TDDB29 Compilers and Interpreters (opt.)
TDDB44 Compiler Construction

Code Generation for RISC and Instruction-Level Parallel Processors

RISC/ILP Processor Architecture Issues
Instruction Scheduling
Register Allocation
Phase Ordering Problems
Integrated Code Generation

Christoph Kessler, IDA,
Linköpings universitet, 2007.

TDDB29 Compilers and Interpreters (opt.)
TDDB44 Compiler Construction

1. RISC and Instruction-Level Parallel Target Architectures

Christoph Kessler, IDA,
Linköpings universitet, 2007.

CISC vs. RISC

CISC
- Complex Instruction Set Computer
- Memory operands for arithmetic and logical operations possible
- M(r1+r2) ← M(r1+r2) * M(r3+disp)
- Many instructions
- Complex instructions
- Few registers, not symmetric
- Variable instruction size
- Instruction decoding (often done in microcode) takes much silicon overhead
- Example: 80x86, 680x0

RISC
- Reduced Instruction Set Computer
- Arithmetic/logical operations only on registers
- add r1, r2, r1
  load r1, r4
  load r3+disp, r5
  mul r4, r5
  store r5, r1
- Few, simple instructions
- Many registers, all general-purpose
  typ. 32 ... 256
- Fixed instruction size and format
- Instruction decoding hardwired
- Example: POWER, HP-PA RISC, MIPS, ARM, SPARC

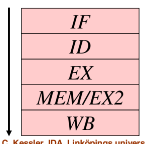
Instruction-Level Parallel (ILP) architectures

Single-Issue: (can start at most one instruction per clock cycle)
- Simple, pipelined RISC processors with one or multiple functional units
  - e.g. ARM9E, DLX

Multiple-Issue: (can start several instructions per clock cycle)
- Superscalar processors
  - e.g. Sun SPARC, MIPS R10K, Alpha 21264, IBM Power2, Pentium
- VLIW processors
  - e.g. Multiflow Trace, Cydrome Cydra-5, Intel i860,
    HP Lx, Transmeta Crusoe;
    most DSPs, e.g. Philips Trimedia TM32, TI 'C6x
- EPIC processors
  - e.g. Intel Itanium family (IA-64)

Pipelined RISC Architectures

- A single instruction is issued per clock cycle
- Possibly several parallel functional units / resources
- Execution of different phases of subsequent instructions overlaps in time. This makes them prone to:
  - data hazards (may have to delay op until operands ready),
  - control hazards (may need to flush pipeline after wrongly predicted branch),
  - structural hazards (required resource(s) must not be occupied)
- Static scheduling (insert NOPs to avoid hazards) vs. Run-time treatment by pipeline stalling



issue	cycle	*PM*	Decoder	ALU_1	DM/ALU_2	Regs
I_1	1	IF_1				
I_2	2	IF_2	ID_1			
I_3	3	IF_3	ID_2	EX_1		
I_4	4	IF_4	ID_3	EX_2	MEM_1	
I_5	5	IF_5	ID_4	EX_3	MEM_2	WB_1
I_6	6	IF_6	ID_5	EX_4	MEM_3	WB_2
...						

Reservation Table, Scheduling Hazards

add:

			ALU		MULTIPLIER				
Time	read src1 opnd	read src2 opnd	stage 0	stage 1	stage 0	stage 1	stage 2	stage 3	write result bus
0	X	X							
1			X						
2				X					
3									X

mul:

			ALU		MULTIPLIER				
Time	read src1 opnd	read src2 opnd	stage 0	stage 1	stage 0	stage 1	stage 2	stage 3	write result bus
0	X	X							
1					X				
2						X			
3							X		
4								X	
5									X

Reservation table
specifies required resource occupations

[Davidson 1975]

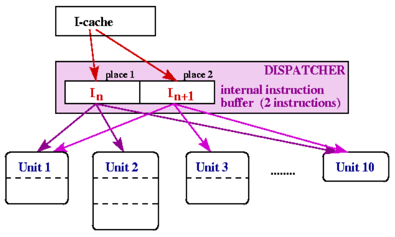
```
t:    mul ...
t+1:  ...
t+2:  add ...
...
```

structural hazard at t=5

Superscalar processor

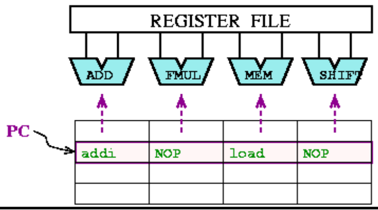
- Run-time scheduling by instruction dispatcher
  - convenient (sequential instruction stream – as usual)
  - limited look-ahead buffer to analyze dependences, reorder instr.
  - high silicon overhead, high energy consumption
- Example: Motorola MC 88110

  2-way, in-order issue superscalar



VLIW (Very Long Instruction Word) architectures

- Multiple slots for instructions in long instruction-word
  - Direct control of functional units and resources – low decoding OH
- Compiler (or assembler-level programmer) must determine the schedule statically
  - independence, unit availability, packing into long instruction words
  - Challenging! But the compiler has more information on the program than an on-line scheduler with a limited lookahead window.
  - Silicon- and energy-efficient



EPIC architectures

- Based on VLIW
- Compiler groups instructions to LIW's (bundles)
- Compiler takes care of resource and latency constraints
- Compiler marks sequences of independent instructions
- Dynamic scheduler assigns resources and reloads new bundles as required

TDDB29 Compilers and Interpreters (opt.)

TDDB44 Compiler Construction

2. Instruction Scheduling

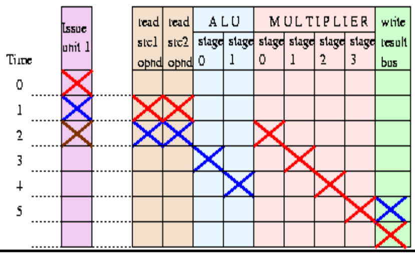
Christoph Kessler, IDA,
Linköpings universitet, 2007.

Instruction Scheduling (1)

- Map instructions to time slots on issue units (and resources), such that no hazards occur

  → ***Global reservation table, resource usage map***

```
t:   mul ...
t+1: add ...
t+2: nop ...
       ...
```



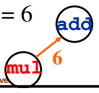
Instruction Scheduling (2)

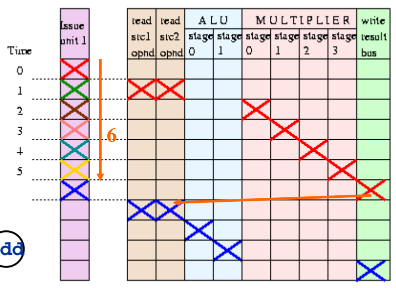
- Data dependences imply **latency constraints**

  → target-level data flow graph / data dependence graph

```
t:   mul R1,...
t+1: nop ...
t+2: nop ...
t+3: nop ...
t+4: nop ...
t+5: nop ...
t+6: add ...,R1
```

latency(mul) = 6

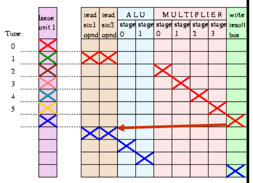




Instruction Scheduling

Generic Resource model
- Reservation table

```
t:   mul R1,...
t+1: nop ...
t+2: nop ...
t+3: nop ...
t+4: nop ...
t+5: nop ...
t+6: add ...,R1
```



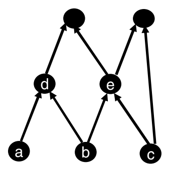
Local Scheduling
(f. Basic blocks / DAGs)
- Data dependences
  → Topological sorting
  - List Scheduling
    (diverse heuristics)

Global Scheduling
- Trace scheduling, Region scheduling, ...
- Cyclic scheduling (Software pipelining)

There exist **retargetable schedulers** and **scheduler generators**, e.g. for GCC since 2003

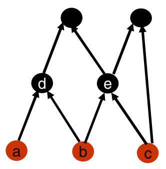
Example: Topological Sorting (0)

- ● Not yet considered
- ● Data ready (zero-indegree set)
- ○ Already scheduled, still alive
- ○ Already scheduled, no longer referenced



Example: Topological Sorting (1)

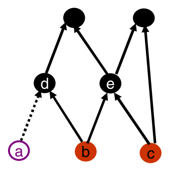
- ● Not yet considered
- ● Data ready (zero-indegree set)
- ○ Already scheduled, still alive
- ○ Already scheduled, no longer referenced



a

Example: Topological Sorting (2)

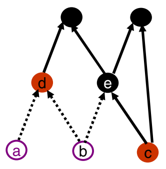
- ● Not yet considered
- ● Data ready (zero-indegree set)
- ○ Already scheduled, still alive
- ○ Already scheduled, no longer referenced



a b

Example: Topological Sorting (3)

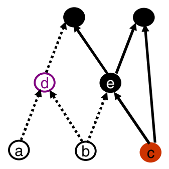
- ● Not yet considered
- ● Data ready (zero-indegree set)
- ○ Already scheduled, still alive
- ○ Already scheduled, no longer referenced



a b d

Example: Topological Sorting (4)

- ● Not yet considered
- ● Data ready (zero-indegree set)
- ○ Already scheduled, still alive
- ○ Already scheduled, no longer referenced



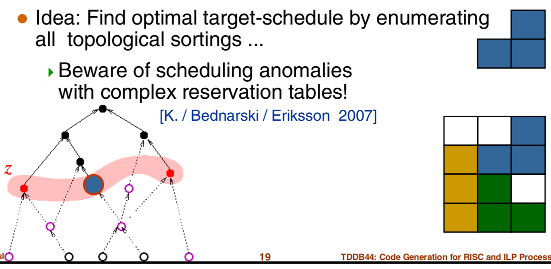
a b d

and so on...

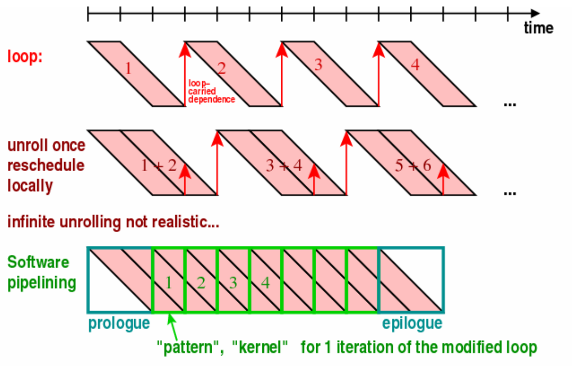
Topological Sorting and Scheduling

- Construct schedule incrementally in topological (= causal) order
  - "Appending" instructions to partial code sequence: close up in target schedule reservation table (as in "Tetris")
  - Idea: Find optimal target-schedule by enumerating all topological sortings ...
    - Beware of scheduling anomalies with complex reservation tables!

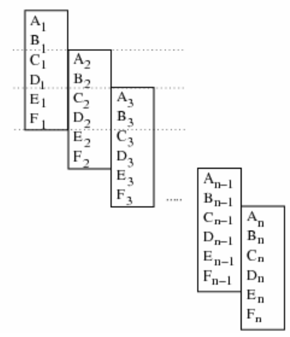
[K. / Bednarski / Eriksson 2007]



Software Pipelining of Loops (1)


time
loop:
1
2
3
4
loop-carried dependence
...
unroll once reschedule locally
1 + 2
3 + 4
5 + 6
...
infinite unrolling not realistic...
Software pipelining
1
2
3
4
prologue
epilogue
"pattern", "kernel" for 1 iteration of the modified loop

Software Pipelining of Loops (2)

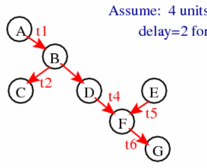


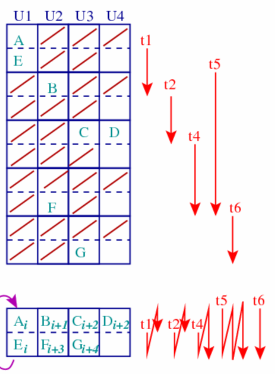
	Unit1	Unit2	Unit3
Prologue:	A_1		
	B_1		
	C_1	A_2	
	D_1	B_2	
	DO i=1, n–2		
Pattern:	E_i	C_{i+1}	A_{i+2}
	F_i	D_{i+1}	B_{i+2}
Epilogue:		E_{n-1}	C_n
		F_{n-1}	D_n
			E_n
			F_n

→ More about Software Pipelining in TDDC86 *Compiler Optimizations and Code Generation*

Software Pipelining of Loops (3) Modulo Scheduling

Assume: 4 units, fully pipelined
delay=2 for all instructions





No dependence cycles
ResMII = ceil(7/4) = 2

Begin with II = ResMII = 2

Apply some local scheduling heuristic
e.g.: list scheduling (A B C D E F G)
Apply some placement heuristic
e.g.: as early as possible
Mark occupied slots in all iterations...

If not possible, increase II and try again...

TDDB29 Compilers and Interpreters (opt.)
TDDB44 Compiler Construction

3. Register Allocation

Christoph Kessler, IDA, Linköpings universitet, 2007.

Register Allocation

- **Register Allocation**: Determines values (variables, temporaries, constants) to be kept when in registers
- **Register Assignment**: Determine in which physical register such a value should reside.

- Essential for Load-Store Architectures
- Reduce memory traffic (→ memory / cache latency, energy)
- Limited resource
- Values that are alive simultaneously cannot be kept in the same register
- Strong interdependence with instruction scheduling
  - scheduling determines live ranges
  - spill code needs to be scheduled

- **Local register allocation** (for a single basic block) can be done in linear time (see function getreg() above).
- **Global register allocation** (with minimal spill code) is NP-complete. Can be modeled as a graph coloring problem [Ershov'62] [Cocke'71].

Local Register Allocation

For variable v and basic block B_i:

$$netsave(v,i) = \#uses_i \cdot usesave \quad + \; \#defs_i \cdot defsave$$
$$- \; l \cdot ldcost \qquad (l = 1 \text{ if Load}(v) \text{ needed at beg. of } B_i, \text{ 0 otherw.})$$
$$- \; s \cdot stcost \qquad (s = 1 \text{ iff Store}(v) \text{ needed at end of } B_i, \text{ 0 otherw.})$$

For loop L estimate benefit of keeping v in a register:

$$benefit(v,L) = 10^{depth(L)} \cdot \sum_{i \in blocks(L)} netsave(v,i)$$

with R registers available:
allocate the R objects v with greatest benefit in L

moves may be necessary instead of Load(v) / Store(v)
if v could reside in (different) registers in $Pred(B_i)$, B_i, $Succ(B_i)$

add worst-case terms $|Pred(v)| \cdot mvcost$, $|Succ(v)| \cdot mvcost$

Live range

(Here, variable = program variable or temporary)

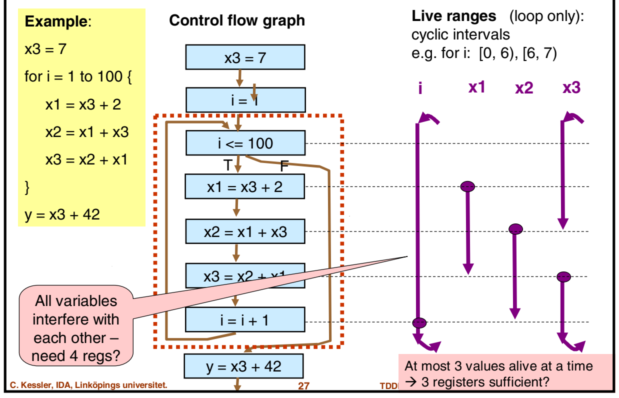
- A variable is being **defined** at a program point if it is written (given a value) there.
- A variable is **used** at a program point if it is read (referenced in an expression) there.
- A variable is **alive** at a point if it is referenced there or at some following point that has not (may not have) been preceded by any definition.
- A variable is **reaching** a point if an (arbitrary) definition of it, or usage (because a variable can be used before it is defined) reaches the point.
- A variable's **live range** is the area of code (set of instructions) where the variable is both alive and reaching.
  - does not need to be consecutive in program text.

Register Allocation for Loops

Example:

```
x3 = 7
for i = 1 to 100 {
    x1 = x3 + 2
    x2 = x1 + x3
    x3 = x2 + x1
}
y = x3 + 42
```

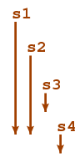
Live ranges (loop only):
cyclic intervals
e.g. for i: [0, 6), [6, 7)


Control flow graph
x3 = 7
i = 1
i <= 100
T
F
x1 = x3 + 2
x2 = x1 + x3
x3 = x2 + x1
i = i + 1
y = x3 + 42
i
x1
x2
x3
All variables interfere with each other – need 4 regs?
At most 3 values alive at a time → 3 registers sufficient?

Register Allocation by Graph Coloring

- **Step 1:** Given a program with symbolic registers s1, s2, ...
  - Determine live ranges of all variables

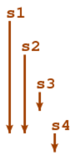
```
i = c+4;  load 8(fp),s1   ! c
          nop
          addi s1,#4,s2
          store s2,4(fp)  ! i
d = c-2;  subi s1,#2,s3
          store s3,12(fp) ! d
c = c*i;  muli s1,s2,s4
          store s4,8(fp)  ! c
```

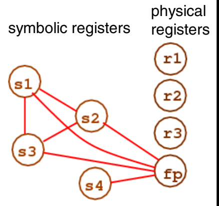


Register Allocation by Graph Coloring

- **Step 2:** Build the **Register Interference Graph**
  - Undirected edge connects two symbolic registers (s*i*, s*j*) if live ranges of s*i* and s*j* overlap in time
  - Reserved registers (e.g. fp) interfere with all s*i*

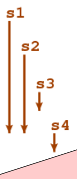
```
i = c+4;  load 8(fp),s1   ! c
          nop
          addi s1,#4,s2
          store s2,4(fp)  ! i
d = c-2;  subi s1,#2,s3
          store s3,12(fp) ! d
c = c*i;  muli s1,s2,s4
          store s4,8(fp)  ! c
```

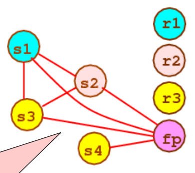




- **Step 3**: Color the register interference graph with *k* colors, where *k* = #available registers.
  - If not possible: pick a victim s*i* to spill, generate spill code (store after def., reload before use)
    - This may remove some interferences. Rebuild the register interference graph + repeat Step 3...

```
i = c+4;  load 8(fp),s1   ! c
          nop
          addi s1,#4,s2
          store s2,4(fp)  ! i
d = c-2;  subi s1,#2,s3
          store s3,12(fp) ! d
c = c*i;  muli s1,s2,s4
          store s4,8(fp)  ! c
```





This register interference graph cannot be colored with less than 4 colors, as it contains a 4-clique

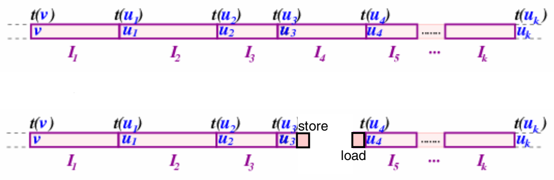
Coloring a graph with *k* colors

- NP-complete for k > 3
- **Chromatic number** γ(G) = minimum number of colors to color a graph G
- γ(G) >= c if the graph contains a c-clique
  - A *c*-**clique** is a completely connected subgraph of *c* nodes
- **Chaitin's heuristic** (1981):

```
S ← { s1, s2, ... }    // set of spill candidates
while  ( S not empty )
    choose some  s  in  S.
    if  s  has less than k neighbors in the graph
        then  // there will be some color left for s:
              delete  s (and incident edges) from the graph
        else  modify the graph  (spill, split, coalesce ... nodes)
              and restart.
// once we arrive here, the graph is empty:
color the nodes greedily in reverse order of removal.
```

Live range splitting

- Instead of spilling completely (reload before each use), it may be sufficient to split a live range at one position where register pressure is highest
  - save, and reload once



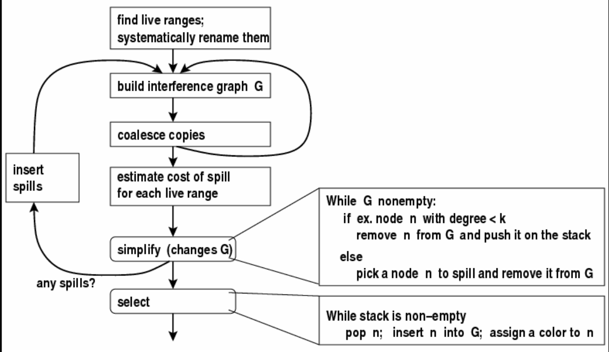
Live range coalescing

- For a copy instruction *sj* ← *si*
  - where *si* and *sj* do not interfere
  - and *si* and *sj* are not rewritten after the copy operation
- Merge *si* and *sj*:
  - patch (rename) all occurrences of *si* to *sj*
  - update the register interference graph
- and remove the copy operation.

```
s2 ← ...
...
s3 ← s2
...
... s3 ...
```

```
s3 ← ...
...
~~s3 ← s3~~
...
... s3 ...
```

Chaitin's Register Allocator (1981)



find live ranges; systematically rename them
build interference graph G
coalesce copies
estimate cost of spill for each live range
simplify (changes G)
select
insert spills
any spills?

```
While  G  nonempty:
   if  ex. node  n  with degree < k
      remove  n  from G  and push it on the stack
   else
      pick a node  n  to spill and remove it from G
```

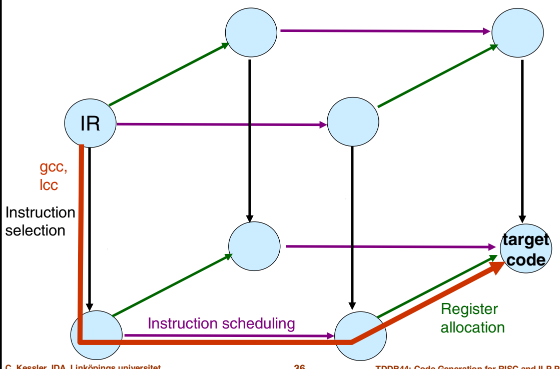
```
While stack is non–empty
   pop  n;   insert  n  into  G;  assign a color to  n
```

TDDB29 Compilers and Interpreters (opt.)
TDDB44 Compiler Construction

4. Phase Ordering Problems and Integrated Code Generation

Christoph Kessler, IDA, Linköpings universitet, 2007.

Phase ordering problems

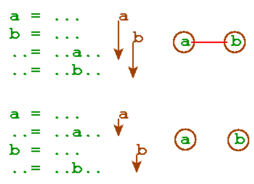


Phase ordering problems (1)

Instruction scheduling vs. register allocation

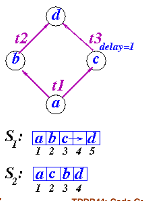
(a) Scheduling first:
determines Live-Ranges
→ Register need,
possibly spill-code to be inserted afterwards

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



```
a = ...
b = ...
..= ..a..
..= ..b..

a = ...
..= ..a..
b = ...
..= ..b..
```



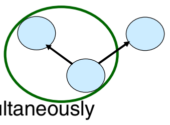
Phase ordering problems (2)

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

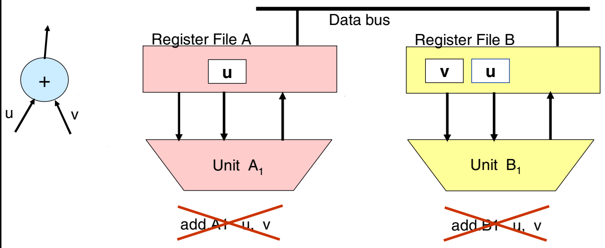
  → Integration with instruction scheduling desirable



- **Mutations** with different resource requirements
  - a = 2 * b or a = b << 1 or a = b + b ?
- Different instructions with different register need

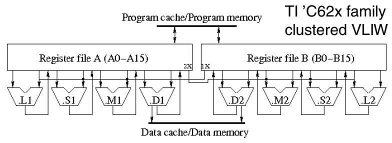
Clustered VLIW processor

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



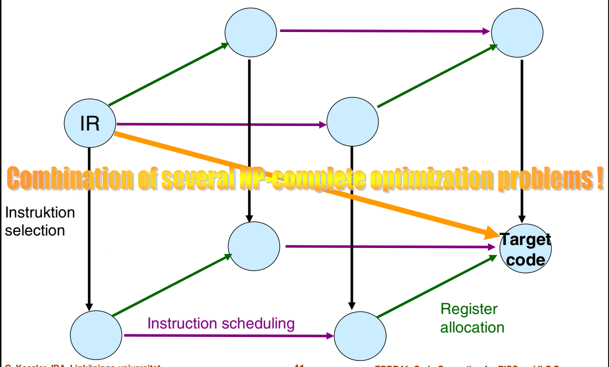
More phase ordering problems

TI 'C62x family
clustered VLIW DSP processor



- 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

5. Integrated Code generation

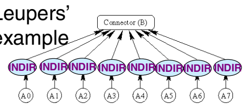


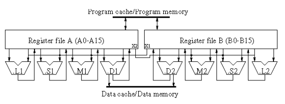
Integrated Code Generation Related Work

- Heuristic Phase Interleaving
  [Goodman/Hsu '88], [Leupers'00], ...
- Combination of 2 phases
  - Instruktion selection and register allocation
    - DP 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 e.g. [Kästner'00]
- Combination of 3 phases
  - ILP [Wilson et al.'94]
    - only for simple, non-pipelined RISC processor

Results (2) – DP

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

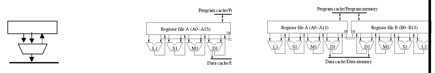




TI-C comp.	Schedule by Leupers' heuristic	Optimal schedule by OPTIMIST
LD *A4,B4	LD *A0,A8 || MV A1,B8	LD *A0,A8 || MV A1,B8
LD *A1,A8	LD *B8,B1 || LD *A2,A9 || MV A3,B10	LD *B8,B1 || LD *A2,A9 || MV A3,B10
LD *A3,A9	LD *B10,B3 || LD *A4,A10 || MV A5,B12	LD *B10,B3 || LD *A4,A10 || MV A5,B12
LD *A0,B0	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 (4) – Optimal integrated code generation, OPTIMIST – DP algorithm

- Single-issue vs. Single-cluster vs. Double-cluster Architecture



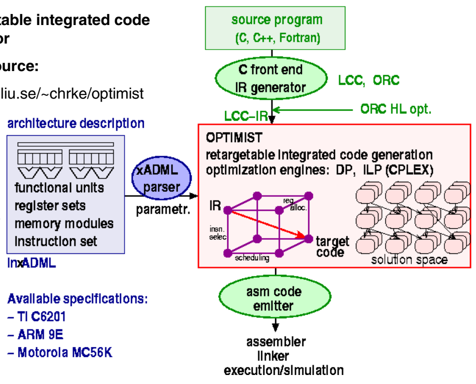
			ARM9E-ARM mode			TI-C62x single-cluster			TI-C62x		
benchmark	BB	size	t[s]	#merged	#ESnodes	t[s]	#merged	#ESnodes	t[s]	#merged	#ESnodes
analyzer	51	34	98.9	546250	62965	12623.2	2547538	630132	—	—	—
analyzer	203	31	56.5	309673	49833	14.1	34559	29780	—	—	—
bmmse	4	14	0.1	489	165	0.1	523	375	10.9	15215	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	24	42	178.5	783854	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	—	—	—

Our project: OPTIMIST

Retargetable integrated code generator

Open Source:

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



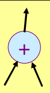
Processor specification language xADML

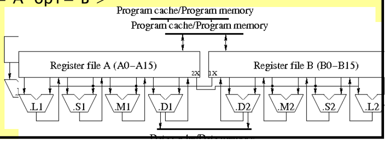
```
<architecture omega="8">
  <registers> ... </registers>
  <residenceclasses> ... </residenceclasse
  <funits> ... </funits>
  <patterns> ... </patterns>
  <instruction_set>
    <instruction id="ADDP4" op="4407">
      <target id="ADD  .L1" op0="A" op1="A" op2="A" use_fu="L1"/>
      <target id="ADD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





Project Literature (Selection)

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