;

Time: with linear search $O(n \cdot 2^{2n})$, with search tree (lex. order) $O(n \cdot 2^n)$
- Selection DAG has $\leq 2^n$ selection nodes
- Comparison $z_1 \leq_{lex} z_2$ costs $O(n)$

Improvement

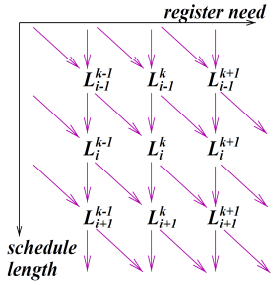
Partition the L_i into sublists $L_i^0, L_i^1, ..., L_i^{MAXREG}$
with $L_i^k = \{z \in L_i : S_z$ needs k registers $\}$

Lemma: The predecessors z of a selection node $z' \in L_{i+1}^k$ are located either in L_i^{k-1} or in L_i^k.

$\rightarrow$ partial order among the L_i^k

$\rightarrow$ change order of construction:
level-wise $\rightarrow$ columnwise

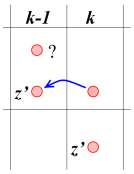
\+ need not search entire L_{i+1}
\+ stop construction as soon as first nonempty L_n^k has been reached



Modified algorithm

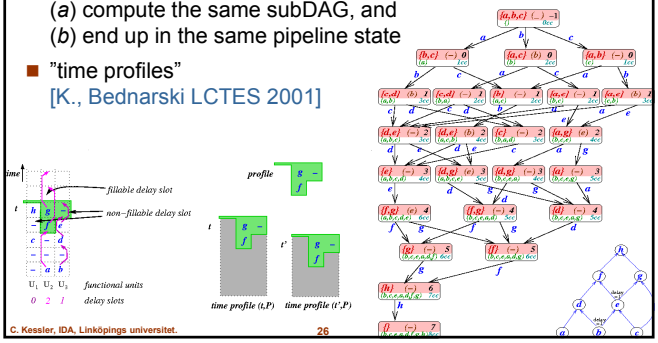
$L_i^k \leftarrow$ *empty list* $\forall i \ \forall k$; $L_0^0 \leftarrow$ **new List**(z_0);
for $k = 1,...,MAXREG$ **do**
  for L_i^{k-1} **from** L_0^{k-1} **to** L_{n-1}^{k-1} **do**
    for all $z \in L_i^{k-1}$ **do**
      for all $v \in z$ **do**
        $(z',...) \leftarrow$ ***selection*** $(v,z,...)$; $m \leftarrow$ reg. need of $S = S_z \bowtie \{v\}$;
        if $(!L_{i+1}^{k-1}$.***lookup***$(z'))$
          if $m = k-1$
            L_{i+1}^k.***remove***(z'); // *(if any)*
            L_{i+1}^{k-1}.***insert***(z'); $S_{z'} \leftarrow S$;
          else if $(!\ L_{i+1}^k$.***lookup***$(z'))$
            L_{i+1}^k.***insert***(z'); $S_{z'} \leftarrow S$; **fi fi**
  od od od
  if L_n^{k-1}.***notempty***$()$ output schedule S_0 in L_n^{k-1}.***first***$()$;
od

+ stop as early as possible
+ experimental results:
  practical up to 40...50 nodes
+ parallelizable:
  (a) diagonalisation across L_i^k
  (b) parallel lists L_i^k

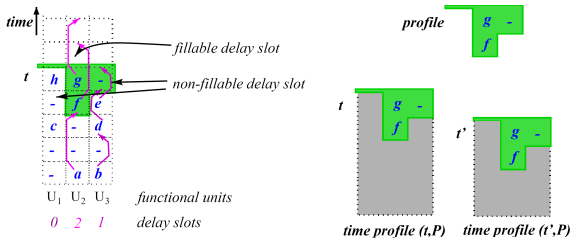


Dynamic Programming, Regular VLIW / Superscalar Processor

- **Compression Theorem I**:
  Two partial solutions (codes) are **equivalent**, if they
  (*a*) compute the same subDAG, and
  (*b*) end up in the same pipeline state
- "time profiles"
  [K., Bednarski LCTES 2001]



Time profiles

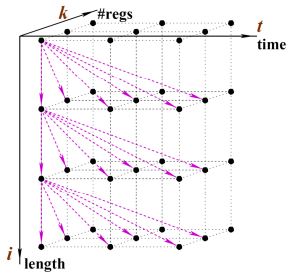


Profile P: window of the instructions scheduled last for each unit that may still influence future scheduling decisions.

Extended selection node (z,t,P)
summarizes all schedules of $scheduled(z)$ that end with time profile (t,P).

Time-inferior extended selection nodes can be pruned.

Simultaneous space and time optimization with time profiles



We observe: By appending a DAG node v to a schedule S we have

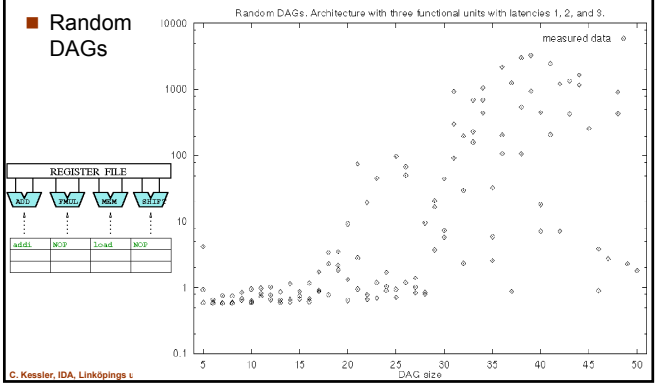
$$time(S) \le time(S \bowtie \{v\}) \le time(S) + 1 + MAX\_DELAY$$

$\rightarrow$ partition lists L_i^k in sublists $L_i^{k,t}$

Order of construction of the $L_i^{k,t}$ according to desired optimization goal

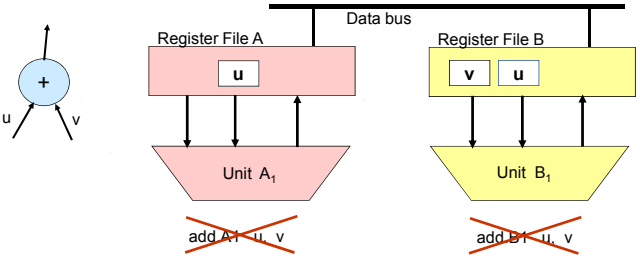
Results (1) – DP

- Regular 3-issue VLIW Processor (no MOVEs)
- Random DAGs



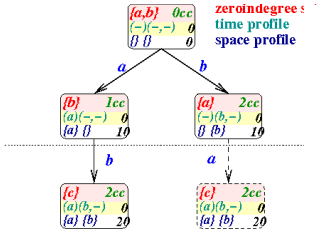
Dynamic Programming, Clustered VLIW Processor (1)

- Register classes / Residence classes
- Instruction selection constrained by operand residence
- Add Move-instructions speculatively → more solutions...



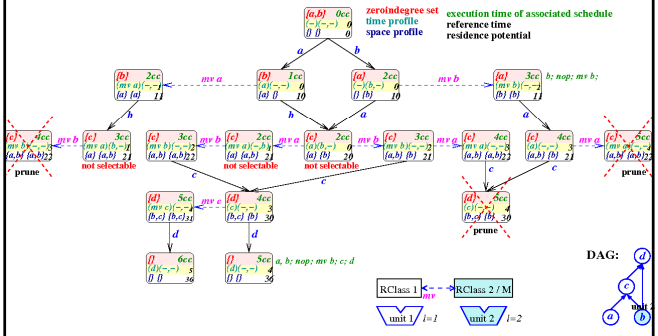
Dynamic Programming, Clustered VLIW Processor (2)

- **Compression Theorem II:**
  Two partial solutions (codes) are **equivalent**, if they
  (*a*) compute the same subDAG and
  (*b*) end up in the same pipeline-state and
  (*c*) hold the live values at the end in the same residence classes



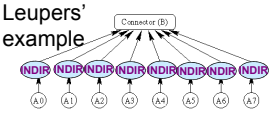
Dynamic Programming, Clustered VLIW Processor (3)

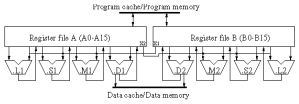
- Partial solutions for (subDAG, pipeState, LiveV-Residences)



Results (2) – DP

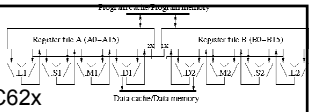
- Clustered 8-issue VLIW Processor TI C6201
- Leupers' example





TI-C comp.	Schedule by Leupers' heuristic	Optimal schedule by OPTIMIST
LD *A4,B4	LD *A0,A8 || MV A1,B8	LD *A0,A8 || MV A1,B8
LD *A1,A8	LD *B8,B1 || LD *A2,A9 || MV A3,B10	LD *B8,B1 || LD *A2,A9 || MV A3,B10
LD *A3,A9	LD *B10,B3 || LD *A4,A10 || MV A5,B12	LD *B10,B3 || LD *A4,A10 || MV A5,B12
LD *A0,B0	LD *B12,B5 || LD *A6,A11 || MV A7,B14	LD *B12,B5 || LD *A6,A11 || MV A7,B14
LD *A2,B2	LD *B14,B7	LD *B14,B7 || MV A8,B0
LD *A5,B5	MV A8,B0	MV A9,B2
LD *A7,A4	MV A9,B2	MV A10,B4
LD *A6,B6	MV A10,B4	MV A11,B6
NOP	MV A11,B6	NOP
MV A8,B1		
MV A9,B3		
MV A4,B7		
12 cycles	9 cycles	9 cycles (15min)

Results (3) – DP



- Clustered VLIW Processor TI C62x
- TI C64x DSP Benchmarks and FreeBench

Measure for degree of compression

Basic block	$\|V\|$	time[s]	space[MB]	#Impr.	#merged	#ESNodes
cplxinit	14	201	38	0	140183	44412
vectcopy init	12	16	11	26	61552	12286
vectcopy loop	14	47	15	58	149899	29035
matrixcopy loop	18	18372	194	1782	5172442	722012
vectsum loop	12	11	10	9	36715	8896
vectsum unrolled	17	1537	58	3143	1316684	198571
matrixsum loop	17	10527	154	2898	3502106	564058
dotproduct loop	17	3495	77	4360	2382094	345256
codebk_srch bb33	17	431	41	306	319648	64948
codebk_srch bb29	13	7	12	0	17221	6133
codebk_srch bb26	11	11	13	144	73761	19275
vecsum_c bb6	20	9749	154	5920	3744583	499740
vec_max bb8	13	79	35	0	99504	37254

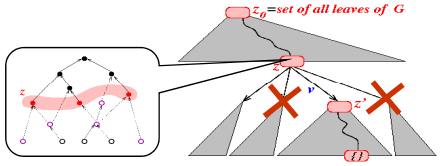
Results (4) – DP

- Single-issue vs. Single-cluster vs. Double-cluster architecture

benchmark	BB	size	ARM9TI-ARM mode t[s]	#merged	#ESnodes	TI-C62x single-cluster t[s]	#merged	#ESnodes	TI-C62x t[s]	#merged	#ESnodes
analyzer	51	34	98.9	546250	62965	12623.2	2547538	630132	—	—	—
analyzer	203	31	56.5	309673	49833	14.1	34559	29780	—	—	—
bmmse	4	14	0.1	489	165	0.1	523	375	10.9	15215	7042
bmmse	10	17	0.3	2558	582	0.8	5349	2413	6404.2	860455	204056
bmmse	11	14	0.1	768	234	0.2	825	562	56.8	38938	15557
codebk_srch	12	15	0.1	595	236	0.2	658	678	163.4	83220	33667
codebk_srch	16	21	0.7	6880	1512	8.3	47665	19613	—	—	—
codebk_srch	20	23	2.2	20380	3723	45.0	240079	76005	—	—	—
codebk_srch	21	42	178.5	783851	98580						
fir_vselp	6	23	0.5	4479	1215	4.4	20483	10331	—	—	—
fourinarow	6	29	0.2	862	561	1.2	4976	2214	10.0	23599	9203
irr	4	21	0.4	4526	1016	5.8	40477	12036	—	—	—
irr	7	38	2.1	8753	3259	38.0	87666	47233	—	—	—

Heuristic Methods

- Heuristic Pruning
  - Limit number of considered schedules, moves



- Genetic Programming

Results (5) – Heuristic Pruning

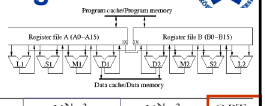
- Regular VLIW processor

program	BB	size	OPT time [s]	OPT qual. [cc]	H1 time [s]	H1 qual. [cc]	H2 time [s]	H2 qual. [cc]	H3 time [s]	H3 qual. [cc]
fir_vselp.c	6	23	185.4	(23)	1.4	(28)	7.2	(24)	31.3	(23)
scalarprod_un1.c	2	25	10.0	(27)	0.5	(32)	2.2	(27)	6.6	(27)
cplx.c	3	25	2.9	(29)	<0.1		<0.1		<0.1	
dot.c	3	17	1.9	(18)	<0.1		<0.1		<0.1	
dot_un1.c	3	25	10.2	(27)	0.5	(32)	2.4	(27)	6.9	(27)
dot_un2.c	3	33	1434.1	(36)	1.7	(42)	10.6	(36)	77.2	(36)
dot_un3.c	3	41			6.1	(52)	43.4	(45)	1063.7	(45)
summatrix.c	4	17	8.0	(17)	<0.1		<0.1		<0.1	
summatrix_un1.c	4	20	51.9	(31)	1.5	(37)	8.0	(31)	30.0	(31)
summatrix_un2.c	4	27	131.8	(26)	4.6	(31)	19.2	(26)	69.6	(26)
summatrix_un3.c	4	32	780.0	(31)	21.4	(37)	83.2	(31)	308.4	(31)

Hx = OPTIMIST with heur. limitation to x alternatives; qual. = schedule quality (time)

Results (6) – Heuristic Pruning

- TI C62x Clustered VLIW DSP

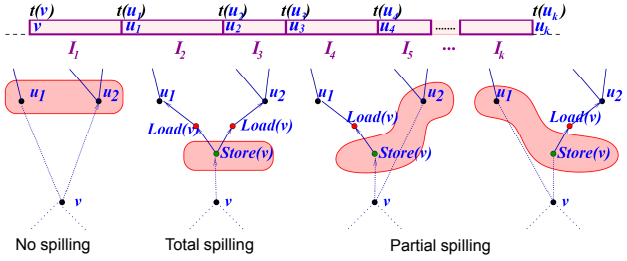


benchmark	BB	size	$H^{N=1}_{M=1}$ t[s]	$H^{N=1}_{M=1}$ τ[cc]	$H^{N=1}_{M=2}$ t[s]	$H^{N=1}_{M=2}$ τ[cc]	$H^{N=2}_{M=1}$ t[s]	$H^{N=2}_{M=1}$ τ[cc]	$H^{N=3}_{M=1}$ t[s]	$H^{N=3}_{M=1}$ τ[cc]	$H^{N=2}_{M=2}$ t[s]	$H^{N=2}_{M=2}$ τ[cc]	OPT τ[cc]
bmmse	4	14	0.1	18	0.1	18	0.5	9	1.2	9	1.0	9	9
bmmse	10	17	0.4	23	0.5	23	4.2	14	5.2	9	4.5	14	9
bmmse	11	14	0.3	18	0.4	18	1.3	10	4.8	9	1.5	10	9
codebk_srch	12	15	0.1	27	0.1	27	1.3	19	4.6	15	1.9	19	15
codebk_srch	20	23	0.2	34	0.2	34	16.9	21	5816.2	19	24.0	21	–
codebk_srch	25	11	0.1	16	0.1	16	0.1	11	0.2	7	0.2	11	7
codebk_srch	29	10	0.1	13	0.1	13	0.1	7	0.1	7	0.2	7	7
codebk_srch	31	15	0.1	15	0.1	15	0.1	14	0.2	14	0.1	14	14
codebk_srch	34	19	0.2	31	0.2	31	12.4	17	21.7	16	21.1	17	–
codebk_srch	42	14	0.1	19	0.1	19	0.9	14	2.1	14	1.6	14	14
fir_vselp	10	17	0.2	24	0.5	24	2.2	15	6.8	15	11.0	15	–
irr	4	21	0.1	27	0.1	27	1.2	13	19.8	13	1.5	13	–
mac_vselp	3	23	0.4	35	0.9	35	20.3	26	155.9	26	56.9	26	–
vec_sum	3	17	0.2	29	0.6	29	5.1	15	35.2	15	30.1	15	15

Extensions (2)

See M. Eriksson, PhD thesis, Linköping U., 2011

Optimal scheduling of spill code [K.'00]

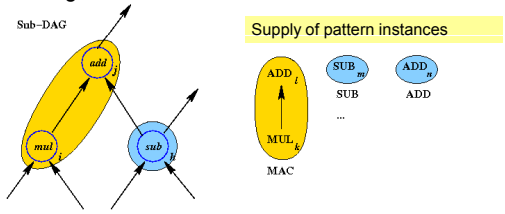


(Limited) admission of recomputations (Rematerialization)

[Paul, Tarjan '78]: ex. class of DAGs: linear decrease of $m \rightarrow$ expon. $time(S)$

Optimal Method II: Integer Linear Programming

- Model by 0/1-variables e.g. $c_{i,p,k,t}$ und linearen Constraints
- Example: covering constraint for instruction selection



for all i in V_G, $\Sigma_{\text{Musterinstanzen } p} \Sigma_{k \text{ in } p} \Sigma_{t=1...Tmax}\ c_{i,p,k,t} = 1$ (1)

ILP-Model (1)

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

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

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

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

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

$$\forall p \in B', s_p \leq 1 \quad (6)$$

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

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

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

ILP-Model (2)

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

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

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

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

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

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

ILP-Model (3)

- Calculate makespan from the variables $c_{i,p,k,t}$

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

- Minimize makespan τ:

$$\min \ \tau$$

- Problem specified in AMPL www.ampl.com
- ILP Solver (CPLEX, Gurobi, lp_solve, ...)

Results (7) – DP vs. ILP (AMPL+CPLEX)

- TI C64x DSP Benchmarks and FreeBench

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

Observations:

On the average,
DP can solve larger problem instances
but ILP is often faster (where it terminates).

DP works better for deep dependence chains.

Selected Project Publications

- Christoph Kessler, Andrzej Bednarski:
  Optimal integrated code generation for VLIW architectures.
  Concurrency and Computation: Practice and Experience **18**: 1353-1390, 2006.
- Andrzej Bednarski, Christoph Kessler:
  Optimal Integrated VLIW Code Generation with Integer Linear Programming.
  Proc. Euro-Par 2006 conference, Springer LNCS 4128, pp. 461-472, Aug. 2006.
- Andrzej Bednarski:
  Optimal Integrated Code Generation for Digital Signal Processors.
  PhD thesis, Linköping university - Institute of Technology, Linköping, Sweden, June 2006.
- Christoph Kessler, Andrzej Bednarski, Mattias Eriksson:
  Classification and generation of schedules for VLIW processors.
  Concurrency and Computation: Practice and Experience **19**: 2369-2389, 2007.
- Mattias Eriksson, Christoph Kessler:
  Integrated Modulo Scheduling for Clustered VLIW Architectures.
  Proc. HiPEAC-2009 High-Performance and Embedded Architecture and Compilers, Paphos, Cyprus, Jan. 2009. Springer LNCS 5409, pp. 65-79.
- Mattias Eriksson: Integrated Code Generation. PhD thesis, Linköping Univ., 2011
- www.ida.liu.se/~chrke/optimist