VU2 185.324

Compilation Techniques for VLIW Architectures

Dietmar Ebner ebner@complang.tuwien.ac.at
Florian Brandner brandner@complang.tuwien.ac.at

http://complang.tuwien.ac.at/cd/vliw

Last Lectures (1)

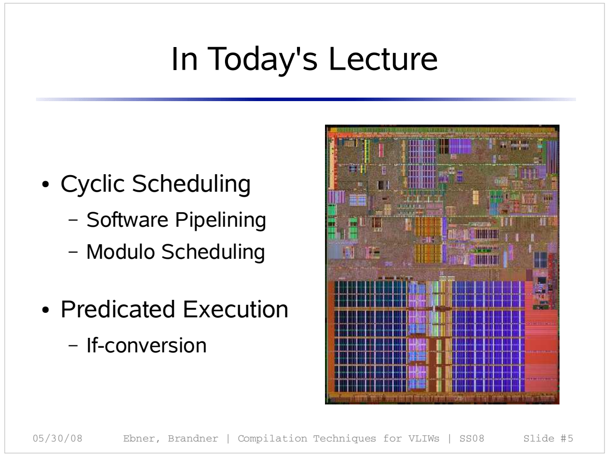
- Instruction scheduling
  - List scheduling
  - Classification
    - Regime, search strategy, region shapes
- Region formation
    - Mutual most likely strategy

Last Lectures (2)

- Region Types
  - Linear
    - Traces / Superblocks
  - Non linear
    - Hyperblocks / Treegion / Trace 2
  - SEME vs MEME
- Details on Superblocks
  - Moderate compile time penalty
  - Major performance improvements
  - Sometimes severe code size increase

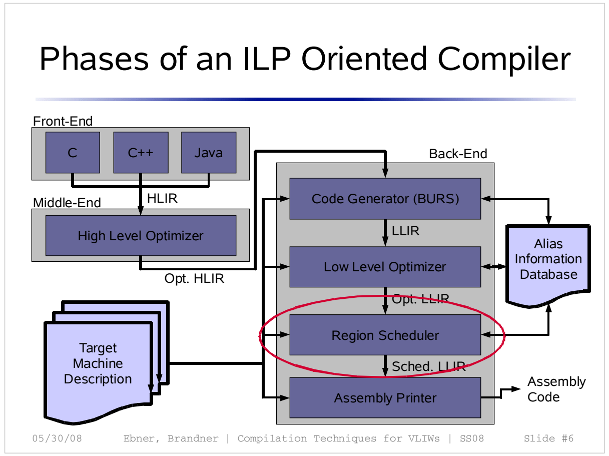
Last Lectures (3)

- Region enlargement
  - Loop unrolling
  - Loop peeling
  - Tail duplication
  - If-conversion (Hyperblocks)
- Dependence elimination
  - Renaming
  - Induction variable / accumulator expansion
  - Operation combining and migration

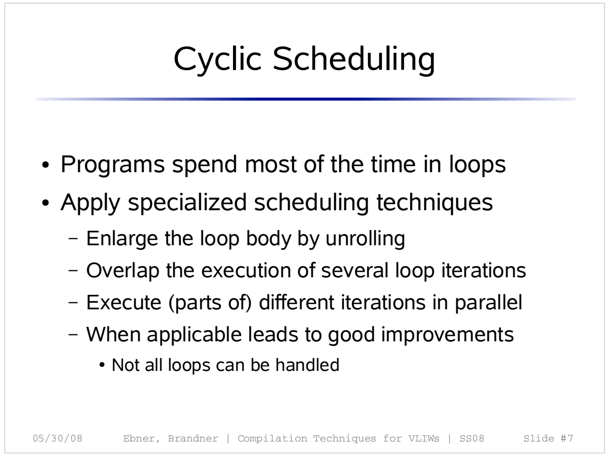


In Today's Lecture

- Cyclic Scheduling
  - Software Pipelining
  - Modulo Scheduling
- Predicated Execution
  - If-conversion

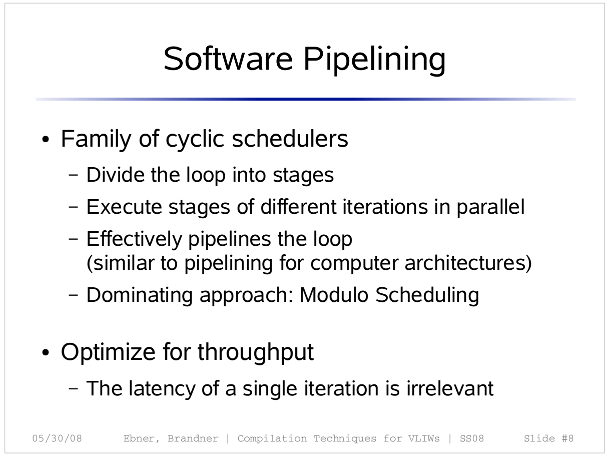


Phases of an ILP Oriented Compiler



Cyclic Scheduling

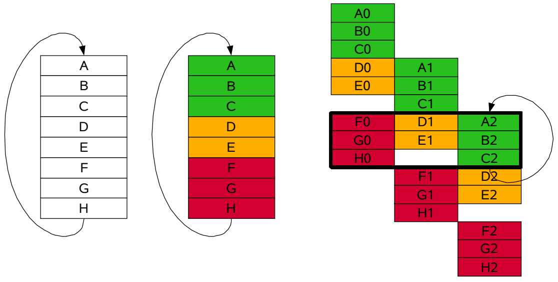
- Programs spend most of the time in loops
- Apply specialized scheduling techniques
  - Enlarge the loop body by unrolling
  - Overlap the execution of several loop iterations
  - Execute (parts of) different iterations in parallel
  - When applicable leads to good improvements
    - Not all loops can be handled



Software Pipelining

- Family of cyclic schedulers
  - Divide the loop into stages
  - Execute stages of different iterations in parallel
  - Effectively pipelines the loop
    (similar to pipelining for computer architectures)
  - Dominating approach: Modulo Scheduling
- Optimize for throughput
  - The latency of a single iteration is irrelevant

Example: Software Pipelining (1)

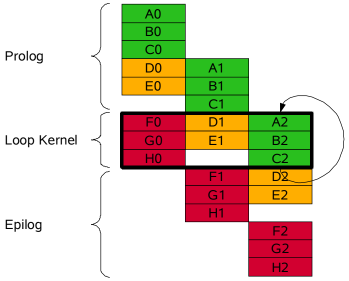


a) original loop

b) divide into stages

c) pipelined loop

Example: Software Pipelining (2)



Modulo Scheduling (1)

- Important SW Pipelining Technique
  - Explores the space of possible loop kernels
  - Initiation interval (II)
    - Constant interval between the start of successive kernel iterations
  - Lower bound of II (MinII)
    - Consider available hardware resources (ResII)
    - Data dependencies and recurrences (RecII)
  - Upper bound of II (MaxII)
    - Length of a linear schedule

Modulo Scheduling (2)

- Search for an II starting from MinII
  - If a valid schedule could be found stop
  - Otherwise decide whether to
    - Backtrack - revert some scheduling decision
    - Increase the II
    - Abort if II is larger than MaxII
- High computational complexity
  - Backtracking and searching the II

Scheduling Heuristics

- Modified version of list scheduling
  - Employ the modulo reservation table (MRT)
    - Similar to regular reservation tables
    - Ensure that a resource is not used at the same cycle, or at following cycles that modulo the II collide with it
  - Allows backtracking to revert scheduling decisions
    - Limited by a backtracking budged
    - May cause significant overhead

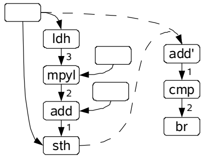
Example: Modulo Scheduling

```
int i,x ;
short a[];
for (i=0; i < 100; ++i)
  a[i+3] = a[i]*x + 7;
```

(a) Original C Code

	ALU0	ALU1	MUL0	MEM	BR
0	cmp (9)	---	mpyl (3)	ldh (0)	---
1	---	---	---	sth (7)	---
2	add (5)	add' (8)	---	---	br (11)

(c) Modulo Reservation Table



(b) Data Dependence Graph

Cycle mod II	Cycle				
0	0	ldh			
1	1				
2	2				
0	3	mpyl	ldh		
1	4				
2	5	add			
0	6		mpyl	ldh	
1	7	sth			
2	8	add'	add		
0	9	cmp		mpyl	ldh
1	10		sth		
2	11	br	add'	add	
0	12		cmp		mpyl
1	13			sth	
2	14		br	add'	add
0	15			cmp	
1	16				sth

(d) Modulo Schedule

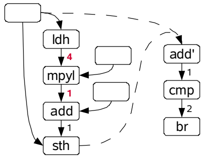
Prolog / Epilog

- Partial copies of the loop kernel
  - May cause some code size increase
  - Strongly depending on the number of stages
- Several versions required in the presence of multiple loop exits and uncertain trip counts
- May harm runtime on small trip counts
  - Loop kernel is never reached (steady state)

Variable Names

- MS increases the register pressure
  - Preserve multiple copies of the same variable for different iterations
    - Life ranges exceeding the II
  - Spilling may disrupt the compact schedule
- Solutions
  - Modulo variable expansion (loop unrolling)
  - Copy operations
    - Hardware support (rotating register files)

Example: Variable Names (1)



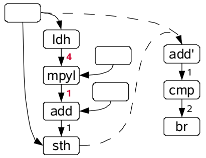
(a) Data Dependence Graph

Cycle mod II	Cycle				
0	0	ldh			
1	1				
2	2				
0	3		ldh		
1	4	mpyl			
2	5	add			
0	6			ldh	
1	7	sth	mpyl		
2	8	add'	add		
0	9	cmp			ldh
1	10		sth	mpyl	
2	11	br	add'	add	
0	12		cmp		
1	13			sth	mpyl
2	14		br	add'	add
0	15			cmp	
1	16				sth

(b) Modulo Schedule

- Minor change in the schedule
- The value of *ldh* exceeds the II
  - *mpyl still requires the value of the last iteration*

Example: Variable Names (2)



(a) Data Dependence Graph

Cycle mod II	Cycle				
0	0	ldh			
1	1				
2	2				
0	3		ldh		
1	4	mpyl			
2	5	add			
0	6			ldh	
1	7	sth	mpyl		
2	8	add'	add		
0	9	cmp		copy	ldh
1	10		sth	mpyl	
2	11	br	add'	add	
0	12		cmp		
1	13			sth	mpyl
2	14		br	add'	add
0	15			cmp	
1	16				sth

(b) Modulo Schedule

- *Solution*
  - *Shown here: A extra copy operation*
  - *Alternative: Modulo Variable Expansion*

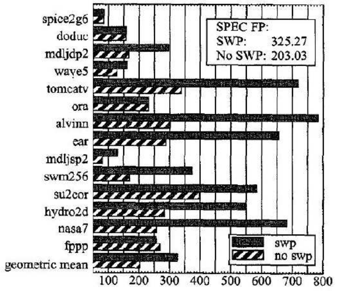
Modulo Variable Expansion

- Unrolling increases the II
  - It is always possible to find a suitable unrolling factor to prevent copy operations
    - Assume v *is the longest lifetime of a variable*
    - *The unrolling factor is given by* $k = \lceil \frac{v}{II} \rceil$
- Larger kernel, prolog, and epilog
- Complicates handling of loop exits

Limitations of Modulo Scheduling

- Internal control flow
  - No internal control flow permitted
    - Applying if-conversion helps
  - Early loop exits are complex to handle
- Nested Loops
  - Recursively invoke the modulo scheduler starting with the innermost loop
- Low trip counts
  - Possibly never reach the steady state
  - Two versions unrolling & pipelined loop (code bloat)

Software Pipelining on a MIPSPro

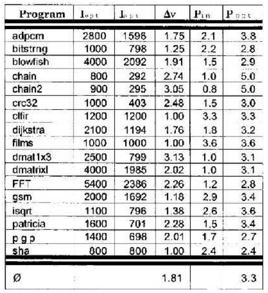


Source: J. Ruttenberg, SGI

- SPECfp92 on a MIPSPro (R8000, 75Mhz)
- SPECmarks with pipelining enabled/disabled

Side note: Papers on software pipelining typically do not show complete benchmarks, but concentrate on single loops only.

Software Pipelining on a VLIW



Program	I_{opt}	I_{spt}	Δv	P_{in}	P_{opt}
adpcm	2800	1596	1.75	2.1	3.8
bitstrng	1000	798	1.25	2.2	2.8
blowfish	4000	2092	1.91	1.5	2.9
chain	800	292	2.74	1.0	5.0
chain2	900	295	3.05	0.8	5.0
crc32	1000	403	2.48	1.5	3.0
clfir	1200	1200	1.00	3.3	3.3
dijkstra	2100	1194	1.76	1.8	3.2
films	1000	1000	1.00	3.6	3.6
dmat1x3	2500	799	3.13	1.0	3.1
dmatrixl	4000	1985	2.02	1.0	3.1
FFT	5400	2386	2.26	1.2	2.8
gsm	2000	1692	1.18	2.9	3.4
isqrt	1100	796	1.38	2.6	3.6
patricia	1600	701	2.28	1.5	3.4
p g p	1400	698	2.01	1.7	2.7
sha	800	800	1.00	2.4	2.4
Ø			1.81		3.3

Source: D. Kästner, AbsInt

- TriMedia TM1000
  - 5 issue VLIW
  - 100 Mhz
- Benchmarks
  - DSPStone
  - MiBench

Side note: Papers on software pipelining typically do not show complete benchmarks, but concentrate on single loops only.

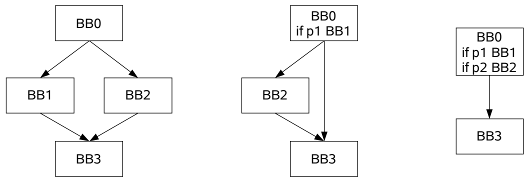
Predicated Execution

- Predicated operations
  - Conditionally nullify the result of the operation
  - Partial vs. full predication
    - All operations can be predicated
    - Conditional move or select operations
- Enable elimination of branches
  - Increase available ILP

If-Conversion

- Restructure the CFG
  - Augment blocks with predicates
  - Merge blocks
  - Eliminate branches
- Additional benefits
  - Increase scope for schedulers (Hyperblocks)
  - Enable software pipelining (internal control flow)
  - Other optimizations may benefit as well

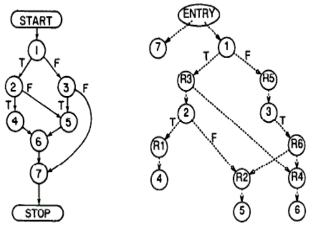
Example: If-Conversion



If-Conversion

- Simple approaches
  - Select a candidate basic block following a heuristic
    - Often following simple patterns (e.g., if-then-else, etc.)
  - Calculate a predicate
  - Predicate all instructions of the block
- More Sophisticated
  - Use the Program Dependence Graph (PDG)
  - Try to place predicate definitions optimally
  - Try to derive a minimal set of predicates

Program Dependence Graph

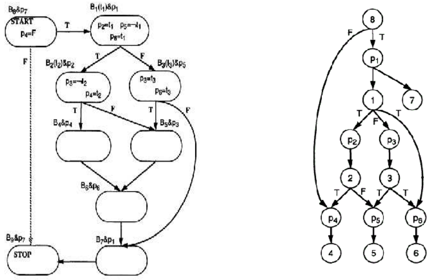


a) Regular CFG

b) Derived PDG

Source: J .Ferrante, HP

If-conversion using RK*



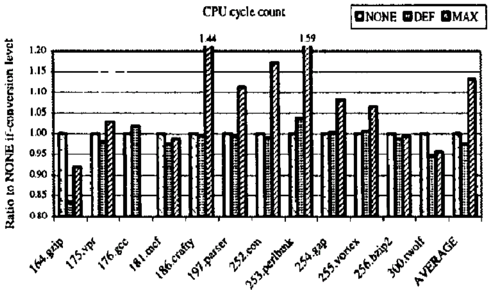
a) CFG

b) Derived Predicates

* R and K are functions on the control dependence graph, introduced by Park et. al.

Source: J. Park, HP

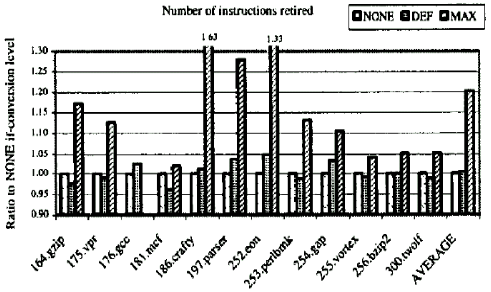
If-conversion on Itanium (1)



* SPECINT2000, running on a near-production silicon

Source: Y. Choi, Intel

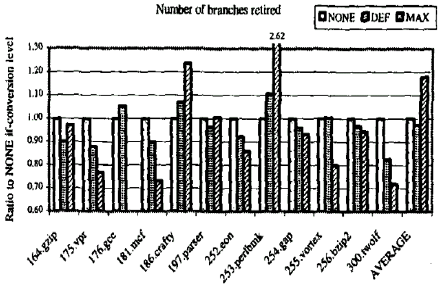
If-conversion on Itanium (2)



* SPECINT2000, running on a near-production silicon

Source: Y. Choi, Intel

If-conversion on Itanium (3)

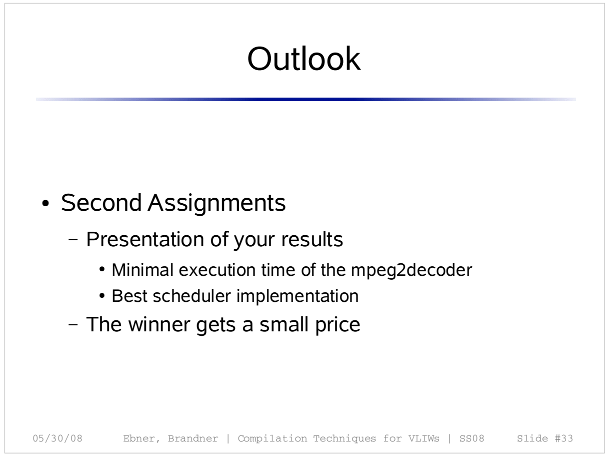


* SPECINT2000, running on a near-production silicon

Source: Y. Choi, Intel

Limitation of If-Conversion

- Relies on hardware support
  - Non predicatable instructions cause problems
- Aggressive application
  - Causes slow-down, increase of code size, and power consumption
  - Estimating profitability is hard
  - Reverse-if-conversion
    - Undo if-conversion if unprofitable



Outlook

- Second Assignments
  - Presentation of your results
    - Minimal execution time of the mpeg2decoder
    - Best scheduler implementation
  - The winner gets a small price